Optimized Blist Form (OBF)

Jarek Rossignac

GVU Center & School of Interactive Computing
College of Computing, Georgia Institute of Technology, Atlanta, Georgia, USA
http://www.gvu.gatech.edu/~jarek/

Abstract

*Any Boolean expressions may be converted into positive-form, which has only union and intersection operators. Let **E** be a positive-form expression with n literals. Assume that the truth-values of the literals are read one at a time. The numbers s(n) of steps (operations) and b(n) of working memory bits (footprint) needed to evaluate **E** depend on **E** and on the evaluation technique. A recursive evaluation performs s(n)=n–1 steps but requires b(n)=log(n)+1 bits. Evaluating the disjunctive form of **E** uses only b(n)=2 bits, but may lead to an exponential growth of s(n). We propose a new Optimized Blist Form (OBF) that requires only s(n)=n steps and* $b(n)=\lceil \log_2 j \rceil$ *bits, where* $j=\lceil \log_2(2n/3+2) \rceil$*. We provide a simple and linear cost algorithm for converting positive-form expressions to their OBF. We discuss three applications: (1) Direct CSG rendering, where a candidate surfel stored at a pixel is classified against an arbitrarily complex Boolean expression using a footprint of only 6 stencil bits; (2) the new Logic Matrix (LM), which evaluates any positive form logical expression of n literals in a single cycle and uses a matrix of at most n×j wire/line connections; and (3) the new Logic Pipe (LP), which uses n gates that are connected by a pipe of* $\lceil \log_2 j \rceil$ *lines and when receiving a staggered stream of input vectors produces a value of a logical expression at each cycle.*

1. Introduction

1.1 Background, terminology, and notation

For simplicity, we consider only ***positive form*** Boolean expressions. We can do so without loss of generality because arbitrary Boolean expressions may be converted to their positive-form as follows. Express all operators in terms of ***union***, which we denote by "+", ***intersection***, which we omit or denote by "•", and ***complement***, which we denote by a preceding "!" and endow with highest priority. For example, the difference a\b is converted to a•(!b), simply denoted a!b, and the symmetric difference (logical XOR) a⊗b is converted to a!b+b!a. Then, convert the expression into its positive form by recursively applying de Morgan laws: !!a=a, !(a+b)=!a!b, and !(ab)=!a+!b. Finally, replace all complemented ***literals*** with new literals that denote their complement. The result is an expression with *n* literals and *n*–1 operators that are either • or +. For example (a•b)+(c•(d+e)) has 5 literals and 4 operators. For simplicity, omitting • and assuming that it has higher priority than +, it will be written ab+c(d+e). Remember that each occurrence of a variable is a different literal. For example, a!b+b!a has four literals. For clarity, we use a different symbol (a, b, c…A, B, C…) to denote each literal. From now on, and throughout this paper, let ***E*** be an expression in positive-form and let *n* be the number of literals in ***E***.

Parsing ***E*** produces a ***binary tree T*** whose 2*n–1* ***nodes*** correspond each to a different literal or operator in ***E***. The node corresponding to the operator executed last when evaluating ***E*** is the ***root***. Nodes corresponding to literals are called ***leaves***. A non-leaf node is called an ***op-node***. There are exactly *n–1* op-nodes. If an op-node N corresponds to a Boolean combination (L+R or L•R) of two sub-expressions or literals, we say that L and R are respectively the ***left*** and ***right children*** of N and that N is their ***parent***. We say that each parent is separated from its children by one ***link***. The ***distance*** between two nodes is the number of links in the shortest path joining them. The ***depth*** of a node is the maximum distance separating it from its leaves. Hence the depth of a leaf is zero and the depth of the root in a+bc is 2. We say that the tree is ***full*** (Fig. 1, left) when the left and right children of each op-node have the same depth. We say that a tree is ***alternating*** (Fig. 1, right) when no op-node has a parent with the same operator.

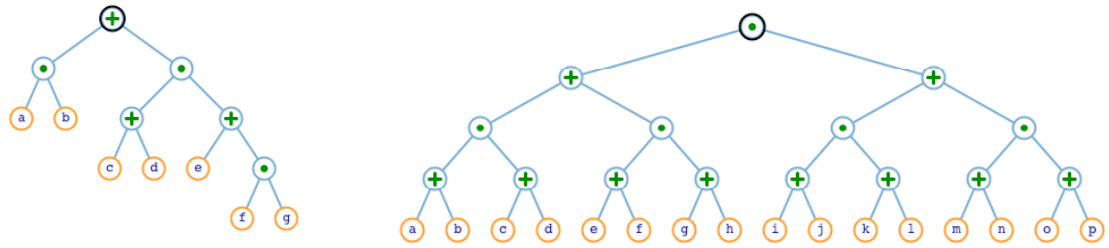

Fig. 1: The alternating expression ab+(c+d)(e+fg) is shown left. The full alternating expression ((a+b)(c+d)+(e+f)(g+h))((i+j)(k+l)+(m+n)(o+p)) with 16 literals is shown right. Its disjunctive form has 256 products.

Note that both either • and + operators are commutative. Hence, we can transform ***T*** to another tree representing an equivalent expression by a sequence of ***pivots***, which each swap the left and right children of an op-node. For instance, a ***left-heavy form*** of ***T*** (Fig. 2 right), in which the depth of the left child of each op-node equals or exceeds the depth of the right child, may be constructed by a recursive procedure that pivots each op-node N if the depth of the right child exceeds the depth of the left child, and that returns the depth of N. For example, bc+a is the left-heavy version of a+bc.

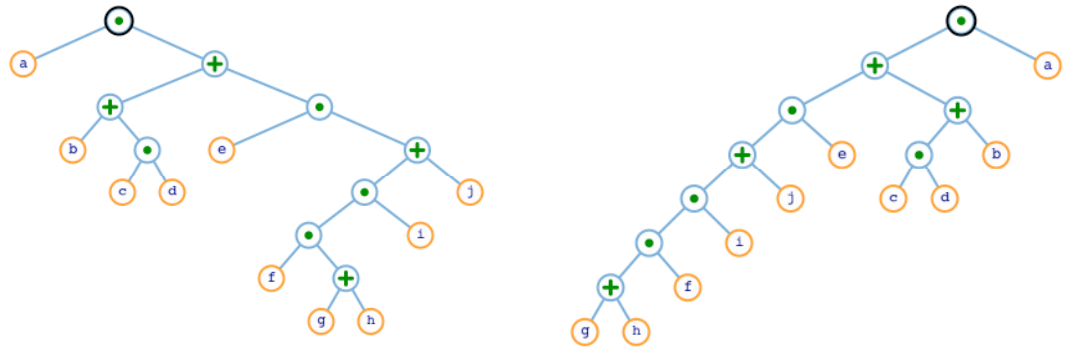

Fig. 2: The positive-form expression a(b+cd+e(f(g+h)i+j)) and its tree (left) may be pivoted into a left-heavy form (right) (((g+h)fi+j)e+(cd+b))a.

1.2 Evaluation cost

Assume that we can read the truth-value of each literal one at a time. We would like to minimize the number $s(n)$ of steps (operations) and the number $b(n)$ of working memory bits (which we call the *footprint*) that are needed to evaluate ***E***. Both $s(n)$ and $b(n)$ depend of course on n, but also on ***E*** and on the particular evaluation technique used.

The ***naïve evaluation*** from left-to-right that respects parentheses and operator priorities will perform $s(n)=n-1$ steps, one per operator, but may require a footprint of n bits. It corresponds to a bottom-up traversal of the binary tree. For example, in a+b(c+d(e+f(g…))) if a=*false*, b=*true*, c=*false*, d=*true* … the left-to-right evaluation will cache the truth values of all the literals before starting to combine them.

Because + and • are commutative, one can often reduce the footprint by ***pivoting*** such expressions (swapping left and right arguments of selected operators, or equivalently the left and right children of tree nodes) to make the tree left-heavy. On can easily show that for some expressions, even with pivoting, $b(n)\geq\lceil\log_2 n\rceil+1$.

To further reduce the footprint, one may consider evaluating the ***disjunctive form*** of ***E*** [GMTF89], which may be pre-computed by distributing all • operators over +. For example, the disjunctive form of a(b+c)(d+e) is the sum (union) of four products (intersections): abd+abe+acd+ace. Note that one does not need to store the entire disjunctive form explicitly, as its products may be easily processed, one at a time, directly from ***T*** [Ross94]. Evaluating the disjunctive form of ***E*** reduce $b(n)$ to 2 bits: one bit records whether any of the already processed literals in the current product was *false* and the other bit records whether any of the previously processed products evaluated to *true*. Unfortunately, such an approach may require an exponential number of steps, since the disjunctive form may have $2^{n/2}$ products of $n/2$ literals each. For example, (a+b)(c+d)(e+f)(g+h)(i+j)(k+l)(m+n)(o+p), shown in Fig. 3, yields 2^8 products.

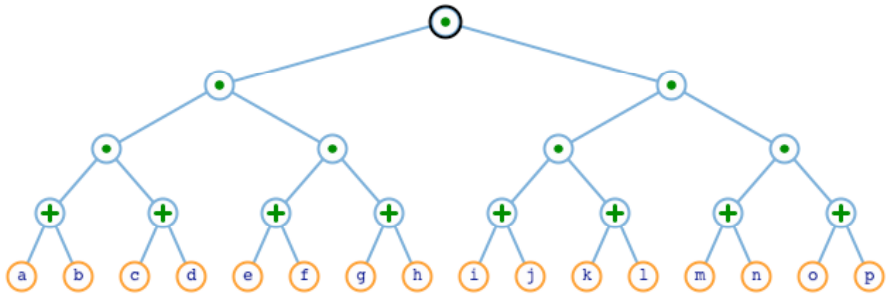

Fig. 3: The expression (a+b)(c+d)((e+f)(g+h))((i+j)(k+l)((m+n)(o+p))) with 16 literals yields a disjunctive form of 256 products of 8 literals each, of the form acegikmo+acegikmp+acegikno+acegiknp+acegilmo…

1.3 Outline of the paper

As an alternative to these two extremes (the naïve evaluation possibly requiring a $\log_2 n$ footprint and the disjunctive form possibly requiring an exponential number of steps), we propose an approach, which requires only $s(n)=n$ steps and $b(n)=\lceil \log_2 j \rceil$ bits, where $j=\lceil \log_2(2n/3+2) \rceil$. It is based on the ***Optimized Blist Form*** (abbreviated OBF) introduced here. OBF is an improvement on the non-optimized ***Blist*** form [Ross99]. For the reader's convenience, we first reintroduce the computation and evaluation of the Blist form, then propose a simple, linear cost algorithm for converting positive form expressions to their OBF and a proof of the upper bound of their footprint.

Then, we discuss the following three applications of OBF.

First, we review two recent approaches, Blister [HaRo05] and Constructive Solid Trimming (CST) [HaRo07], to the ***direct rendering of CSG*** models on the GPU, which use the Blist form to classify candidate surfels against the Boolean expression of the CSG model [Requ80] or of the CSG expression of the ***active zone*** of a primitive [RoVo88]. The complexity of the CSG models that can be handled by these approaches was limited by the fact that the classification of each surfel amounts to the evaluation of a Boolean expression on a small footprint of 6 stencil bits. To alleviate this limitation, these approaches used the Blist of the left-heavy version of the original expression. This prior approach limits the number of CSG primitives (i.e., literals in the Boolean expression) for which a solution is guaranteed to 3909. The OBF solution proposed here removes this limitation, ensuring that any CSG model with up to 2.7×10^{19} primitives can be processed using a footprint of 6 stencil bits per pixel for the evaluation of the corresponding Boolean expression.

Then, we introduce the ***Logic Matrix*** (LM), which evaluates any positive-form ***logical expression*** of n literals in a single cycle and uses n gates, each connected to 3 wires that cross j lines, where $j=\lceil \log_2(2n/3+2) \rceil$. The connections between the wires and the lines in the LM are trivially derived from the OBF of the expression. The LM offers two significant advantages over the RayCasting Engine (RCE) [El&91] developed for the same purpose. The RCE is using a matrix of n by $\log_2 n$ units and hence has same order space complexity, but the RCE units are each a Boolean gate while the LM units are a simple wire/line connection. Furthermore, once the input vector of n truth-values are available on one side of the RCE array, the value of the Boolean function may not be available until $n+\log_2 n$ cycles later, while the LM makes this value available in one cycle.

Finally, we introduce the ***Logic Pipe*** (LP), which uses n gates connected by a pipe of $\lceil \log_2 j \rceil$ lines, with $j=\lceil \log_2(2n/3+2) \rceil$. When the input vectors are staggered, so that each gate reads a different truth-value bit of a different input vector, the LP produces at each cycle a new value of the logical expression corresponding to the next input vector. Both the LP and RCE may be used in this manner for evaluating the same logical expression for a stream of input vectors. The advantage of the LP lies in the fact that it requires only n logical units, each one connected to the next one by $\lceil \log_2 j \rceil$ lines, while the RCE requires roughly n×j units.

2. Boolean List (Blist)

Blist represents each literal as a gate (switch). Consider a gate representing literal A (Fig. 4a). When the gate is up (i.e. A is *true*), if current arrives to the input node at the left of the yellow triangle, current will flow to the upper right exit node (Fig. 4b). When A is *false*, the switch is down and if current arrives at the input node, it flows to the lower output node (Fig. 4c). Hence, the top output represents A and the bottom !A (Fig. 4d). We can then wire two such Blist gates to model a union (Fig. 4e) or an intersection (Fig. 4f) of two literals. For instance, in the union circuit (Fig. 4e), when A

is *true*, arriving current exits from A and reaches directly the top right output node of the combined circuit, regardless of the value of B. If however A is *false*, arriving current flows from its bottom output node to B. If in that case B is *true*, then current flows to its upper output node.

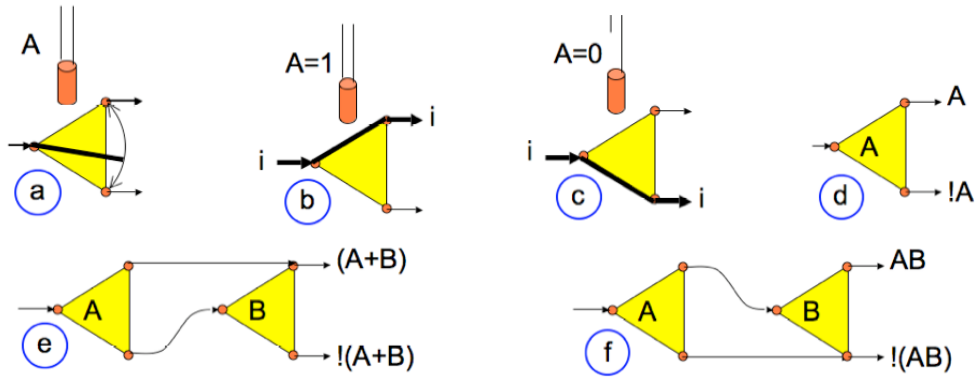

Fig. 4: Current arriving at the input node of gate A (a) flows up if A is *true* (b) or down otherwise (c). The top output represents A (d). Two gates may be wired to represent A+B (e) or AB (f).

Assume now that we have Blist circuits for two Boolean sub-expressions, L and R. We can wire them (Fig. 5) to model L+R or LR.

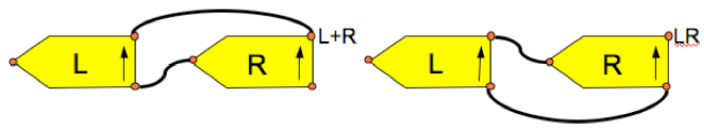

Fig. 5: Blist circuits for Boolean expressions L and R can be combined to model L+R (left) or LR (right).

This process may be applied recursively to construct the Blist of any positive-form Boolean expression. An example is show in Figure 6.

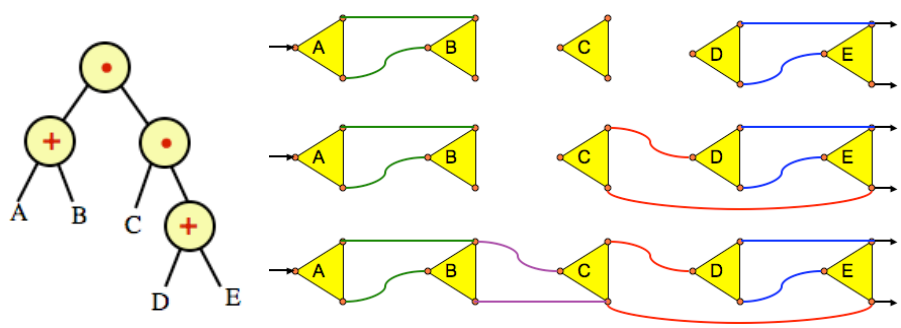

Fig. 6: The expression (A+B)(C(D+E)) may be represented by the tree shown left. We first wire A+B and D+E (top right). Then we wire C(D+E). Finally, we combine two expressions as an intersection (bottom right).

This wiring process associates with each gate G two names: G.T is the name of the gate whose input is reached by the top-output line of G (this is where incoming current would flow if the truth-value associated with G is *true*); G.F is the name of the gate whose input is reached by the bottom line of G (this is where incoming current would flow if the truth-value associated with G is *false*). For instance, in Figure 6, A.T=C and A.F=B. The top of the last gate connects to *true* and the bottom of the last gate to *false*. Therefore, in Figure 6, B.T=C and B.F=*false*. Notice that to reduce the number of lines, which is important for the optimization discussed in the next section, the wiring of the root op-node in Fig. 6 (shown in magenta bottom-right) connects B.F to C.F (as opposed to E.F). There is no need to extend the magenta line to E.F, since C.F is already connected to E.F by a red line.

The Blist evaluation of a Boolean expression uses a footprint called ***next*** that is initialized to the label of the first literal and then always contains the label of the next literal whose truth-value will affect the final value of the expression or the label associated with the final results *true* or *false*. Let P.V be the truth-value of literal P. For each literal P, we perform:

```
if (P==next) {if (P.V) next=P.T; else next=P.F;}
```

2.1 Weaver, the Blist wiring algorithm

We propose (below) an implementation of our ***Weaver*** algorithm, which performs the wiring process described above. We wish to point out the remarkable conciseness of our implementation. We represent each node of ***T*** and each gate (associated with a leaf-node) as a different object. We use a table Nodes[] of node-objects assuming that Nodes[0] is the root. A method of the node class has access to the following fields (internal variables) of a node object:

O is the node-type ('+' or union, '•' for intersection, and ' ' for a literal),
L and R identify the left and right children for op-nodes,
G identifies the gate-object (see below) associated with a leaf node,
T and F identify the gates to reach if the truth-value of the node is *true* and *false* respectively,
t and f stores the costs associated with the node and are used by Flipper (discussed later)

Similarly, we use a table Gates[] of gate-objects. A method of the gate class has access to the following fields (internal variables) of the gate object:

N identifies the literal (name) associated with the gate
T and F identify the gates to reach if the truth value of the node is *true* and *false* respectively
i, t and f are integers identifying the three labels G.i, G.t, and G.f associated with a gate G

We also use two special gates, Tgate and Fgate, which when reached indicate that the expression evaluates to *true* or to *false* respectively. We set Tgate.N='t' and Fgate.N='f'.

To understand how and why Weaver works, note that for a leaf P, P.F is the left-most leaf of the right child of the lowest '+' node whose left child contains P and that P.T is the left-most leaf of the right child of the lowest '•' node whose left child contains P. (The term lowest here means the node with smallest depth.) Hence, Weaver first performs a recursive traversal of the tree of ***E*** initiated by the call Nodes[0].lmg(). During that traversal, each node "denounces" to its parent the gate of its left-most leaf (Fig. 7 center). The code for the lmg method is presented below:

```
gate lmg() {gate g; if(O==' ') g=G; else {if(O=='+') F=R.lmg(); else T=R.lmg(); g=L.lmg(); }; return(g);}
```

Then, Weaver performs a second recursive traversal of the tree of ***E***, initiated by Nodes[0].ptf(Tgate,Fgate), during which each node "passes on" to its children the appropriate t and f values, depending on the operator (Fig. 7 right).

```
void ptf(gate pT, gate pF) { if(O==' ')  {G.T=pT; G.F=pF; T=pT; F=pF;};
    if(O=='+') {L.ptf(pT,F); R.ptf(pT,pF); };    if(O=='•') {L.ptf(T,pF); R.ptf(pT,pF); };  }
```

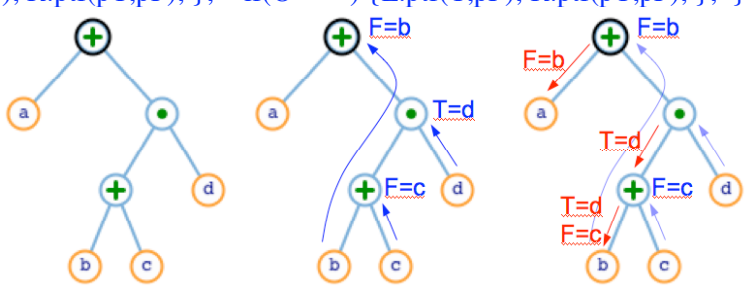

Fig. 7: The expression a+(b+c)d yields the tree shown left. Weaver first propagates the names of the left-most leaves (blue arrows center) up the tree. Then it passes these on to the left children recursively (red arrows right).

2.2 Label assignment

Instead of using literal names for gates, we assign to each gate a ***label*** (positive integer identifier). Hence, Weaver assigns three labels to each gate G: G.i identifies the gate G; G.t is the identifier of the gate referenced by G.T; G.f is the identifier of the gate referenced by G.F.

Notice that Blist evaluation operates left-to-right. Hence, when a gate G is reached, G.i is no longer needed and may be assigned to another gate that follows G and has not yet been assigned a label. To do so, Weaver initializes the identifier G.i of each gate to –1 and executes the loop: for (int i=0; i<nG; i++) Gates[i].assignLabels(); followed by Gates[0].i=0, where assignLabels() is implemented as:

```
void assignLabels() { if(i!=-1) Lab.release(i); if(T.i==-1) T.i=Lab.grab(); t=T.i; if(F.i==-1) F.i=Lab.grab(); f=F.i; }
```

It uses a simple manager of free labels called Lab that keeps track of which labels are in use, and can provide the first (lowest integer) available label (call Lab.grab) or release a label i that is no longer needed (Lab.release(i)). Lab maintains a linked-list of free labels encoded in a table of integers. Its trivial implementation is omitted here.

2.3 Comparing Blists to decision graphs

The Blist form is a special case of a Reduced Function and of an Ordered Binary Decision Diagram (OBDD) [Brya86,Brya95], developed for logic synthesis. Both are Acyclic Binary Decision Graphs [Aker78], which may be constructed through repeated Shannon's Expansion [YaOH97]. For instance: (A+B)!(CD)=(A+B)(C!+D!) yields A!(B(C!+D!)+A(C!+D!), A!(B!+B(C!+D!)+A(C!+D!), and finally A!(B!+B(CD!+C!)+A(CD!+C!), as shown in Fig. 8 left. The space-time complexity of VLSI implementation of Boolean functions has been studied by Bryant [Brya91]. The size (number of nodes) of such a decision graph may be exponential in the number of variables and depends on their order [PaMe77]. Minimizing it by merging isomorphic subgraphs and eliminating parents of isomorphic children (Fig. 8 right) is NP-hard [BoWe96]. Surprisingly, in contrast, a Blist has always exactly *n* nodes and has linear construction and optimization costs (as shown below). This crucial advantage of Blist stems from the fact that Blist treats each literal as a different primitive. In other words, ***E*** is a Boolean expression of *n* literals, not a Boolean function of *n* variables. The footprint for evaluating the Boolean expression using the decision graphs requires O(log*n*) bits. In contrast, the OBF proposed below guarantees a footprint size in O(loglog*n*).

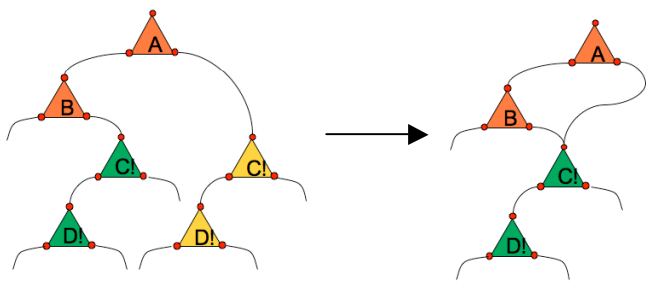

Fig 8: The size of the decision tree for A!(B!+B(CD!+C!)+A(CD!+C!) (left) may be reduced to a DAG (right) by recursively merging identical sub-graphs.

3. Optimized Blist Form (OBF)

The details of the main theoretical contribution of the paper is presented in this section.

3.1 The cost of an expression

We define the ***cost*** of each expression as the maximum number of lines running through the interval between two consecutive Blist gates. Trees and Blists are shown in Fig. 9 for several simple positive-form expressions and in Fig. 10 for a more complex positive-form expression.

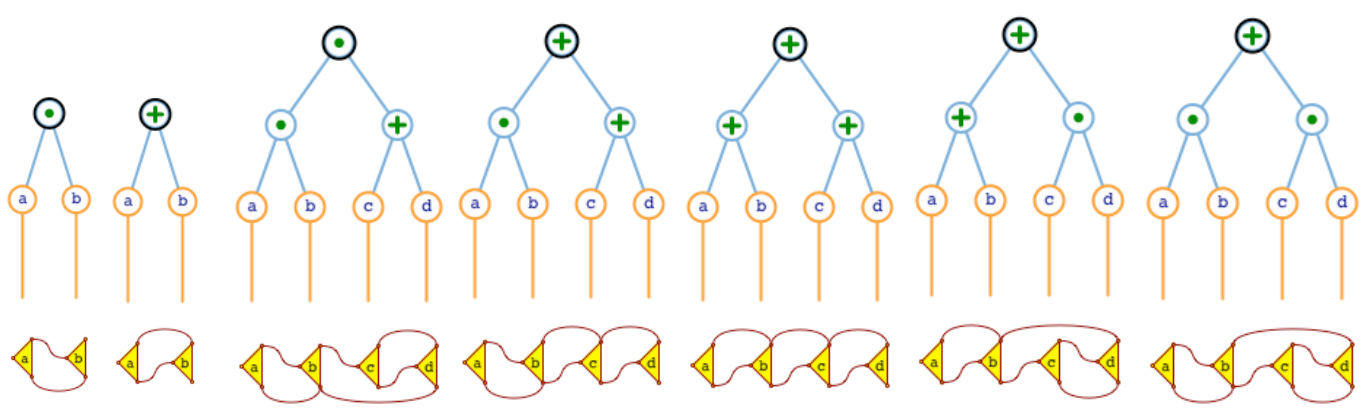

Fig. 9: Tree T (above) and corresponding Blist wiring (below) for the following positive-form expressions, from left to right: ab has cost 2, a+b has cost 2, ab(c+d) has cost 3 since there are 3 lines running through the interval (c,d), ab+(c+d) has cost 2, a+b+(c+d) has cost 2, a+b+cd has cost 3, and ab+cd has cost 3.

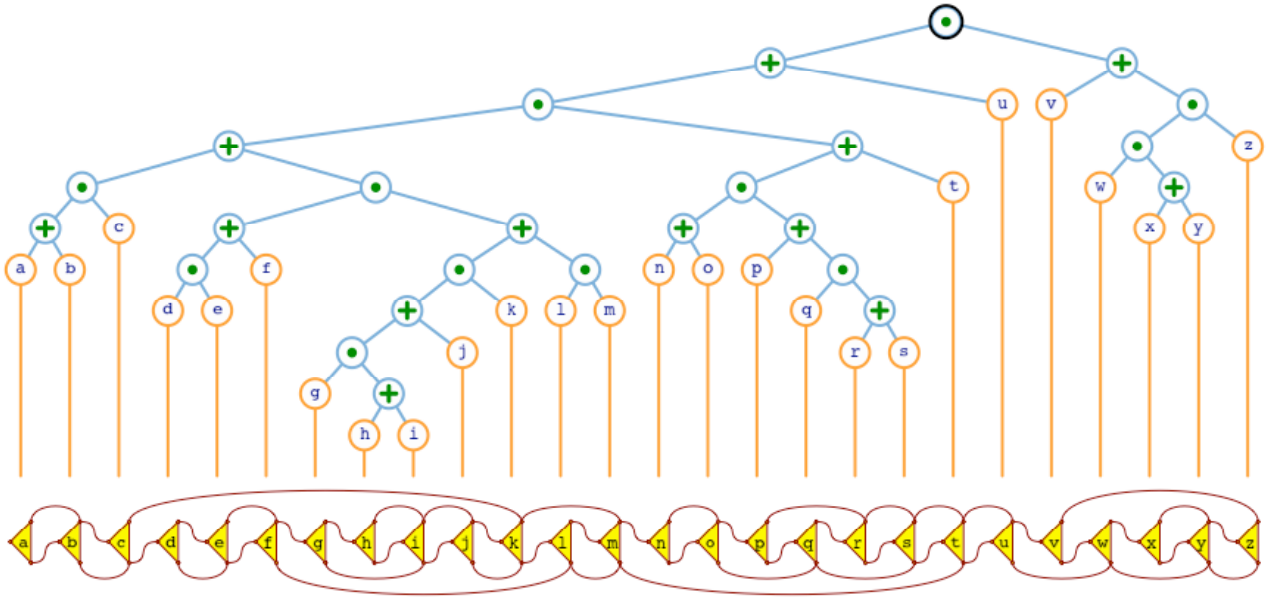

Fig. 10: (((a+b)c+(de+f)((g(h+i)+j)k+lm))((n+o)(p+q(r+s))+t)+u)(v+w(x+y)z) has cost 5 because 5 lines run through the interval between gates h and i.

3.2 Left-heavy strategy

Making the tree T left-heavy, as was proposed in [HaRo05], tends to decrease the cost, as illustrated in Fig. 11.

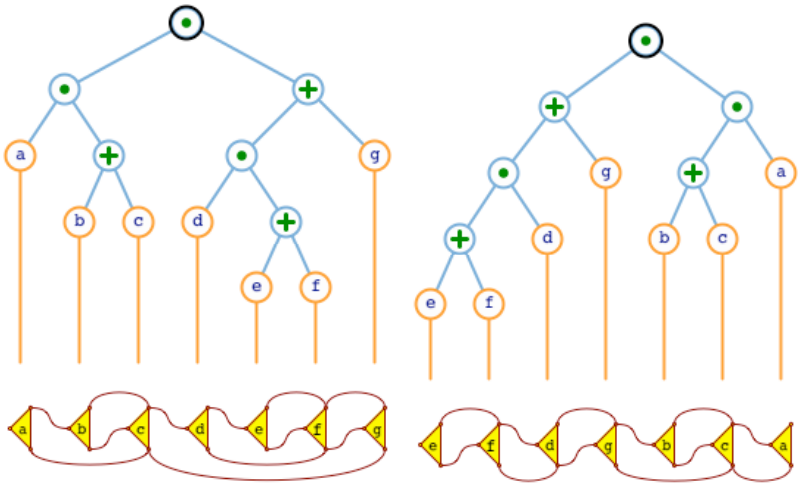

Fig. 11: Pivoting a(b+c)(d(e+f)+g) which has a cost of 4 (left) to make it left-heavy produces ((e+f)d+g)((b+c)a) which has a cost of 3 (right).

Left-heavy pivoting may be trivially implemented as follows:

```
int pH() { if(O==' ') return(0);          // return if reached a leaf-node
  int cL=L.pH(); int cR=R.pH();           // otherwise use recursion to compute the depths cL and cR
  if(cR>cL) {node N; N=L; L=R; R=N; };    // if the right child is deeper, swap them
  return (1+max(cL,cR)); }                // return the height of the node
```

Unfortunately, making T left-heavy does not necessarily minimize the cost. In fact, it may even increase it, as illustrated in Fig. 12. Hence, we propose a different approach for cost minimization, which we call Flipper. It uses a more complex rule for deciding which nodes to pivot. We propose below a detailed implementation of Flipper and provide guaranteed upper bounds on the cost of the resulting Optimized Blist Forms (OBFs).

As motivation, we start by illustrating (Fig. 13) the benefit of Flipper over the left-heavy pivoting strategy.

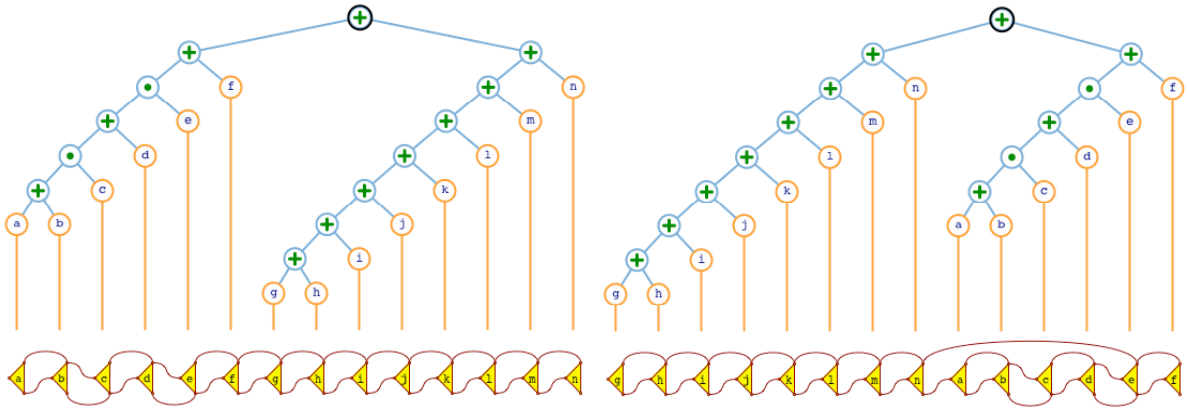

Fig. 12: Pivoting ((a+b)c+d)e+f+(g+h+i+j+k+l+m+n) which has a cost of 2 (left) to make it left-heavy produces g+h+i+j+k+l+m+n+(((a+b)c+d)e+f) which has a cost of 3 (right).

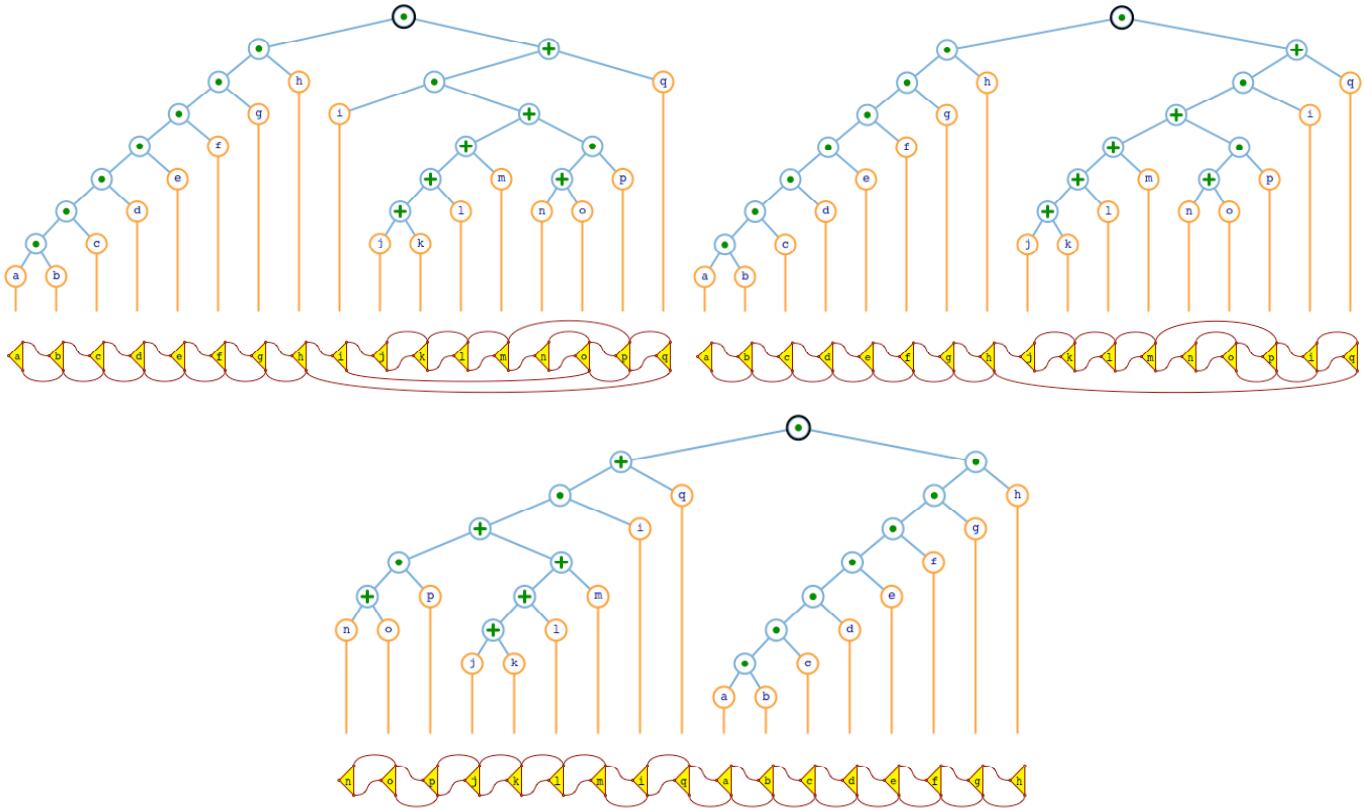

Fig. 13: Pivoting abcdefgh(i(j+k+l+m+(n+o)p)+q) which has a cost of 5 (top left) to make it left-heavy produces abcdefgh((j+k+l+m+(n+o)p)i+q) which has a cost of 4 (top right). Optimization produces (((n+o)p+(j+k+l+m))i+q)(abcdefgh), which has a cost of 2 (bottom).

3.3 Price-tags

To optimize the cost and to prove the corresponding worst-case bounds, we associate with each node a Boolean tt and three costs: c1, c2, and c3. Their values (collectively called the ***price-tag***) are printed in the various figures below to the right of the corresponding node as a 4 characters with no separators. The first character is 't' when tt is *true* and 'f' otherwise. The other three are each a digit representing the integer value of costs c1, c2, and c3 respectively.

To explain the intuition behind these three costs, consider first the expression R=(b+c)(d+e) in Fig. 13 (left). It has three intervals between consecutive gates: (a,b), (b,c) (c,d). The costs c1, c2, and c3 respectively measure the number of lines in these intervals. Hence the 3 costs of the top node representing R are 223. Now, consider that R is the right argument

of expression E=L+R, where L=a, as shown in Fig. 13 (center). Notice that the costs of intervals (b,c) and (c,d) has increased by 1, while the cost of (d,e) has not. Now, consider that R is the right argument of expression E=LR, where L=a, as shown in Fig. 13 (right). Notice that the cost of interval (b,c) has increased by 1, while the costs of (c,d) and (d,e) has not. These increases do not depend on L. Note that c1 is increased in both L+R and LR configurations. Also, note that c3 is never increased. Finally, note that c2 is increased in L+R, but not in LR. To reflect this bias, we set tt=*true*.

More generally, we classify the intervals of an arbitrary expression R in 3 sets of intervals by considering whether their cost has increased in L+R and/or in LR. First consider all the intervals for which cost increases both in L+R and in LR. Notice that, if they exist, they are consecutive and found at the beginning of R. The cost c1 associated with R and denoted R.c1 is the maximum of their costs. Now consider all the intervals for which cost remains constant both in L+R and in LR. Notice that, if they exist, they are consecutive and at the end of R. The cost R.c3 is the maximum of their costs. Finally, consider the remaining intervals. Note that they are also consecutive in R. The maximum of their cost is R.c2. The Boolean associated with R is set as follows: R.tt=tr*u*e if the cost of these (c2) intervals would be increased in LR, tt=*false* if the cost of these intervals is increased in L+R. The price-tag of a leaf-node is initialized to t102.

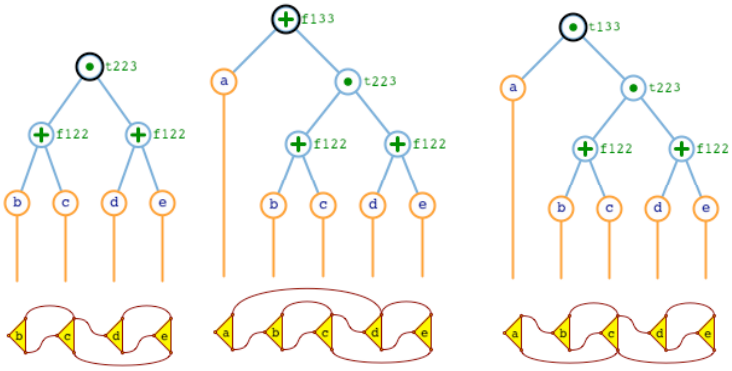

Fig. 13: Let L=a. The expression R= (b+c)(d+e), shown left, can be merged into L+R= a+(b+c)(d+e), shown center, or into LR= a((b+c)(d+e)), shown right. R.c1=2 is the cost of interval (b,c) because it increases both in LR and L+R. R.c3=3 is the cost of interval (d,e) because it does not increase in LR or L+R. R.c2=2 is the cost of the remaining interval (c,d). Since that cost increases for L+R, R.tt=*true*. Hence, the price-tag of R is t223.

A more complex example with several intervals for each cost is shown in Fig. 14.

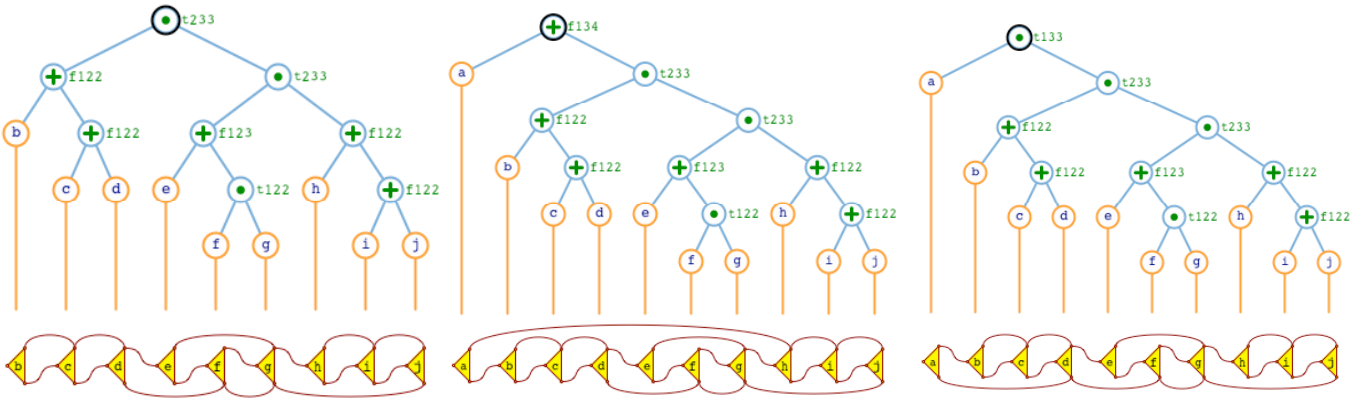

Fig. 14: R=(b+(c+d))((e+fg)(h+(i+j))), shown left, can be merged into L+R= a+(b+(c+d))((e+fg)(h+(i+j))), shown center, or into LR= a((b+(c+d))((e+fg)(h+(i+j)))), shown right. Cost R.c1=2 is the cost of intervals (b,d) because it increases both in LR and L+R. R.c3=3 is the cost of intervals (h,j) because it does not increase in LR or L+R. Cost R.c2=3 is the cost of the remaining intervals (d,h). Since R.c2 increases for L+R, R.tt=*true*.

3.4 The Flipper algorithm

Flipper redefines the ***cost*** for a node P as max(P.c1,P.c2,P.c3). We want to find a set of pivots that will minimize the cost of the root. There are n–1 op-nodes and hence 2^{n-1} different sets of pivots. Clearly, this large number of options prohibits an exhaustive search for complex expressions.

The Flipper algorithm, which we introduce here is a greedy optimization algorithm. It uses a recursive traversal of ***T*** and decides which node to pivot in a bottom-up order. While doing so, and after the possible pivot, it updates the price-tag of each op-node P using the price-tags of its left and right children, L and R.

Our implementation of Flipper does the following. If a leaf-node is reached, the price-tag is simply set to t102. Otherwise, assume that we have reached an op-node P with children L and R. A recursive call to Flipper for L and R, optimizes them. Then, Flipper computes the price-tag of P by a call to pricetag and computes the originalCost by a call to costMeasure. Then it pivots P by a call to flip, updates the price-tags and computes the flippedCost. If the originalCost is not larger than the flippedCost, Flipper pivots P back and updates the price-tag again.

```
void flipper() {if(O==' ') {c1=1; c2=0; c3=2; tt=true;}
    else {L.flipper(); R.flipper();
    pricetag(); int originalCost=costMeasure();
    flip(); pricetag(); int flippedCost=costMeasure();
    if (originalCost<=flippedCost) {flip(); pricetag();}; }; }
```

The procedure flip simply swaps references to the two children nodes using an auxiliary variable N.

```
void flip() {node N; N=L; L=R; R=N;  }
```

To compute the price-tag of P from the price-tags of its children L and R, we proceed as follows. The price-tag variables of P are identified by tt, c1, c2, and c3, while the same variables for L are identified by L.tt, L.c1, L.c2, and L.c3, with the same convention for R. When the operator O of P is '•', we set tt=*true* and consider 4 cases (corresponding to the four combinations of truth-values of L.tt and R.tt). The formulae c1, c2, and c3 are derived by considering the four cases illustrated in Fig. 15 top from left to right and corresponding to the four lines in pricetag top-to-bottom. The four cases for a '+' operator are symmetric and are shown in Fig. 15 bottom.

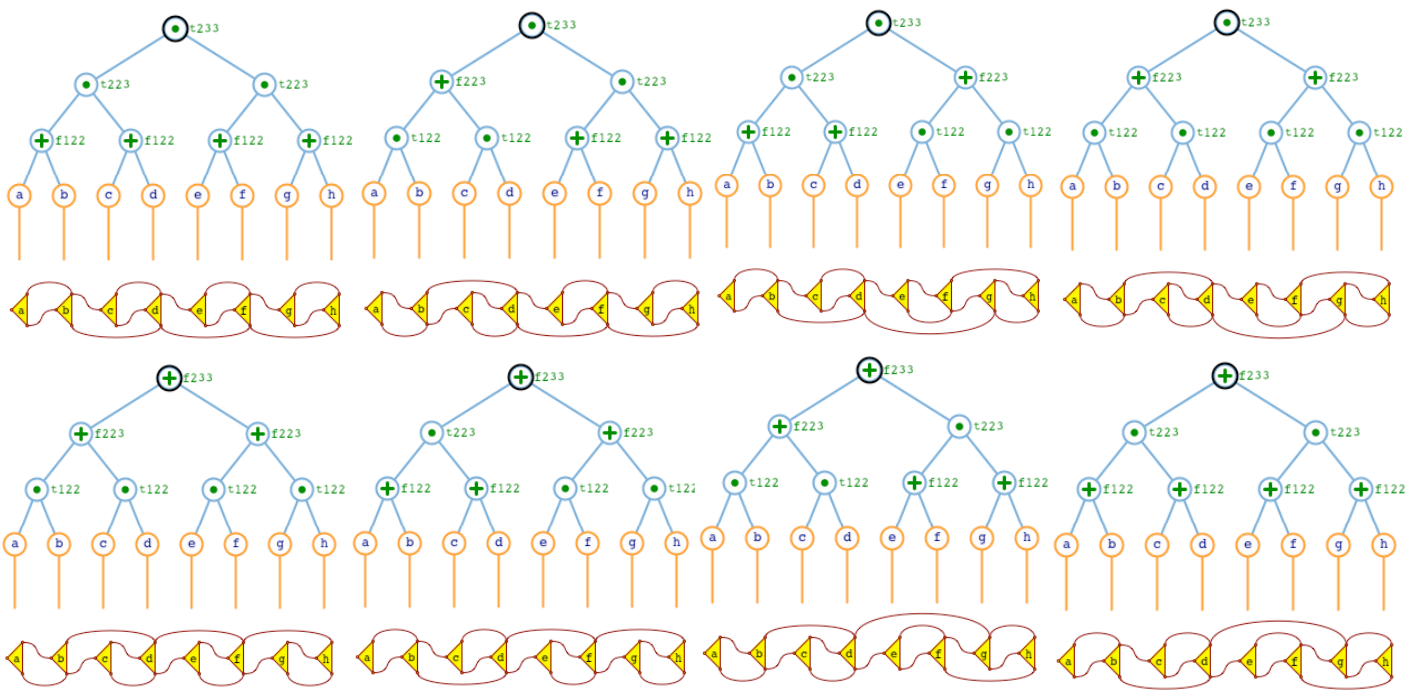

Fig. 15: Form left to right: In (a+b)(c+d)((e+f)(g+h)) top and ab+cd+(ef+gh) bottom, both L.tt and R.tt are *true*. In (ab+cd)((e+f)(g+h)) top and ab+cd+(e+f)(g+h) bottom, L.tt is *false* and R.tt is *true*. In (a+b)(c+d)(ef+gh) top and ab+cd+(e+f)(g+h) bottom, L.tt is *true* but R.tt is *false*. In (ab+cd)(ef+gh) top and (a+b)(c+d)+(e+f)(g+h) bottom, L.tt and R.tt are *false*.

```
void pricetag() {
   tt=O=='•';
   if((tt&& L.tt&& R.tt) || (!tt&&!L.tt&&!R.tt)) { c1=L.c1; c2=max(max(L.c2,L.c3),max(R.c1+1,R.c2)); c3=R.c3;};
   if((tt&& L.tt&&!R.tt) || (!tt&&!L.tt&& R.tt)) { c1=L.c1; c2=max(L.c2,L.c3,R.c1+1); c3=max(R.c2+1,R.c3);};
   if((tt&&!L.tt&& R.tt) || (!tt&& L.tt&&!R.tt)) { c1=max(L.c1,L.c2); c2=max(L.c3,R.c1+1,R.c2); c3=R.c3;};
   if((tt&&!L.tt&&!R.tt) || (!tt&& L.tt&& R.tt)) { c1=max(L.c1,L.c2); c2=max(L.c3,R.c1+1); c3=max(R.c2+1,R.c3);};    }
```

The crux of the Flipper soluiton lies in the decision to flip or not a given node P. The decision is encoded in the function costMeasure, which uses c1, c2, and c3 to compute a measure of cost which defines whether we should pivot the node P or not.

Since we wish to minimize the maximum of the number of lines used in any interval, a naïve solution that comes to mind immediately is to pivot an op-node P if the pivot reduces the cost of P defined earlier as max(c1,c2,c3). Hence, the naïve implementation of costMeasure is:

```
int costMeasure (int c1, int c2, int c3) { return  max(c1,c2,c3); }
```

Unfortunately, this naïve solution does not work. In fact, it may actually increase the cost, as shown in Fig. 16.

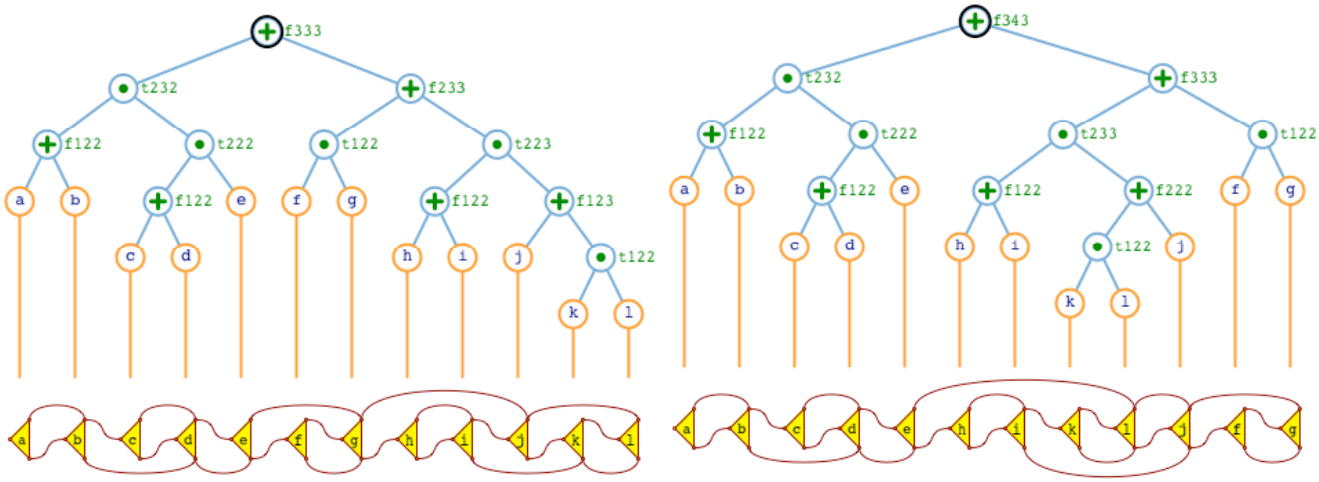

Fig. 16: The use of the naïve costMeasure for (a+b)((c+d)e)+(fg+(h+i)(j+kl)), which has a cost of 3 (left), produces (a+b)((c+d)e)+((h+i)(kl+j)+fg), which has a cost of 4 (right).

Before we describe our correct solution, let us investigate why the naïve approach fails. Notice that it pivots j+kl, which has price tag f123, into kl+j, which has price tag f222, because this pivot decreases the max of the three costs from 3 to 2. This pivot is correct. However, the naïve approach fails to pivot (h+i)+(kl+j), which has a price-tag of t233 into (kl+j)+(h+i), which has a price tag t223, because this pivot would not reduce the max of the three costs, which is 3 in both cases.

To overcome this limitation, we propose to use instead a modified costMeasure, which (as in the naïve implementation) performs a pivot when such a pivot reduces the cost of the node. However, when the original and pivoted versions of the node have the same cost, our modified version of costMeasure uses the other two costs to decide whether to pivot or not.

More specifically, and assuming for simplicity that we have less then 2^{100} literals, we compute the costMeasure as follows.

```
int costMeasure() {int cmax=max(c1,c2,c3); int cmin=min(c1,c2,c3); int cmid=c1+c2+c3-cmax-cmin;
     return  cmax*10000+cmid*100+cmin;}
```

This modified solution often improves upon the naïve solution, as illustrated in Fig. 17.

Notice that the space and time computational complexity of Flipper is linear in the number *n* of literals.

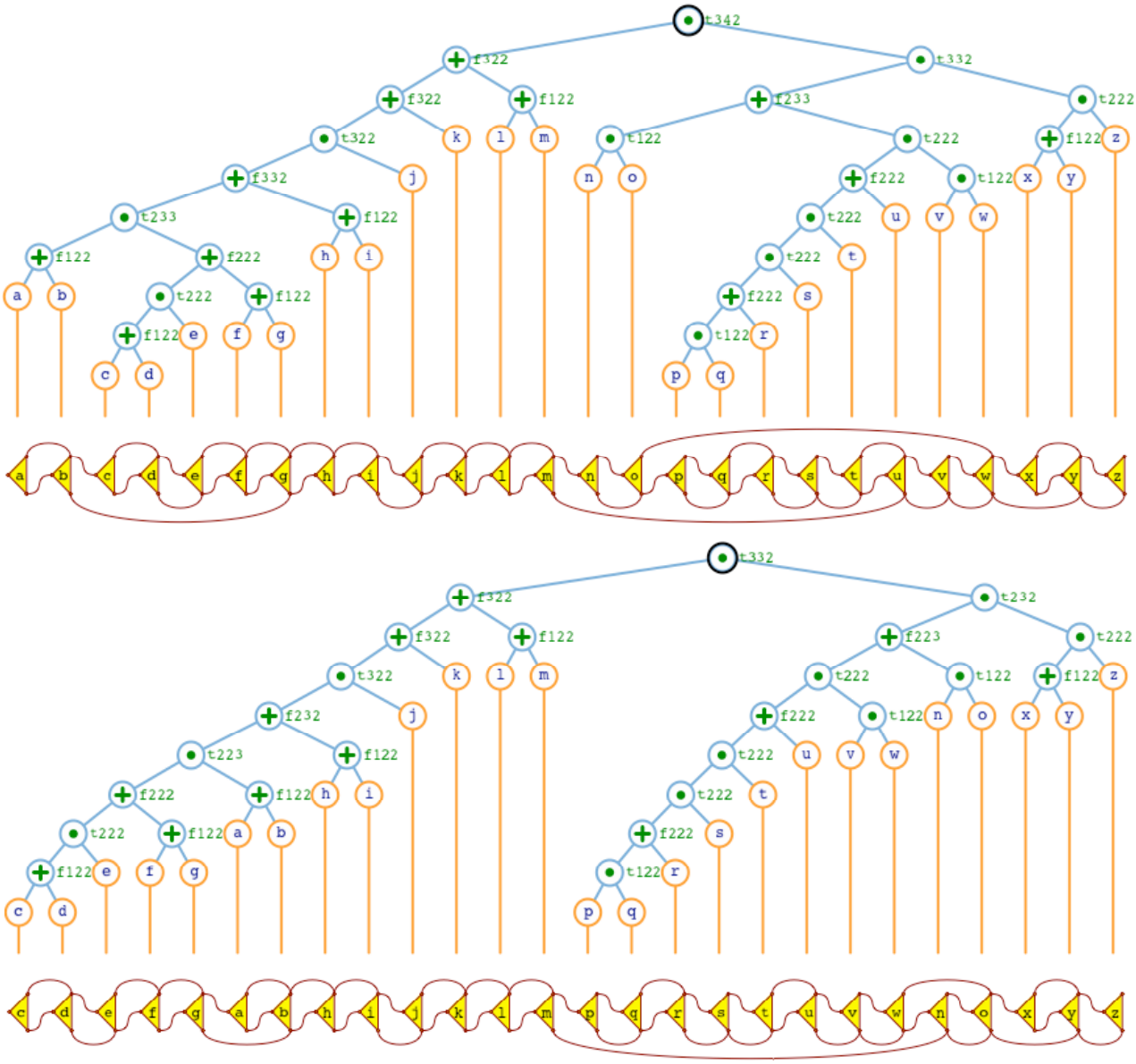

Fig. 17: (((a+b)((c+d)e+(f+g))+(h+i))j+k+(l+m))((no+((pq+r)st+u)(vw))((x+y)z)) has cost 4 (top), not improved by naïve optimization. Flipper yields ((((c+d)e+(f+g))(a+b)+(h+i))j+k+(l+m))((((pq+r)st+u)(vw)+no)((x+y)z)), which has a cost of 3.

3.5 Minimal Flipper-Resilient Expressions (MFRE)

The remainder of the section is dedicated to establishing an upper bound of the cost, as a function of *n*, of the OBF of an expression E that has been optimized by Flipper.

To establish a worst-case lower bound on the cost, we first establish the worst-case upper bound on the lowest number m(c) of literals that commend a given cost c. Hence, we consider only expression that are ***flipper-resilient*** (i.e., whose cost is not altered by Flipper, or more generally, whose cost cannot be reduced by any pivot operation). Furthermore, we focus on flipper-resilient expressions that have the lowest number of literals for a given cost c. Specifically, we say that an expression ***E*** is a ***Minimal Flipper-Resilient Expression*** with cost *c* (abbreviated MFRE(*c*)) when no flipper-resilient expression with fewer literals exist. One can easily verify by exhaustive inspection that for each value of *c* in {2, 3, 4} there are only two forms of MFRE(*c*). They are shown in Fig. 18.

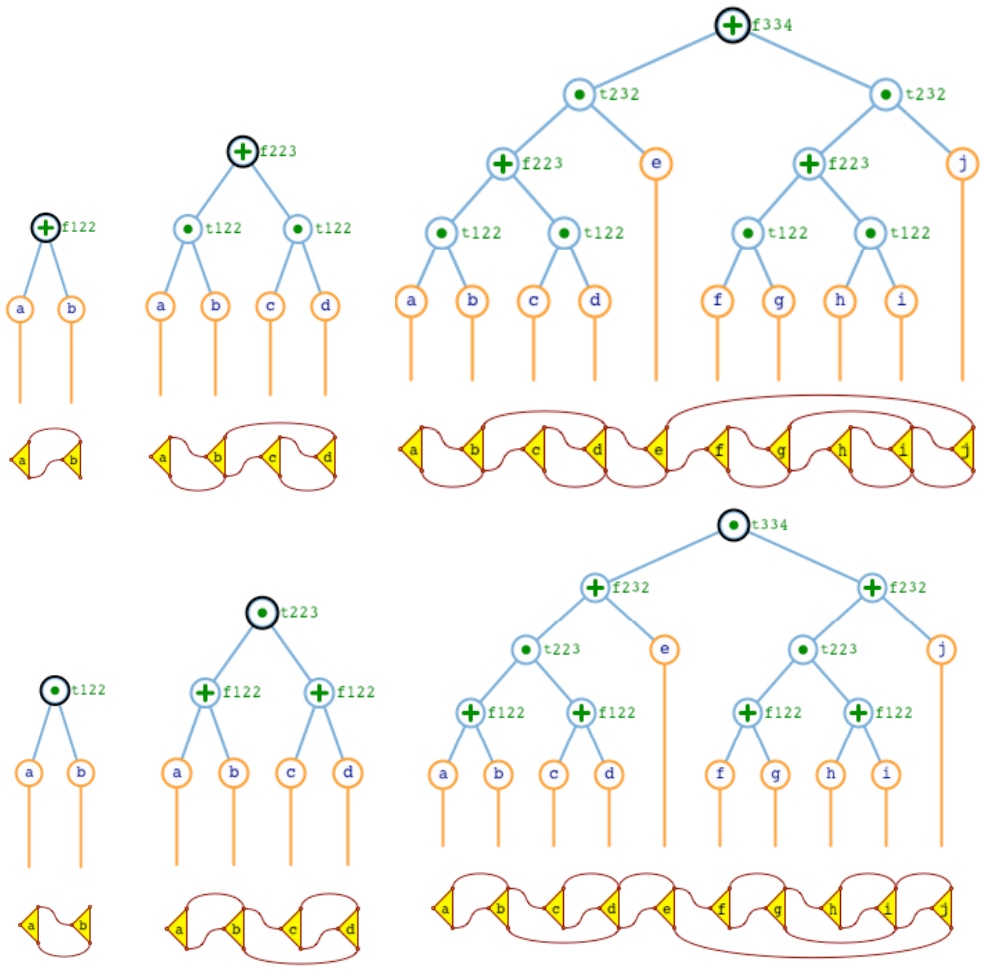

Fig. 18: Top, from left, a+b is MFRE(2), ab+cd is MFRE(3), and (ab+cd)e+(fg+hi)j is MFRE(4). Bottom: ab is MFRE(2), (a+b)(c+d) is MFRE(3), and ((a+b)(c+d)+e)((f+g)(h+i)+j) is MFRE(4).

Our task then is to identify the possible forms of MFRE(c) for arbitrary c larger than 4.

First, for reference, in Fig. 19 we show several simple expression that are not Flipper-resilient, and in Fig. 20, expressions that are Flipper-resilient, but (somewhat surprisingly) not MFRE.

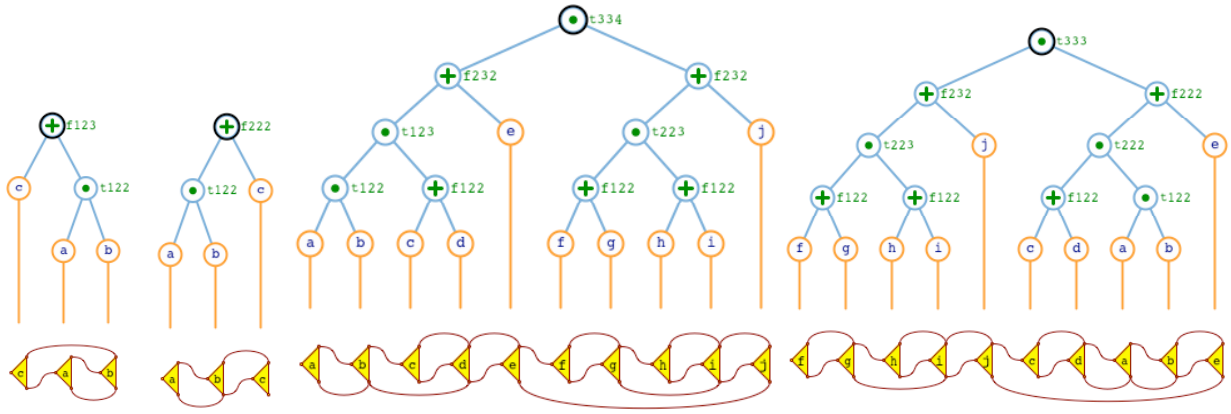

Fig. 19: Expressions that are not Flipper-resilient and their Fipper-optimized version, from left: c+ab with a cost of 3 and its optimized form ab+c with a cost of 2; (ab(c+d)+e)((f+g)(h+i)+j) with a cost of 4 and its optimize version ((f+g)(h+i)+j)((c+d)(ab)+e) with a cost of 3.

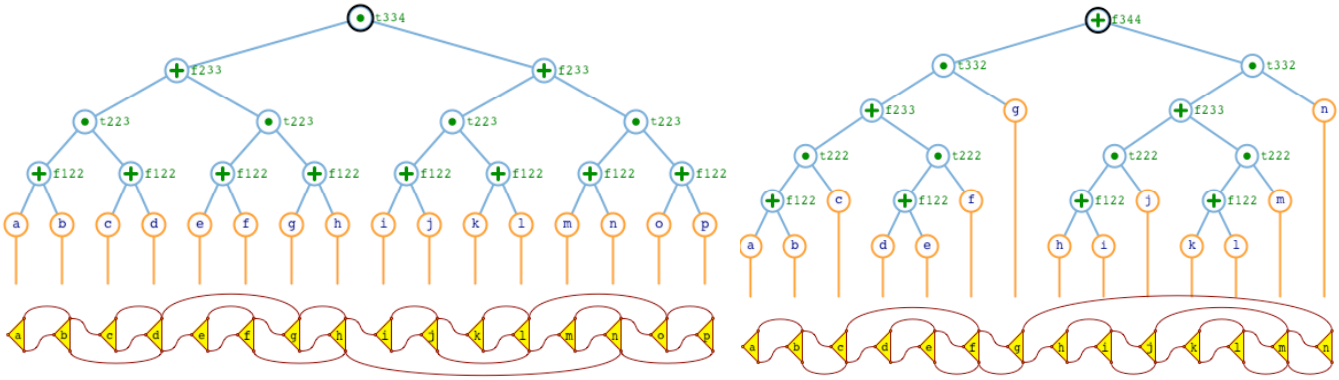

Fig. 20: Expressions that are flipper-resilient, but not MFRE: ((a+b)(c+d)+(e+f)(g+h))((i+j)(k+l)+(m+n)(o+p)), which has a cost of 4 and 16 literals and ((a+b)c+(d+e)f)g+((h+i)j+(k+l)m)n which has a cost of 4 and 14 literals.

Notice that the cost of an expression may always be attributed to an ***alternating zigzag*** in some path from the root to a leaf, where the nature ('+' or '•') of the operator oscillates as one moves along the path. In fact, each back-and-forth oscillation increases the cost by 1. An example of the minimum-cost tree with a long alternating zigzag is shown in Fig. 21 (left). This is a minimum-cost expression because any pivot or operator change will reduce the cost. For example, the cost of most such expressions is reduced to 2 by Flipper, as shown in Fig. 21 (right). There are two forms of such alternating zigzags, one starting with a '•' as shown in Fig. 20, and one starting with a '+' (not shown).

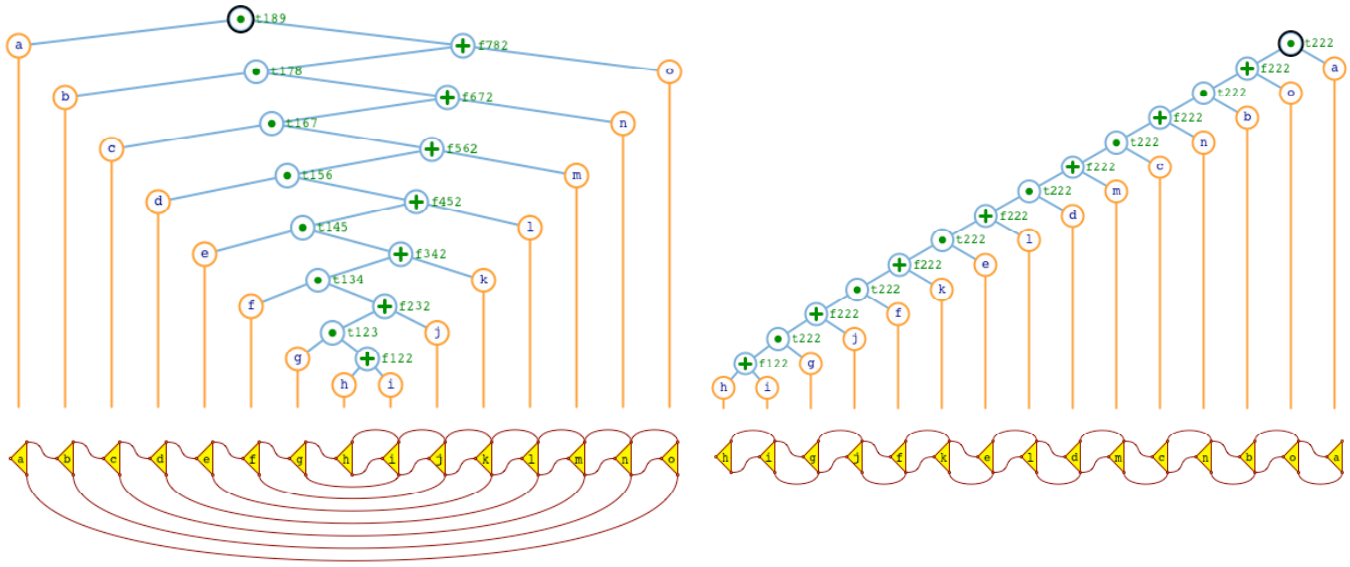

Fig. 21: The expression a(b(c(d(e(f(g(h+i)+j)+k)+l)+m)+n)+o), shown left, is an alternating zigzag with cost 9. Its Fipper optimized version, (((((((h+i)g+j)f+k)e+l)d+m)c+n)b+o)a, has cost 2.

Now, we investigate how to extend these alternating zigzags, by expanding some leaves into sub-trees, to make them flipper-resilient. We also want to use as few additional literals as possible, so as to make the expressions MRFE.

Consider a zigzag that starts with '•', as shown above in Fig. 21 (left). To be Flipper-resilient, each left child L of a '•' op-node P must be expanded into a sub-expression that has the same cost as the right child R of P. Using induction, assume that R is an MFRE(*k*). Then, L must be an MFRE(*k*). (If L had a lower cost than R, P would not be Flipper-resilient and if L had a higher number of literals than R, P would not be an MFRE.) We have pointed out (above) that there are two forms of MFRE(*k*) for k=2, 3, and 4, one rooted with an '•' op-node and the other with a '+' op-node. We can verify easily that, to ensure that P (which is a '•') has higher cost than R, L must be a MFRE with a '+' root op-node. Hence, by induction, we conclude that L must be identical to R.

This conclusion implies that there are two general form of MFRE(*n*). One is illustrated in Fig. 22 for *n*=5. The other one is derived from it by exchanging all '•' and '+' operators.

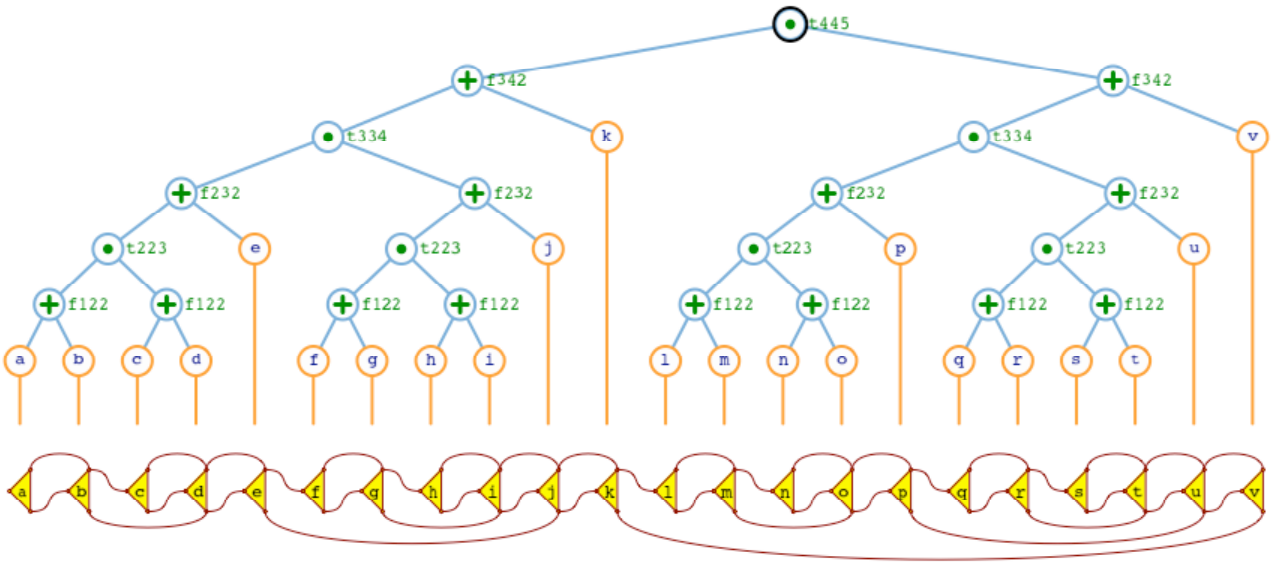

Fig. 22: (((a+b)(c+d)+e)((f+g)(h+i)+j)+k)(((l+m)(n+o)+p)((q+r)(s+t)+u)+v) is one form of an MFRE(5). The other form is obtained by swapping all operators.

3.6 Number of literals in an MFRE(c) and worst case bound on the cost as a function of the number of literals

Now, we must develop an expression of the number m(c) of literals in an MFRE(c). From the form of the general expression, as illustrated in Fig. 21, we note that, $m(c+1)=2(m(c)+1)$. For example, $m(2)=1$, $m(3)=4$, $m(4)=10$, and $m(5)=22$. This recurrence relation yields $m(c)=3\times2^{c-2}-2$, which may be proven by induction: The formula is true for c=2 since $3\times2^{2-2}-2=1$. Assuming that it holds for c, we prove that it must hold for c+1 by noticing that $m(c+1)=2(m(c)+1)=2(3\times2^{c-2}-2+1)= 3\times2^{c+1-2}-2$. Consequently, the number *o*f lines required by an OBE of n literals is less or equal to $j=\lceil\log_2(2n/3+2)\rceil$. For illustration, for each number c of lines between 2 and 20, we list below the numbers m such that no OBE with $n\leq m$ requires more than j lines. For example, all expressions with up to 93 literals require at most 6 lines.

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
3	9	21	45	93	189	381	765	1533	3069	6141	12285	24573	49149	98301	196605	393213	786429	1572861

4. APPLICATION TO CSG RENDERING

Digital models of complex solid parts, such as those found in mechanical assemblies, may often be conveniently specified by combining ***primitive*** solids (blocks, cylinders, spheres…) through regularized set theoretic Boolean operations (union, intersection, difference). The result of such a design process is a Constructive Solid Geometry (CSG) model [Requ80], which represents the desired solid as the topological closure of the interior of a set theoretic Boolean expression that combines primitive solids (literals).

Note that the set theoretic Boolean expressions of CSG model may be parsed into a binary tree ***T***, where each leaf (literal) corresponds to a primitive. Note that even though the same primitive may appear in several places in the CSG model, each instance has typically a different position and orientation and may yield a different classification of a surfel candidate. Hence, it is necessary to treat each instance as a different literal.

The tree ***T*** in fact represents a Boolean expression ***E***, obtained by replacing the set theoretic operators by their Boolean equivalent (union by +, intersection by •, and difference by •!) and by associating a truth value with each leaf. As we did before, and without loss of generality, we assume that the CSG expression is converted into a positive form Boolean expression ***E*** of n literals.

Consider a candidate point q that does not lie on the boundary of any primitive. The point-in-primitive classification of q with respect to a primitive P returns a Boolean (truth value) which is *true* when q∈P and false otherwise. These classification results with respect to each CSG primitive define the truth-values of the literal. To obtain the classification

of q with respect to the CSG solid, (i.e., to decide whether q is in the CSG solid or not) it suffices to evaluate ***E***. If q lies on the boundary of a primitive Q of the CSG solid, we cannot use the above approach, because we do not have a truth-value for literal Q and because replacing it by *true* or by *false* will in general not yield the correct result. In such cases, we classify q against the CSG expression of the Active Zone [RoVo88, Ross96] of Q in the CSG expression, which represents the portion of space where the boundary of Q contributes to the boundary of the solid.

Although many applications require computing a boundary representation (BRep) of the CSG model [ReVo85] , the associated process is expensive and delicate, both numerically and algorithmically. Shaded images of a CSG model may be produced in realtime directly from the CSG representation, avoiding the problems of the BRep computation [RoRe86]. Such direct CSG algorithms exploit the speed of contemporary graphics adapters, which operate on pixels in parallel. Although many variations have been proposed, most approaches generate a set of ***candidate surfels*** (surface samples and associated color values densely distributed on the boundaries of the primitives) and **classify** them so as to identify which are on the boundary of CSG model. The usual z-buffer test is used to discard the occluded ones. Fig. 23 illustrates this process.

Fig. 23: Successive layers obtained by a front-to-back peeling order are trimmed by classifying their surfels against the CSG model. Retained surfels are incorporated in the final image (left-to-right) using a depth-test.

Because a primitive may be self-occluding, the candidate surfels on a primitive Q are produced in layers (in front-to-back "peeling" order using a depth-interval buffer [RoWu92] or in rasterization order using a counter [GHF86]) by having the graphics hardware rasterize a triangulation of Q.

To classify all the candidate surfels of a layer of Q in parallel against a primitive P, we rasterize P and keep track (for each candidate surfel) of the parity of the number of times it is occluded by a triangle of P. Odd parity indicates that the surfel is in P. Even parity indicates that it is out. (The delicate on/on cases where a candidate surfel of Q happens to also lie on P are handled by offsetting the candidate surfels by a small distance away from the viewer [RoWu92] or by assigning an artificial depth order to primitives [HaRo07].) One stencil bit per pixel is used to efficiently keep track of the parity (i.e., classification) information. Another stencil bit per pixel is used to distinguish pixels that have a candidate surfel from locked pixels. Unfortunately, there are typically only 8 such stencil bits per pixel and copying their values to texture memory is slow. Therefore, unless the CSG expression is trivial, one cannot afford to store all of the surfel/primitive classifications truth-values for each pixel at which a candidate surfel is present. In fact, we must combine these truth-valies into a final result using only these 6 stencil bits as working memory (footprint) for each pixel. Specifically, we combine the surfel/primitive classification results according to the Boolean expression of the CSG model [HaRo05] or preferably [HaRo07] of the active zone [RoVo88] Z of Q.

We can perform this surfel classification using a Blist form of ***E***. As explained before, Blist associates with each primitive (literal) P of ***E*** three integer labels: P.i (the ID representing the identifier of P), P.t (the ID representing the next relevant primitive that may affect this pixel if V is true), and P.f (the ID representing the next relevant that may affect this pixel if V is false).

At any given moment, the stencil of each pixel holds:
- a mask bit M indicating whether the pixel contains a candidate surfel or is locked
- a classification (parity) bit V indicating whether the candidate surfel is in the primitive P
- a label *next* stored in the remaining 6 bits indicating the name of the next relevant primitive

For each literal P in ***E***, we do the following:

1. We rasterize all the triangles of P. During that rasterization, each time a triangle of P covers each pixel where M==true and occludes the surfel stored at that pixel (depth test) we toggle the V bit of the pixel.
2. We broadcast (P.i, P.t) and at each pixel do: if (M && (P.i== *next*) && V) *next* =P.t .
3. We broadcast (P.i, P.f) and at each pixel do if (M && (P.i== *next*) && V) *next* =P.f .

In the end, we know that the surfel candidates of pixels where *next*==0 are in the CSG model. For example, if we were classifying surfels on Q against the active zone of Q, then surfels that pass this classification are on the solid. So, in fact, the Blist evaluation for this application may be viewed as an example of clocked ***sequential logic***.

The problem of course lies in the fact that we have only 6 stencil bits for storing the ID *next* at each pixel, and can hence only accommodate 2^6=64 different ID. If we use the OBF version of the Blist, b stencil bits suffice to support all CSG trees with $3\times2^{c-1}-1$ primitives, where $c=2^b$. For example, 2 stencil bits suffice to correctly process all CSG expression with up to 21 primitives, 3 stencil bits suffice for expressions with up to 381 primitives, 4 stencil bitts suffice for up to 98301 primitives, 5 stencil bitts suffice for up to 6.4×10^9 primitives, and 6 stencil bits suffice for all expressions with up to 2.7×10^{19} primitives.

5. Logic Matrix (LM)

We wish to produce a hardware circuit that would establish in one cycle whether a particular Boolean expression ***E*** evaluates to *true* or *false*, given the truth-values of its literals. To do so, we introduce the Logic Matrix (Fig. 24), abbreviated LM. An LM circuit is composed of a series of horizontal lines L_0, L_1… and of a set of gates, each associated with three vertical wires (orange, green, blue). Each gate is associated with a different literal of ***E*** in left-to-right order as the literals appear in ***E***. Hence, we will use the literal name to identify its gate. The state (up/down) of a gate G (black edge) reflects the truth-value (false/true) of the literal. The left-most wire of each gate is the ***in-wire*** (orange). The central wire is the ***true-wire*** (green). The right-most is the ***false-wire*** (blue). The in-wire connects to the true-wire if G is down (*true*) and to the false-wire otherwise. In the example of Fig. 24, Literals A, C, and E are set to *false*. Their gates are up allowing the current arriving at their in-wire (left) to flow to their false-wire (right). Literals B and D are set to *true*. Their gates are down allowing the current arriving at their in-wire to flow to their true-wire (center).

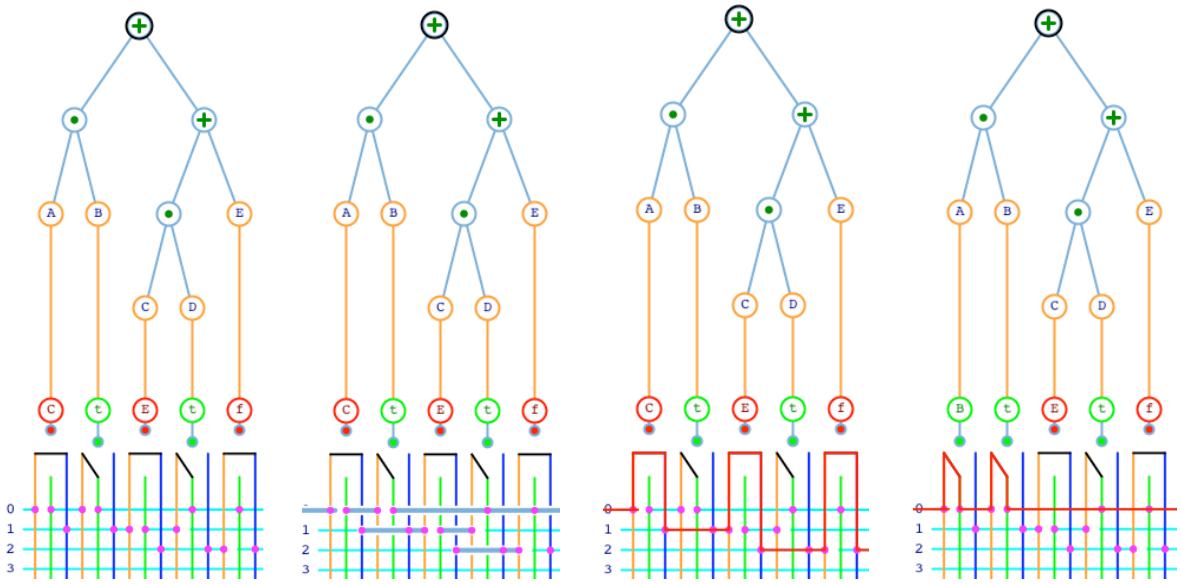

Fig 24: The tree (top) for AB+(CD+E) and (bottom) the corresponding Logic Matrix (far-left). The usable sections of the horizontal lines highlighted (center-left). The red passage of the current for a particular input vector (center-right) indicating that the expression evaluates to *false*. The current for a different input vector (far-right) reaches the right side of the top line L_0, indicating that the expression evaluates to *true*.

Each wire is connected to a horizontal line (magenta dot). Note that the connection of a line L_k with an in-wire of a gate G is always a T-junction, which implies that current arriving from L_k onto the in-wire of G must pass through G. The connection of a line L_k with the true-wire and false-wire of G may be a T-junction or a cross-connection. A cross-junction would allow current arriving on L_k to stay on L_k, hence bypassing G, or to arrive onto L_k through G. Current may flow on these lines between the magenta connections. The usable portions of the lines (where current could flow) are highlighted on Fig. 24 center-left for clarity.

A literal name, say 'R', marked in the circle above a gate G indicates that, given the shown state (truth-value) of G, current arriving on the in-wire of G would eventually reach the in-wire of gate R. A 't' marked in the circle above gate G indicates that if current were to arrive on the in-wire of G, it would reach the right side of the top-most line L_0, regardless of the state of all subsequent gates, indicating that ***E*** evaluates to *true*. An 'f' marked in the circle above a gate G indicates that, given the current state of G, if current were to arrive on the in-wire of G, it would then reach the right side of another line, different from L_0, indicating that ***E*** evaluates to *false* regardless of the state of the other gates. The passage of the current for some initial input vector (truth values of the literals) is shown in red in Fig. 24 center-right. Because in this figure the current does not arrive to the right of L_0, we conclude that, with these input values, ***E*** evaluates to *false*. Flipping the left-most gate (setting the truth-value of literal A to *true*) changes the passage of the current (as shown in Fig. 24 far right), so that it ends on L_0 implying that for the new input vector ***E*** evaluates to *true*.

Note that the literal R marked in the circle above gate G changes with the state of G. We will use G.T to denote its value when G is *true* and G.F otherwise. Hence, G.T is the name of the gate to which current would flow next if it were to arrive on the in-wire of G when G is *true* (down). Similarly, G.F is the name of the gate to which current would flow next if it were to arrive on the in-wire of G when G is *false* (up).

Note that, regardless of the complexity of ***E***, the truth-value of ***E*** is available instantaneously as soon as the gates have commuted to their proper position. Therefore we claim that a LM evaluates any logical expression in a single cycle.

To properly connect gate G, we need to know the indices of the lines where its three wires connect. We will use G.i to denote the index of the line where the in-wire of G connects, G.t to denote the index of the line where the true-wire of G connects, and G.f to denote the index of the line where the false wire of G connects. Hence, the wiring of a LM is entirely defined by these triplets of indices (G.i,G.t,G.f) for each literal G.

These triplets of indices are exactly those derived by the Weaver algorithm described earlier. Hence, the wiring of a Logic Matrix may be trivially derived from the Blist of the expression, as shown in Fig. 25 left. Note the regularity of the MP layout, which is important for scalability and more difficult to achieve for OBDDs [CaKo03].

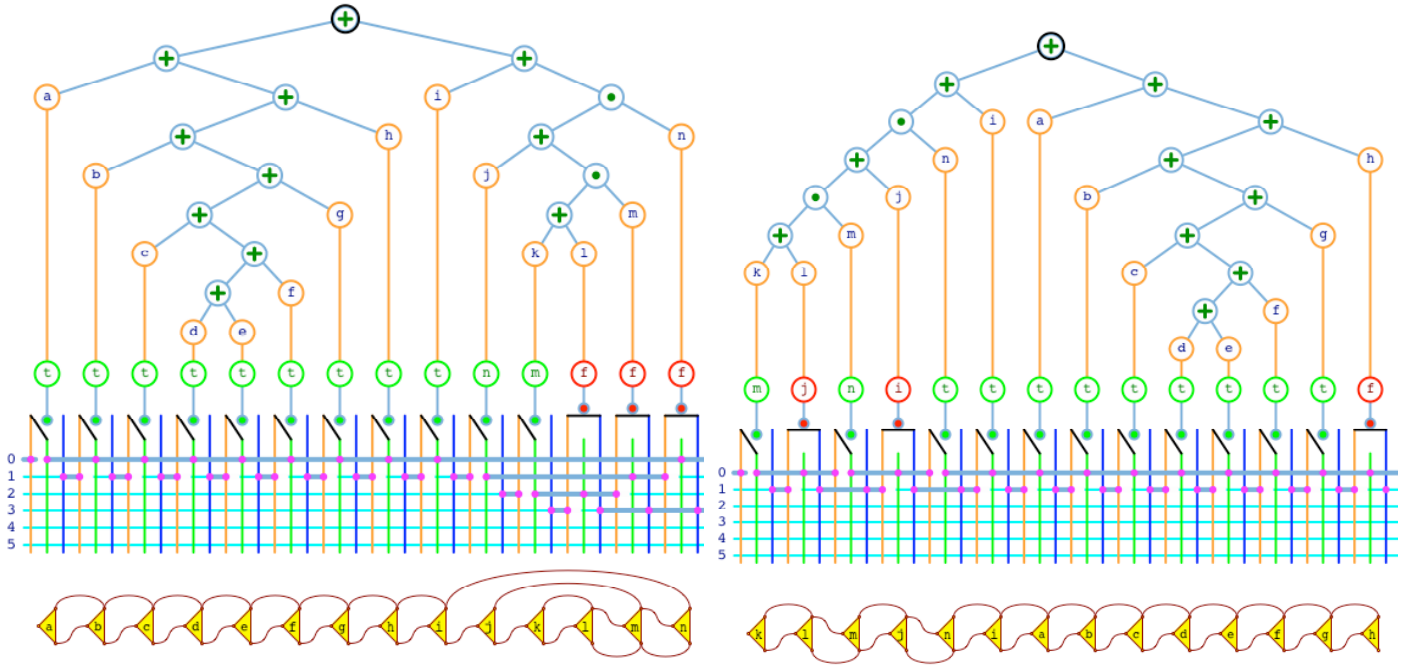

Fig 25: On the left, we show the tree for a+(b+(c+(d+e+f)+g)+h)+(i+(j+(k+l)m)n) with its Logic Matrix (below) and with its Blist (bottom). Notice that the LM gates are drawn up-side-down with respect to the Blist gates: when G is *true*, the Blist gate is up (arriving current flows to the upper output), while the corresponding LM gate is pushed down (arriving current flows to the central green wire, which starts below the gate for visual clarity). Notice that this LM uses 5 lines. The OBF version (shown right) produced by Flipper is ((k+l)m+j)n+i+(a+(b+(c+(d+e+f)+g)+h)). Its LM uses only 2 lines.

Note that in order to accommodate large Boolean expressions in a finite size LM, we would like to reduce as much as possible the number of lines needed. To achieve this goal, we take advantage of the OBF introduced here (Fig. 25 right). Hence, we guarantee that $j=\lceil \log_2(2n/3+2) \rceil$ lines always suffice. As an example, we list here the upper bound n(j) on the number of literals for which j lines suffice: n(2)=3, n(3)=9, n(4)=21, n(5)=45, n(6)=93, n(7)=189, n(8)=381, n(9)=765, n(10)=1,533, n(11)=3,069, n(12)=6,141, n(13)=12,285, n(14)=24,573, n(15)=49,149, n(16)=98,301, n(17)=196,605, n(18)=393,213, n(19)=786,429, n(20)=1,572,861.

6. Logic Pipe (LP)

If a delay between the availability of the input vector and the availability of the result is acceptable, we can reduce the footprint of the Logic Matrix by using the ***Logic Pipe*** (LP) introduced here. Essentially, the LP mimics the Blist diagram. However, instead of using j lines, as in an ML layout, where at any moment, the current passes through a single line in each segment, the LP uses only $\lceil \log_2(j) \rceil$ lines, where $j=\lceil \log_2(2n/3+2) \rceil$, which together transmit the ID of the next literal from one gate to the next (Fig. 26). For reference, we list the upper bound n(j) on n for which j lines suffice: n(2) = 21, n(3) = 381, n(4) = 98301, n(5) = 6.44×10^9, n(6) = 2.76×10^{19}.

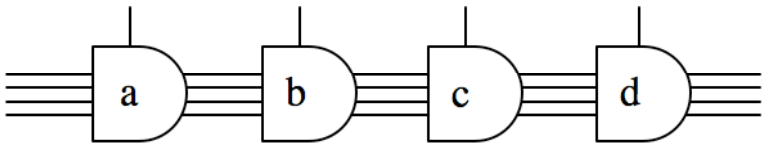

Fig 26: In the Logic Pipe, $\lceil \log_2(j) \rceil$ lines are used transmit the *next* ID from one gate to the next.

As before, each gate G is associated with three IDs: its name G.i, the name G.i of the next gate to process the input if G is *true*, and the name G.f of the next gate to process the input if G is *false*. Let *nextIn* be the ID encoded on the lines incoming to G and *nextOut* be the ID encoded on the lines leaving from G. *nextOut is set using the following rule:*

```
If (G.i==nextIn) if (G.V) nextOut=G.t else nextOut=G.f else nextOut=nextIn;
```

The rule may be implemented using an MP for each output line and hence evaluated in one cycle. Consequently, we can, at each cycle, produce the value of ***E*** for a different input vector. To acieve this, we stagger the input. For example, the different phases of the staggered input vectors will be {a0,,,}, {a1,b0,,}, {a2,b1,c0,}, {a3,b2,c1,d0}, {a4,b3,c2,d1}, {a5,b4,c3,d2}… After the first staggered input sets, at each clock cycle, the output of d contains the consecutive values of E for the consecutive input vectors {a0,b0,c0,d0}, {a1,b1,c1,d1},{a2,b2,c2,d2}…

7. Conclusions

We have introduced an optimization process, which transforms an arbitrary positive-form Boolean expression ***E*** of n literals into its OBF (Optimized Blist Form). The process involves a preprocessing phase (Flipper), which pivots selected op-nodes, and a conversion phase (Weaver) that associates three integer labels with each literal in the expression. Both Flipper and Weaver have linear space and time complexity and can be implemented with a few lines of code (included here). The OBF may be used in different way to compute the truth-value of ***E***.

If we wish to evaluate ***E*** in software sequentially using as little storage (footprint) as possible, we traverse the literals of ***E*** one by one and occasionally replace the content of the footprint with one of the three integer labels of the current literal. We have shown that we need at most $\lceil \log_2\lceil \log_2(2n/3+2) \rceil \rceil$ bits in the footprint. For example, a footprint of 2 bits suffices for all expressions of 21 literals or less and 6 bits suffice for all expressions with up to 2.76×10^{19} literals. Hence we have a linear evaluation cost requiring $O(\log_2\log_2 n)$ bits. This, we believe, is a previously unknown result and is hence the main theoretical contribution reported here. The reader should remember that these results are limited to Boolean expressions that can be written in positive-form using n literals. They do not apply to more general Boolean expressions, which may for example include XOR operators or Boolean functions of n variables that cannot be written as a Boolean expression without repeating the name of a variable. Nevertheless, given the tightness on the footprint memory upper bound and the efficiency of the OBF computation, these results may be of value for dealing with such more general expressions or functions, for example by expanding them into positive form or by treating each occurrence of a variable as a different literal. Applying this result to the direct CSG rendering problem permits to shade arbitrarily complex CSG models on the GPU using only 8 stencil bits per pixels.

If we wish to evaluate ***E*** in hardware in a single cycle (no latency), we can use the LM (Logic Matrix) introduced here, which is a matrix of $3n$ wires passing over j lines, with contacts between each wire and one or more lines. We have shown that only $j=\lceil \log_2(2n/3+2) \rceil$ lines are necessary. For example, with 20 lines, we can evaluate all logical expressions with up to 1,572,861 literals.

Finally, if a latency of n cycles is acceptable, we can use a LP (Logic Pipe) with a stream of staggered input vectors, which uses n gates, each one connected to the next one by $\lceil \log_2 \lceil \log_2(2n/3+2) \rceil \rceil$ lines, to produce at each cycle the truth-value of ***E*** for a new input vector. For example, an LP with only 4 lines accommodates all logical expressions with up to 98,301 literals.

The regularity of LM and LP layouts enhance their scalability and application to future technologies.

8. Bibliography

[Aker78] Akers, S. 1978. Binary decision diagrams. *IEEE Trans. Comput*. C-27 (June), 509–516.

[BoWe96] Bollig, B.; Wegener, I. 1996. Improving the variable ordering of OBDDs is NP-complete. Computers, IEEE Transactions on Computers, Volume 45, Issue 9, Sept. Page(s):993 - 1002

[Brya86] Bryant, R. 1986. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Comput*. 35(8):677–691.

[Brya91] Bryant, R. 1991. On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Applications to Integer Multiplication. *IEEE Transactions on Computers*, 40(2)205-213.

[Brya95] Bryant, R. 1995. Binary decision diagrams and beyond: enabling technologies for formal verification. *IEEE/ACM International Conference on Computer-Aided Design*, 236–243.

[CaKo03] Cao, A. and Koh, C.K. 2003. Non-Crossing Ordered BDD for Physical Synthesis of Regular Circuit Structure. Purdue University. http://docs.lib.purdue.edu/ecetr/136. NSF Grant CCR-9984553.

[El&91] Ellis J., Kedem G., Lyerly G. T., Thielman D., Marisa R., Menon J. 1991. The Ray Casting Engine and ray representations. *ACM Symposium on Solid Modeling Foundations and Applications*, 255–268.

[GHF86] Goldfeather, J., Hultquist, J. P. M., Fuchs, H. 1986. Fast constructive solid geometry display in the pixel-powers graphics system. *Annual Conference on Computer Graphics and Interactive Techniques,* 107–116.

[GMTF89] Goldfeather, J., Molnar, S., Turk, G., Fuchs, H. 1989. Near realtime CSG rendering using tree normalization and geometric pruning. *IEEE Computer Graphics and Applications*, 9(3):20–28.

[HaRo05] Hable, J. Rossignac, J. 2005. Blister: GPU-based rendering of Boolean combinations of free-form triangulated shapes, *ACM Transactions on Graphics (SIGGRAPH)*. 24(3):1024–1031.

[HaRo07] Hable, J. Rossignac, J. 2007. CST: Constructive Solid Trimming for rendering BReps and CSG. IEEE Transactions on Visualization and Computer Graphics, 2007 (to appear). GVU Tech Report GIT-GVU-06-16.

[PaMe77] Payne, h., Meisel, w. 1977. An algorithm for constructing optimal binary decision trees. *IEEE Trans. Comput.* TC-26, 9:905–916.

[Requ80] Requicha, A. 1980. Representations for Rigid Solids: Theory, Methods, and Systems. *ACM Comput. Surv*. 12(4):437–464.

[ReVo85] Requicha, A., Voelcker, H. 1985. Boolean operations in solid modeling: Boundary evaluation and merging algorithms. *Proceedings of the IEEE*, 75(1):30–44.

[RoRe86] Rossignac, J., Requicha, A. 1986. Depth-buffering display techniques for constructive solid geometry, *IEEE Computer Graphics and Applications*, 6(9):26–39.

[Ross94] Rossignac, J. 1994. Processing Disjunctive forms directly from CSG graphs, Jarek Rossignac. Proceedings of CSG 94: Set-theoretic Solid Modelling Techniques and Applications, Information Geometers, pp. 55-70, Winchester, UK.

[Ross96] Rossignac, J. 1996. CSG formulations for identifying and for trimming faces of CSG models. In *CSG'96: Set-theoretic solid modeling techniques and applications,* Information Geometers, Ed. John Woodwark. 1–14.

[Ross99] Rossignac, J. 1999. BLIST: A Boolean list formulation of CSG trees, *Technical Report GIT-GVU-99-04*. GVU Center, Georgia Tech. http://www.cc.gatech.edu/gvu/reports/1999/

[RoVo88] Rossignac, J., Voelcker, H. 1988. Active zones in CSG for accelerating boundary evaluation, redundancy elimination, interference detection, and shading algorithms, *ACM Transactions on Graphics*, 8(1):51–87.

[RoWu92] Rossignac, J., Wu, J. 1992. Correct shading of regularized CSG solids using a depth-interval buffer, *Advanced Computer Graphics Hardware V: Rendering, Ray Tracing and Visualization Systems*, Eurographics Seminars, 117–138.

[YaOH97] Yang, B., O'Hallaron, D. 1997. Parallel breadth-first BDD construction. *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming,* 145-156.