Efficient DDD-based Term Generation Algorithm for Analog Circuit Behavioral Modeling

Sheldon X.-D. Tan
Department of Electrical Engineering
University of California, Riverside, CA 92521, USA
stan@ee.ucr.edu

C.-J. Richard Shi
Department of Electrical Engineering
University of Washington, Seattle, WA 98195, USA
shi@ee.washington.edu

Abstract— An efficient approach to generating symbolic product terms for behavioral modeling of large linear analog circuits is presented. The approach is based on a compact determinant decision diagram (DDD) representation of transfer functions and characteristics of analog circuits. The new algorithm is based on the concept that a dominant term in a DDD graph can be found by searching the shortest path in the graph. But instead of traversing a whole DDD graph each time, we show that a shortest path can be found by just updating a small number of the newly added vertices after the first shortest path is found. Experimental results indicate that the new symbolic term generation algorithm outperforms both pure shortest path based algorithm and dynamic programming based algorithm, which is the fastest symbolic term generation algorithm published so far.

I. Introduction

Behavioral modeling aims at generating compact and simulation ready models for analog circuit blocks that capture the circuit characteristics of interests. Behavioral models can be used to speed up full system design analysis and verification. Behavioral modeling is a critical technique for emerging system-on-a-chip (SoC) designs as efficient implementations of analog building blocks in SoC systems becomes increasingly important. One way to derive behavioral models of analog modules is by means of symbolic analysis. As illustrated in [3], simple yet accurate symbolic expressions can also be interpretable by analog designers to gain the insight into circuit behavior, performance and stability, and are important for many applications in circuit design such as transistor sizing and optimization, topology selection, sensitivity analysis, fault simulation, testability analysis and yield enhancement [4].

Research on symbolic analysis can date back to 1960s [6]. Recently, various schemes to drive approximate symbolic expressions have been developed. Approximation can be carried out after generation of all the symbolic product terms [3, 11, 16]; or during generation [2, 15, 17] and even before generation [5, 17].

Recently Tan and Shi proposed an efficient DDD graph based method of deriving simple yet accurate symbolic expressions for behavioral modeling of linear(ized) analog circuits [12]. Their DDD-based approximation method has both the reliability in the approximation-after-generation methods [3, 11, 16], and the capability in approximation-before [5, 17]/during-generation [2, 15, 17] methods for analyzing large analog circuits.

In this paper, we consider how to obtain the dominant product terms from exact or simplified DDD graphs representing circuit characteristics. An efficient algorithm was proposed in [12] where finding a dominant term is transformed into searching the shortest path in a DDD graph. Once the shortest path is found, it can be *subtracted* from the DDD graph by simple DDD graph operations. The next dominant term can be found on the resulting DDD graph in the same way. Recently, Verhaegen and Gielen presented another DDD-based dominant-term generation algorithm, which is based on dynamic programming concept [14]. It is shown in [13] that the dynamic programming based algorithm is faster than the shortest path based algorithm in general but at cost of more memory use.

In this paper, we present a more efficient shortest path based algorithm for dominant term generation. The success of the new algorithm is based on the observation that if the source vertex in a DDD graph are properly defined, a shortest path can be found by just updating a small number of the newly added vertices after the first shortest path is found. Experimental results show that our incremental shortest path based algorithm outperforms both the pure shortest path based algorithm and the dynamic programming based algorithm for different types of analog circuits.

This paper is organized as follows. Section II reviews the concepts of DDDs and s-expanded DDDs. Section III presents new incremental shortest path based algorithm. We also briefly review our implementation of the dynamic programming based algorithm based on DDD graphs. Experimental results are described in Section IV. Section V concludes the paper.

II. DDDs and s-Expanded Coefficient DDDs

In this section, we provide a brief overview of the notion of determinant decision diagrams [8]. We review how a s-expanded DDD can be used to represent the symbolic coefficients of a s polynomial.

Determinant Decision Diagrams [8] are compact and canonical graph-based representation of determinants. The concept is best illustrated using a simple RC filter circuit shown in Fig. 1. Its system equations can be written as

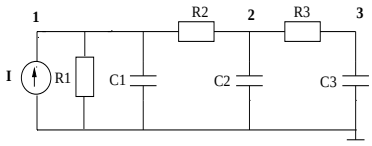

Fig. 1. A simple RC circuit.

$$\begin{bmatrix} \frac{1}{R_1}+sC_1+\frac{1}{R_2} & -\frac{1}{R_2} & 0 \\ -\frac{1}{R_2} & \frac{1}{R_2}+sC_2+\frac{1}{R_3} & -\frac{1}{R_3} \\ 0 & -\frac{1}{R_3} & \frac{1}{R_3}+sC_3 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} I \\ 0 \\ 0 \end{bmatrix}$$

We view each entry in the circuit matrix as one distinct symbol, and rewrite its system determinant in the left-hand side of Fig. 2. Then its DDD representation is shown in the right-hand side. Please refer to [8] for the formal definition of a DDD graph.

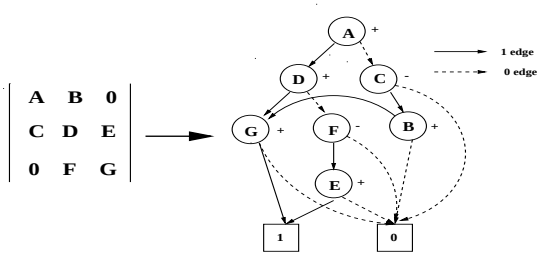

Fig. 2. A matrix determinant and its DDD.

A *1-path* in a DDD corresponds a product term in the original DDD, which is defined as a path from the root vertex (A in our example) to the 1-terminal including all symbolic symbols and signs of the vertices that originate all the 1-edges along the 1-path. In our example, there exist three 1-paths representing three product terms: ADG, $-AFE$ and $-CBG$. The root vertex represents the sum of these product terms. Size of a DDD is the number of DDD vertices, denoted by $|DDD|$. Note that the size of a DDD depends on the size of circuits in a very complicated way. $|DDD|$ is a linear function of the circuit size for ladder circuits and it may grow superlinearly in general with sizes of circuits [8].

To exploit the DDD to derive circuit characteristics, we need to directly represent circuit parameters not matrix entries. To this end, s-expanded DDDs are introduced [10]. Consider again the circuit in Fig. 1 and its system determinant. Let us introduce a unique symbol for each circuit parameter in its admittance form. Specifically, we introduce $a=\frac{1}{R_1}$, $b=f=\frac{1}{R_2}$, $d=e=-\frac{1}{R_2}$ $g=k=\frac{1}{R_3}$, $i=j=-\frac{1}{R_3}$, $C_1=c, h=C_2, l=C_3$. Then the circuit matrix can be rewritten as

$$\begin{bmatrix} a+b+cs & d & 0 \\ e & f+g+hs & i \\ 0 & j & k+ls \end{bmatrix}$$

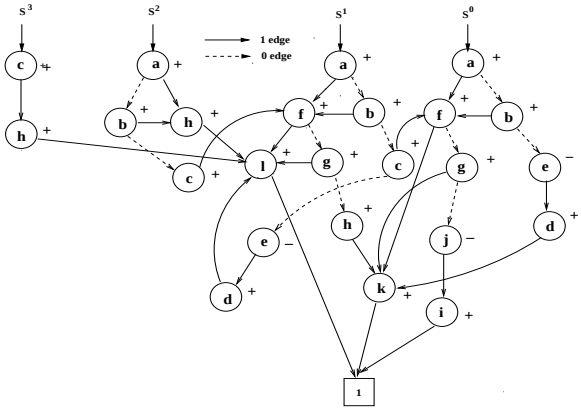

Fig. 3. An s-expanded DDD.

The original 3 product terms will be expanded to 23 product terms in different powers of s. We can represent these product terms nicely using a slight extension of the original DDD, as shown in Fig. 3. This DDD has exactly the same properties as the original DDD except that there are four roots representing coefficients of s^0, s^1, s^2, s^3. Each DDD root represents a symbolic expression of a coefficient in the corresponding s polynomial. Each such DDD is called a *coefficient* DDD, and the resulting DDD is a *multiple-root* DDD. The original DDD in which s is contained in some vertices is called *complex* DDD.

The s-expanded DDD can be constructed from the complex DDD in linear time in the size of the original complex DDD [9, 10].

III. Dominant Term Generation Methods

Many circuit characteristics are dominated by a small number of product terms called *significant* or *dominant* terms. Those dominant terms can be used to well approximate the circuit behavior with very small accuracy loss. In the sequel, we first review the implementation of the dynamic programming based term generation algorithm based on DDD graphs and then discuss the graphic properties the algorithm depends on and its time complexity. Then we will details our new incremental shortest path (SP) based term generation algorithm and compare it with the dynamic programming (DP) based algorithm.

A. Dynamic Programming (DP) Based Generation of Dominant Terms

Verhaegen and Gielen [14] recently proposed a fast term generation algorithm based on dynamic programming concept. Their implementation, however, is based on the DDD structure that is slightly different from the one developed originally by Tan and Shi [8, 10]. Inspired by their work, we implemented a DP based algorithm based on the original DDD structure.

In a DDD graph, a 1-edge pointed vertex (vertex which have incoming 1-edge) represents a minor in a determinant with the root vertex representing the whole determinant. So recursively,

we can cache all the generated dominant terms from the minor at the vertex to avoid regenerating them again. To avoid generating a term more than twice, each vertex also has a counter to keep track of the terms generated through the vertex.

Specifically, let D be a 1-edge pointed vertex. We use *D.counter* to keep track of the number of dominant terms generated for the vertex D (i.e. D is included in those terms). We use an ordered array, denoted as *D.term-list*, to keep track of those generated dominant terms in the minor represented by D, where *D.term-list[1]*, *D.term-list[2]*,... represents the largest term (first dominant term), the second largest term (second dominant term). *D.counter* is initially set to 1 for all 1-edge pointed vertices, and can be increased up to k. We use $V.child1(V.child0)$ to represent the vertex pointed to by the 1-edge (0-edge) originating from vertex V. The pseudo code of the algorithm is shown in Fig. 4.

```
GETKDOMINANTTERMS(D,k)
1   if ( D = 1)
2       return 1
3   if ( D = 0 )
4       return NULL
5   else if ( D.term-list[k] exists )
6       return D.term-list[k]
7   else
8       COMPUTEKDOMINANTTERMS(D,k)
9   return D.term-list[k]

COMPUTEKDOMIANTTERMS(D,k)
1   if ( D = 1)
2       return 1
3   while (D.term-list[k] not exists) do
4       for each 0-edge linked vertex V starting with D do
5           if ( V = 1 and V.counter > 1)
6               continue
7           term = GETKDOMINANTTERMS(V.child1,V.counter)
8           if ( term exists)
9               new_term = UPDATETERM(term,V)
10              term_value = COMPUTETERMVALUE(new_term)
11              if (term_value is the largest)
12                  new_largest_term = new_term
13                  vertex_need_update = V
14      if (new_largest_term exists)
15          D.term-list.push(new_largest_term)
16          vertex_need_update.counter++
17      else
18          break
19  return
```

Fig. 4. Dynamic programming based dominant term generation algorithm.

To find the k dominant terms at a 1-edge pointed vertex D, we first check if such k terms already exist in the *term-list* in GETKDOMINANTTERMS(D,k). If they do not exist, they will be generated by invoking COMPUTEKDOMINANTTERM(D,k). In COMPUTEKDOMINANTTERM(D,k), the largest term is computed and stored in the *term-list* of D by visiting all the 0-edge linked DDD vertices. Each time a largest term is computed, the corresponding vertex *V.counter* will be increased by 1. UPDATETERM() adds a vertex (its symbol) into a term represented by a DDD tree. COMPUTETERMVALUE() computes the numerical value of a given term.

We have to point out that the DP based algorithm does not work for all the DDD graphs. It was shown in [13] that some graphic properties of DDDs have to be held. The most important one is that the incoming edges of a non-terminal vertex in a DDD graph are either all 0-edges or all 1-edges. In other words, a 0-edge pointed vertex can only belong to one minor and can not be shared by different minors (no incoming 1-edge). With this property, it is sufficient to keep all the generated dominant terms in the 1-edge pointed vertices as required by the DP based algorithm.

We notice that cancellation-free s-expanded DDDs (whose canceling terms are removed) do not satisfy required property [13]. Verhaegen and Gielen [14] resolved this problem by duplicating vertices. Their approach, however, will destroy the DDD canonicity, a property crucial to the efficiency of many DDD-based graph manipulations as DDD (also BDD) operations are based many caching techniques which require the canonical property of DDDs to make caching operations effective [7]. In this paper, we apply the DP approach on the s-expanded DDDs before de-cancellation.

The algorithm takes linear time in terms of the size of a DDD, if UPDATETERM() and COMPUTETERMVALUE() are implemented to use constant time each time when a vertex is added to a term. This can be accomplished by using memory caching. We also note that after the first dominant term, the algorithm takes $O(n)$ to get the next dominant term if the numbers of 0-edge linked vertices is bounded by a constant m and $m << n$, where n is depth of the DDD graph or the size of the circuit matrix. This typically is the case for a sparse matrix. For a dense matrix, the numbers of 0-edge linked vertices become proportional to $O(n)$, so the time complexity of finding next a dominant term will become $O(n^2)$. In other words, the time complexity of the DP based algorithm depends on the topologies of circuits.

B. Incremental Shortest Path (SP) based Generation of Dominant Terms

Tan and Shi proposed an elegant algorithm for finding k dominant terms in [12]. The algorithm does not require DDDs to satisfy aforementioned graphic property, and thus can be applicable to any DDD graph. The algorithm is based on the observation that the k dominant product terms can be transformed to the k shortest paths in a *DDD*. In this manner, we can find the k shortest paths in time

$$O(k \cdot |DDD| + n\frac{k(k-1))}{2}), \tag{1}$$

where n is the depth of the DDD graph. In this paper, we introduce a more efficient term generation algorithm based on the work in [12]. The new algorithm is also based on searching the shortest paths in the DDD graphs to find the dominant terms. But unlike the previous method, the new algorithm does not need to visit every vertex in a DDD graph to find the dominant term as required by the shortest path search algorithm [1] after the first dominant term. The new algorithm is based on the observation that not all the vertices are needed to be relaxed (a operation that checks if a path from a vertex's parent

is the shortest path seen so far and remember the parent if it is) after the DDD tree is altered due to the subtraction of a dominant term. We show that only the newly added DDD vertices are needed to be relaxed and the number of newly added DDD vertices is bounded by the depth of the DDD graph. In the sequel, we first introduce the concept of reverse DDD graphs.

As shown in Fig. 2, a DDD graph is a direct graph with two terminal vertices and one root vertex. Remember that the 1-path in a DDD graph is defined from the root vertex to the 1-terminal. In this paper, we define a new type DDD graph, called *reverse* DDD graphs where all the edges have their directions reversed and the root of the new graph are 1-terminal and 0-terminal vertices and the root vertex of the original DDD graph becomes the only terminal vertex in the reverse DDD graph. The reverse DDD graph for the DDD graph in Fig. 2 is shown in Fig. 5. But in the sequel, the root vertex and terminal vertices are still referred to those in the original DDD graphs.

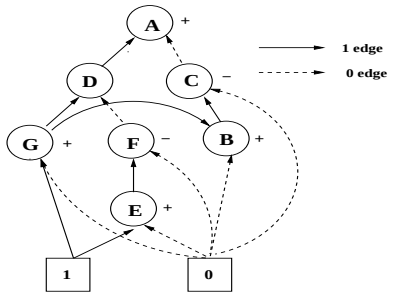

Fig. 5. A reverse DDD graph.

With the concept of the reverse DDD graph, we further define 1-path and path weight in a reverse DDD graph.

Definition 1 *A 1-path in a reverse DDD is defined as a path from the 1-terminal to root vertex (A in our example) including all symbolic symbols and signs of the vertices that the 1-edges point to along the 1-path.*

Definition 2 *The cost of a path in a DDD is defined to be the total cost of the edges along the path where each 0-edge costs 0 and each 1-edge costs $-log|a_i|$, and $|a_i|$ denotes the numerical value of the DDD vertex a_i that the corresponding 1-edge points to.*

We can show the following result:

Lemma 1 *The most significant product (dominant) term in a symbolic determinant D corresponds to the minimum cost (shortest) path in the corresponding reverse DDD between the 1-terminal and the root vertex.*

The shortest path in a *s*-expanded DDD, which is a DAG direct acyclic graph), can be found by depth-first search in time $O(V+E)$, where V is the number of DDD vertices $|DDD|$ and E is number of edges [1]. For DDDs, $E = 2|DDD|$, thus the SP based algorithm takes time $O(|DDD|)$.

Following the same strategy in [12], after we find the shortest path from a DDD, we can subtract it from the DDD using *subtract* DDD operation [8], and then we can find the next shortest path in the resulting DDD. *Subtraction* is a DDD graph operation that generates a new DDD graph that does not contain the path subtracted. Then we have following result:

Lemma 2 *In a reverse DDD graph, after all the vertices have been relaxed (after finding the first shortest path), the next shortest path can be found by relaxing newly added vertices created by the subtraction operation.*

Proof: The proof of Lemma 2 lies in the canonical nature of DDD graphs. A new DDD vertex is generated if and only if the subgraph rooted at the new vertex is a new and unique subgraph for the existing DDD graph. In other words, there do not exist two identical subgraphs in a DDD graph due to the canonical nature of DDD graphs. On the other hand, if a existing DDD vertex becomes part of the new DDD graph, its corresponding subgraph will remain the same. As a result, the shortest path from 1-terminal to all vertices in the subgraph will remain the same. Hence, it is sufficient to relax the newly added vertices to find the shortest paths from 1-terminal to those vertices. The root vertex of the new DDD graph is one of those newly added vertices. □.

It turns out that relaxation for the new vertices can be done very efficiently when those new vertices get created. The relaxation can take a almost free ride during the subtraction operation. Suppose all the vertices in reverse *D* have been relaxed. Then the pseudo code of the new algorithm for searching the next dominant term is given in Fig. 6.

```
GETNEXTSHORTESTPATH(D)
1   if (D = 0)
2       return 0
3   P = EXTRACTPATH(D)
4   if (P exists and P not equal to 1)
5       D = SUBTRACTANDRELAX(D, P)
6   return P

SUBTRACTANDRELAX(D, P)
1   if (D = 0)
2       return 0
3   if (P = 0)
4       return D
5   if (D = P)
6       return 0
7   if (D.top > P.top)
8       V = GETVERTEX(D.top, D.child1, SUBTRACTANDRELAX(D.child0, P))
9   if (D.top < P.top)
10      V = SUBTRACTANDRELAX(D, P.child0)
11  if (D.top = P.top)
12      T1 = SUBTRACTANDRELAX(D.child1, P.child1))
13      T0 = SUBTRACTANDRELAX(D.child0, P.child0)
14      V = GETVERTEX(D.top, T1, T0)
15  if (V not equal to D)
16      RELAX(V.child1, V)
17      RELAX(V.child0, V)
18  return V
```

Fig. 6. Incremental shortest path based dominant term generation algorithm.

In GETNEXTSHORTESTPATH(D), EXTRACTPATH(D) obtains the found shortest path from D and returns the path in a single DDD graph form. This is done by simply traversing from the root vertex to 1-terminal as each vertex will remember its immediate parent who is on the shortest path to the vertex in a fully relaxed graph. Once the shortest path is found, we *subtract* it from the existing DDD graph and relax the newly created DDD graphs (line 15-17) at same time to find the shortest paths to those vertices.

In function SUBTRACTANDRELAX(D,P), GETVERTEX($top, D.child1, D.child0$) is to generate (or copy) a vertex for a symbol top and two subgraphs $D.child1$ (pointed by 1-edge) and $D.child0$ (pointed by 0-edge). RELAX(P, Q) performs the relaxation operation for vertices P and Q where P is the immediate parent of Q in the reverse DDD graph. In the reverse DDD graph, each vertex has only two incoming edges (from its two children in the normal DDD graph), so the relaxation with its two parents in line 16 and 17 is sufficient for vertex V. Moreover, the relaxation for V happens after all its parents have been relaxed due to the DFS-type traversal in SUBTRACTANDRELAX(). This is consistent with the ordering requirement of the shortest path search algorithm. So by repeatedly calling function GETNEXTSHORTESTPATH(D), we can find all the dominant terms in a decreasing order.

Let n be the number of vertices in a path from 1-terminal to the root vertex, given the fact that D is a DDD graph and P is a path in DDD form, then we have the following Lemma

Lemma 3 *The number of new DDD vertices created in function* SUBTRACTANDRELAX*(D, P) is bounded by n and the time complexity of the function is* $O(n)$.

Proof: As we know that DDD graph D contains the path P. As P is a single path DDD graph, $P.child0$ is always 0-terminal. So line 10 and 13 will immediately return D and $D.child0$ respectively (actually, line 10 will never be reached if D contains the path P). As a result, we will descend one level down in graph D each time depending on which line we choose to go for line 8 and 12. As a matter fact, function SUBTRACTANDRELAX(D, P) actually will traverse the path P in D until it hits a common subgraph in both D and P as indicated in line 5. After this, $m-1$ new vertices will be created on its way back to the new root vertex, m is the number of vertices visited in D in the whole operation and $m < n$. If the common subgraph is 1-terminal, then $m = n$. □

Notice that both DP based algorithm and incremental SP based algorithm have time complexity $O(|DDD|)$ to find a dominant term in general, where $|DDD|$ is the size of a DDD graph. After the first dominant term, however, both algorithms show better time complexities for generating next dominant terms. But in contrast to DP based algorithm, the $O(n)$ time complexity of the incremental SP based algorithm does not depends on the topologies of circuits.

Notice that this new term generation algorithm can be performed on the any DDD graph, including cancellation-free s-expanded *DDD*. We also note that since the variant of DDD used by Verhaegen and Gielen in [14] does not satisfy the canonicity, and thus cannot apply the SP based algorithm.

Following the same strategy in [15], our approach also handles numerical cancellation. Since numerical canceling terms are extracted one after another, they can be eliminated by examining two consecutive terms.

IV. EXPERIMENTAL RESULTS

The proposed approach has been implemented and tested on a number of practical analog circuits. For each circuit, DC analysis is first carried out using SPICE and our program reads in small-signal element values from the SPICE output. The algorithms described in [8, 10] are used to construct complex DDDs and s-expanded DDDs.

Table I summarizes the comparison results in terms of CPU time for the three algorithms. A total of 10000 dominant terms are generated for a number of test circuits ranging from more regularly structured ladder circuits to less regularly structured such as *Cascode* and $\mu A741$ amplifiers. In Table I, column 1, 2 and 3 list for each circuit, respectively, its name *Circuit*, the number of nodes *#nodes*, and the number of nonzero elements *#nonzero* in its circuit MNA matrix. Columns 4 to 6 show, respectively, the CPU time, in terms of seconds, for generating 10000 dominant terms by the DP based algorithm, *Dyna Programming*, by the pure SP based algorithm, *Shortest Path*, and by incremental SP based algorithm, *Incr Shortest Path*.

From Table I, we see that the incremental SP based algorithm consistently outperforms the DP based algorithm for all the circuits in CPU time. The difference becomes even more significant for circuits with regular structures like ladder circuits. For less regularly structured circuits like $\mu A741$, the incremental SP based method also shows impressive improvements over the DP based algorithm.

As we know that for both DP based algorithm and incremental SP based algorithm, the actual time to generate a new path is close to $O(n)$, where n is the size of the circuit. But for pure SP based algorithm, we have to visit all the vertices every time to generate a new path. Such difference is clearly demonstrated in circuits *Cascode* and $\mu A741$ where the sizes of DDDs are significantly larger than the sizes of the circuits. So the CPU time for pure SP based algorithm is quite lager than two other algorithms.

Fig. 7 shows the CPU time for different ladder circuits. The CPU time increases almost linearly with the size of ladder circuits for all three algorithms. So for ladder circuits, both the SP algorithms consistently outperforms DP based algorithm in terms of CPU time. The reason is that sizes of DDDs for representing ladder circuits grow linearly with the sizes of the ladder circuits, that is n [8], so the time complexities of all three algorithms, $O(|DDD|)$, become $O(n)$. But the DP based algorithm need to take extra efforts to loop through all 0-linked vertices to compute the dominant terms and restore them at each 1-edged pointed vertex. Those extra efforts will become significantly when the graph become very deep as with the high section ladder circuits.

TABLE I
COMPARISON OF THREE TERM GENERATION ALGORITHMS.

Circuit	#nodes	#nonzero	Dyna Programming	Shortest Path	Incr Shortest Path
rclad10	8	31	17.3	14.8	5.1
rclad21	22	64	105.1	21.5	15.4
rclad60	61	181	133.8	101.0	80.3
rclad100	101	301	369.6	172.8	132.9
rclad150	151	451	912.2	281.7	248.3
rclad200	201	601	1630.9	387.3	320.0
rclad250	251	751	2431.3	493.8	426.4
rclad300	301	901	3388.4	598.0	557.4
rctreeA	40	119	41.4	45.6	41.0
rctreeB	53	158	132.9	60.4	57.2
Cascode	14	76	21.3	620.1	15.3
$\mu A741$	23	90	50.6	1412.2	21.0
bigtst	32	112	91.7	144.1	32.7

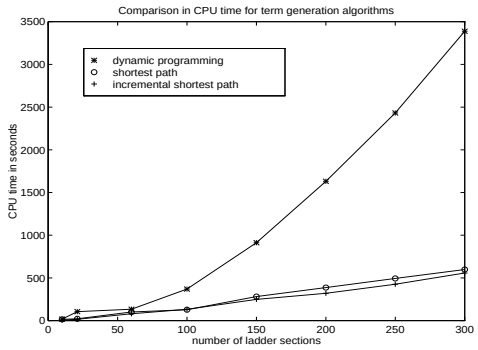

Fig. 7. CPU time vs number of ladder sections.

V. CONCLUSIONS

An efficient approach is proposed to generate dominant terms for behavioral modeling of large linear analog circuits. The new algorithm is based on the fact that only a small portion of DDD vertices need to be updated to find a new shortest path when the first one is found. Such incremental updating scheme significantly improve efficiency of generating dominant terms from DDD graphs. Experimental results have demonstrated that the new symbolic term generation algorithm consistently exceeds both pure shortest path based algorithm and dynamic programming based algorithm, the fastest DDD-based term generation algorithm reported so far.

REFERENCES

[1] T. Cormen, C. E. Leiserson and R. L. Rivest, *Introduction to Algorithms*, The MIT Press, Cambridge, Massachusetts 1990.

[2] F. V. Fernández, P. Wambacq, G. Gielen, A. Rodríguez-Vázquez, and W. Sansen, "Symbolic analysis of large analog integrated circuits by approximation during expression generation," in *Proc. IEEE Int. Symp. Circuits and Systems*, pp. 25–28, 1994.

[3] G. Gielen and W. Sansen, *Symbolic Analysis for Automated Design of Analog Integrated Circuits*, Kluwer Academic Publishers, 1991.

[4] G. Gielen, P. Wambacq and W. Sansen, "Symbolic analysis methods and applications for analog circuits: A tutorial overview", *Proc. IEEE*, vol. 82, no. 2, pp. 287–304, Feb. 1994.

[5] J.-J. Hsu and C. Sechen, "DC small signal symbolic analysis of large analog integrated circuits", *IEEE Trans. Circuits and Systems-I: Fundamental*, vol. 41, no. 12, pp. 817–828, Dec. 1994.

[6] P. M. Lin, *Symbolic Network Analysis*, Elsevier Science Publishers B.V., 1991.

[7] S. Minato, *Binary Decision Diagrams and Application for VLSI CAD* Kluwer Academic Publishers, Boston, 1996.

[8] C.-J. Shi and X.-D. Tan, "Canonical symbolic analysis of large analog circuits with determinant decision diagrams", *IEEE Trans. Computer-Aided Design*, vol. 19, no. 1, pp. 1–18, Jan. 2000.

[9] C.-J. Shi and X.-D. Tan, "Effi cient derivation of exact s-expanded symbolic expressions for behavioral modeling of analog circuits", in *Proc. IEEE Custom Integrated Circuits Conf. (CICC)*, pp. 463–466, 1998.

[10] C.-J. Shi and X.-D. Tan, "Compact representation and effi cient generation of s-expanded symbolic network functions for computer-aided analog circuit design", *IEEE Trans. Computer-Aided Design*, vol. 20, No. 7, pp. 813-827, July 2001.

[11] S. J. Seda, M. G. R. Degrauwe and W. Fichtner, "A symbolic analysis tool for analog circuit design automation," in *Proc. IEEE Int. Conf. Computer-Aided Design (ICCAD)*, pp. 488–491, 1988.

[12] X.-D. Tan and C.-J. Shi, "Interpretable symbolic small-signal characterization of large analog circuits using determinant decision diagrams", in *Proc. Design, Automation and Test in Europe (DATE'99)*, pp. 448–453, Munich, Germany, Mar. 10-13, 1999.

[13] X.-D. Tan and C.-J. Shi, "Parametric analog behavioral modeling based on cancellation-free DDDs", in *Proc. IEEE International Workshop on Behavioral Modeling and Simulation (BMAS'02)*, Santa Rosa, California, Oct. 2002.

[14] W. Verhaegen and G. Gielen, "Effi cient DDD-based symbolic analysis of large linear analog circuits", in *Proc. ACM/IEEE 38th Design Automation Conference (DAC)*, pp. 139–144, Las Vegas, June 2001.

[15] P. Wambacq, G. Gielen and W. Sansen, "A cancellation-free algorithm for the symbolic simulation of large analog circuits", in *Proc. IEEE Int. Symp. Circuits and Systems*, pp. 1157–1160, May 1992.

[16] P. Wambacq, G. Gielen and W. Sansen, "A new reliable approximation method for expanded symbolic network functions", in *Proc. IEEE Int. Symp. Circuits and Systems*, pp. 584–587, 1996.

[17] Q. Yu and C. Sechen, "A unifi ed approach to the approximate symbolic analysis of large analog integrated circuits", *IEEE Trans. Circuits and Systems*, vol. 43, no. 8, pp. 656–669, Aug. 1996.