Graph-Based Algorithms for Boolean Function Manipulation

Department of Computer Science & Engineering
Konkuk University

Introduction

- Boolean Algebra forms a cornerstone of computer science and digital system
- Many problems in logic design, AI, combinatorics can be expressed as a sequence of a Boolean functions
- **But many of the tasks require NP-Complete or co-NP Complete**
- It makes hard to compare with Brute Force Algorithm

Introduction (Cont.)

- Classical approaches are impractical
- There representation of size is 2^n or more
- Other practical approaches suffer from several drawbacks.
  - 1. Certain common functions still require representations of exponential size.
  - 2. Performing a simple operation yield an exponential representation.
  - 3. None of these are canonical forms —> hard to check equivalence or satisfiability

Introduction (Cont.)

- We tackled these problems. **But unfortunately, it’s not perfect**
- We must choose some **ordering** of the system inputs
- Some functions are highly sensitive to this ordering
- Computing itself is a co-NP complete Problem
- But there is an algorithm to choose efficient ordering with some heuristics

Notation

- We assume the functions to be represented all have the same n arguments, written $x_1, \ \ldots \ , x_n$
- When some argument x_i of function f is replaced by a constant b is called a **restriction** of f and denoted $f|_{x_i=b}$
- $f|_{x_i=b}(x_1, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, b, x_{i+1}, \ldots, x_n)$
- Some argument x_i can replaced by a function g, and this is called composition of f and g
- $f|_{x_i=g}(x_1, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, g(x_1, \cdots, x_n), x_{i+1}, \ldots, x_n)$

Notation (Cont.)

- The **Shannon expansion** of a function around variable x_i is given by

$$f = x_i \cdot f|_{x_i=1} + \bar{x} \cdot f|_{x_i=0} \equiv x_i \to f|_{x_i=1}, f|_{x_i=0}$$

- The **dependency set** of a function f, denoted I_f contains those arguments on which the function depends $I_f = \{i \,|\, f|_{x_i=0} \neq f|_{x_i=1}\}$

- For example, $f : x_1 \cdot x_2 + x_1 \cdot \bar{x_2} + \cdots = x_1 \cdot (x_2 + \bar{x_2}) + \cdots$

- $x_1 \notin I_f$

- **Satisfying set** of function f, denoted S_f, is defined as

$$S_f = \{(x_1, \cdots, x_n) \,|\, f(x_1, \cdots, x_n) = 1\}$$

Representation

- Definition : A function graph is a **rooted, directed graph** with vertex set V containing two types of vertices : $nonterminal$ and $terminal$

- A $nonterminal$ vertex v has an $index(v) \in \{1, \cdots, n\}$, and two children $low(v), high(v) \in V$

- For any nonterminal vertex v, if $low(v)$ is also nonterminal, $index(v) < index(low(v))$

- For any nonterminal vertex v, if $high(v)$ is also nonterminal, $index(v) < index(high(v))$

- A $terminal$ vertex v has a $value(v) \in \{0,1\}$

- Due to the ordering restriction, **function graph form a proper subset of BDD.**

- Also, **function graph must be acyclic**

Representation (Cont.)

- A function graph G having root vertex v denotes a function f_v
- If v is a terminal vertex and $value(v) = 1$, then $f_v = 1$. If $value(v) = 0$, then $f_v = 0$
- If v is a nonterminal vertex with $index(v) = i$, then f_v is the function $f_v(x_1, \cdots, x_n) = \bar{x}_i \cdot f_{low(v)}(x_1, \cdots, x_n) + x_i \cdot f_{high(v)}(x_1, \cdots, x_n)$
- $(x_1, \cdots, x_n)$ is a path, and there is no unreachable vertex

Representation (Cont.)

- Function graph G and G' are $isomorphic$ if there exists a one - to - one function σ from the vertices of G onto the vertices G' that satisfies these constraints

- For any vertex v, $value(v) = value(v')$ if v is terminal vertex, $index(v) = index(v')$, $\sigma(low(v)) = low(v')$ and $\sigma(high(v)) = high(v')$ if v is nonterminal vertex

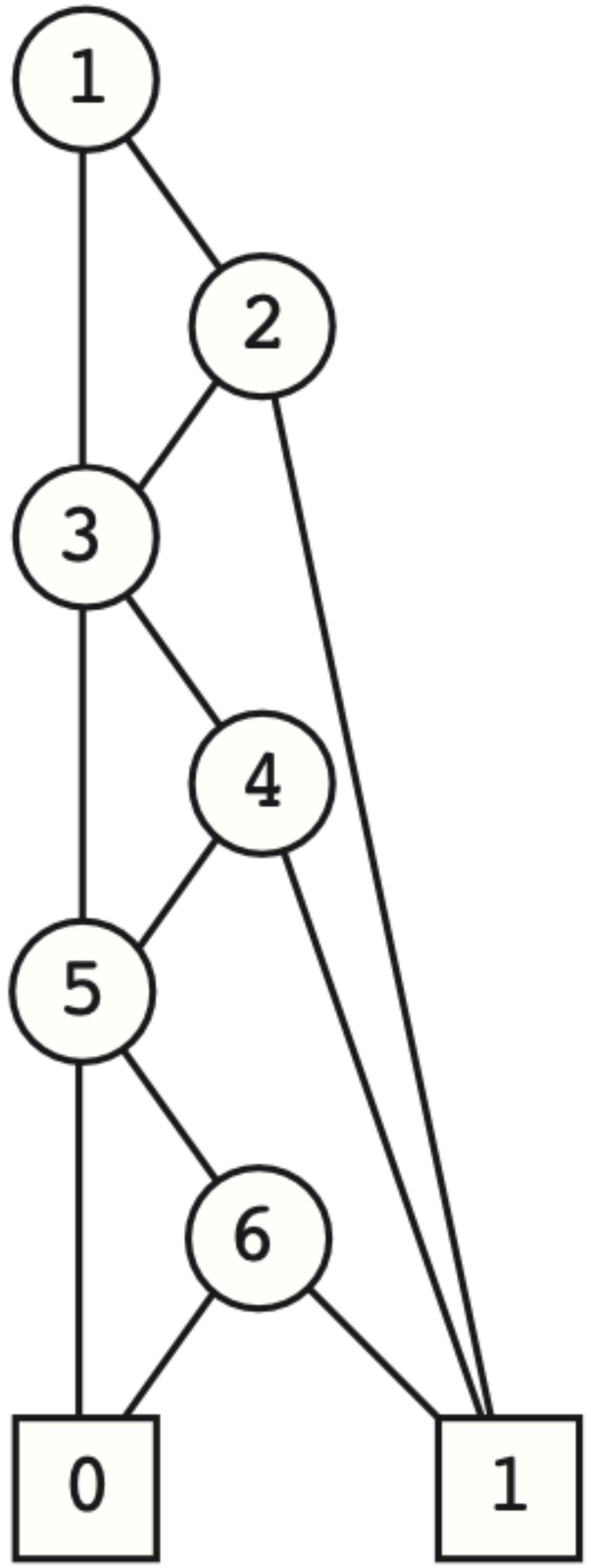
- For any vertex v in a function graph G, the $subgraph\ rooted\ by\ v$ is defined as the graph consisting of v and all of descendants

- If G is isomorphic to G', the subgraph rooted by v is isomorphic to the subgraph rooted by $\sigma(v)$

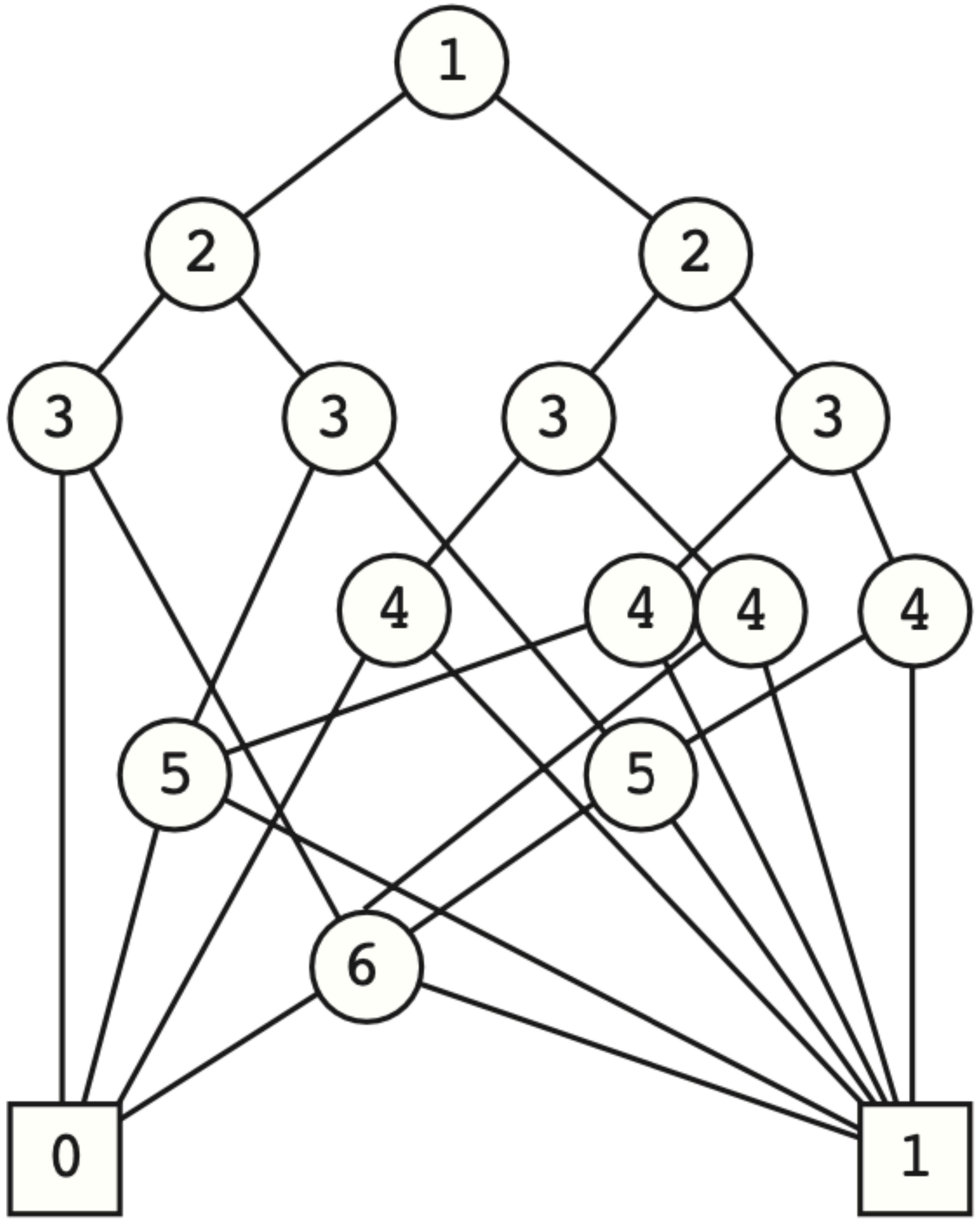
Ordering Dependency

- Ordering is a critical issue in BDD
- For example, function $f = x_1 \cdot x_2 + x_3 \cdot x_4 + x_5 \cdot x_6$ requires 8 vertices, whereas $x_1 \cdot x_4 + x_2 \cdot x_5 + x_3 \cdot x_6$ requires 16 vertices
- It means less information may be stored
- For example, to compute $x_1 \cdot x_2 + \cdots x_{2n-1} \cdot x_{2n}$, we only need preceding pairs information and previous value
- On the other hand, to compute $x_1 \cdot x_{n+1} + \cdots + x_n \cdot x_{2n}$, we need to store n arguments

$x_1 \cdot x_2 + x_3 \cdot x_4 + x_5 \cdot x_6$



$x_1 \cdot x_4 + x_2 \cdot x_5 + x_3 \cdot x_6$



Operations

Procedure	Result	Time Complexity
Reduce	G reduced to canonical form	$O(\lvert G \rvert \cdot \log \lvert G \rvert)$
Apply	f_1 <op> f_2	$O(\lvert G_1 \rvert \cdot \lvert G_2 \rvert)$
Restrict	$f \rvert_{x_i = b}$	$O(\lvert G \rvert \cdot \log \lvert G \rvert)$
Compose	$f_1 \rvert_{x_i = f_2}$	$O(\lvert G_1 \rvert^2 \cdot \lvert G_2 \rvert)$
Satisfy-one	some element of S_f	$O(n)$
Satisfy-all	S_f	$O(n \cdot \lvert S_f \rvert)$
Satisfy-count	$\lvert S_f \rvert$	$O(\lvert G \rvert)$

Table 1.Summary of Basic Operations

Structure Vertex

```
type vertex = record
        low, high: vertex;
        index: 1..n+1;
        val: (0,1,X);
        id: integer;
        mark: boolean;
end;
```

Field	Terminal	Nonterminal
low	null	$low(v)$
high	null	$high(v)$
index	$n+1$	$index(v)$
val	$value(v)$	X

Procedure Traverse

- **This procedure is called at the top level** with the root vertex as argument and **with the mark fields** of the vertices being either **all true or all false**
- We assume that operation will be done in constant time
- Time complexity : $O(|G|)$

```
procedure Traverse(v:vertex);
begin
        v.mark := not v.mark;
        ... do something to v ...
        if v.index ≤ n
        then begin {v nonterminal}
                if v.mark ≠ v.low.mark then Traverse(v.low);
                if v.mark ≠ v.high.mark then Traverse(v.high);
        end;
end;
```

Figure 3.Implementation of Ordered Traversal

Reduction

- The reduction algorithm transforms an arbitrary function graph into a reduced graph denoting the same function
- For each vertex v it assigns a label $id(v)$ such that for any two vertices u and v, $id(u) = id(v)$ iff $f_u = f_v$
- By **working from the terminal vertices** up to the root, a procedure can label the vertices by the following inductive method
- Two terminal vertices should have 0 or 1
- If some vertex v is redundant, it will be labeled with same id recently labeled

A brief view of Implementation

- Vertices are collected into lists according to their indices by procedure $Traverse$
- For each vertex v, we create a key of the form $(value)$ for a terminal vertex or of the form $(lowid, highid)$ for a nonterminal vertex
- The remaining vertices are sorted according to their keys
- We work through this sorted list, assigning a given label to all vertices having the same key
- We also store a unique vertex's pointer to create reduced graph

```
function Reduce(v: vertex): vertex;
        var subgraph: array[1..|G|] of vertex;
        var vlist: array[1..n+1] of list;
begin
        Put each vertex u on list  vlist[u.index]
        nextid := 0;
        for i := n+1 downto 1 do
        begin
                Q := empty set;
                for each u in vlist[i] do
                        if u.index = n+1
                                then add <key,u> to Q where key = (u.value) {terminal vertex}
                                else if u.low.id = u.high.id
                                        then u.id := u.low.id {redundant vertex}
                                        else add <key,u> to Q where key = (u.low.id, u.high.id);
                Sort elements of Q by keys;
                oldkey := (–1;–1);       {unmatchable key}
                for each <key,u> in Q removed in order do
                        if key = oldkey
                                then u.id:= nextid;   {matches existing vertex}
                                else begin  {unique vertex}
                                        nextid := nextid + 1; u.id := nextid; subgraph[nextid] := u;
                                        u.low := subgraph[u.low.id]; u.high := subgraph[u.high.id];
                                        oldkey := key;
                                end;
        end;
        return(subgraph[v.id]);
end;
```

Figure 4.Implementation of *Reduce*

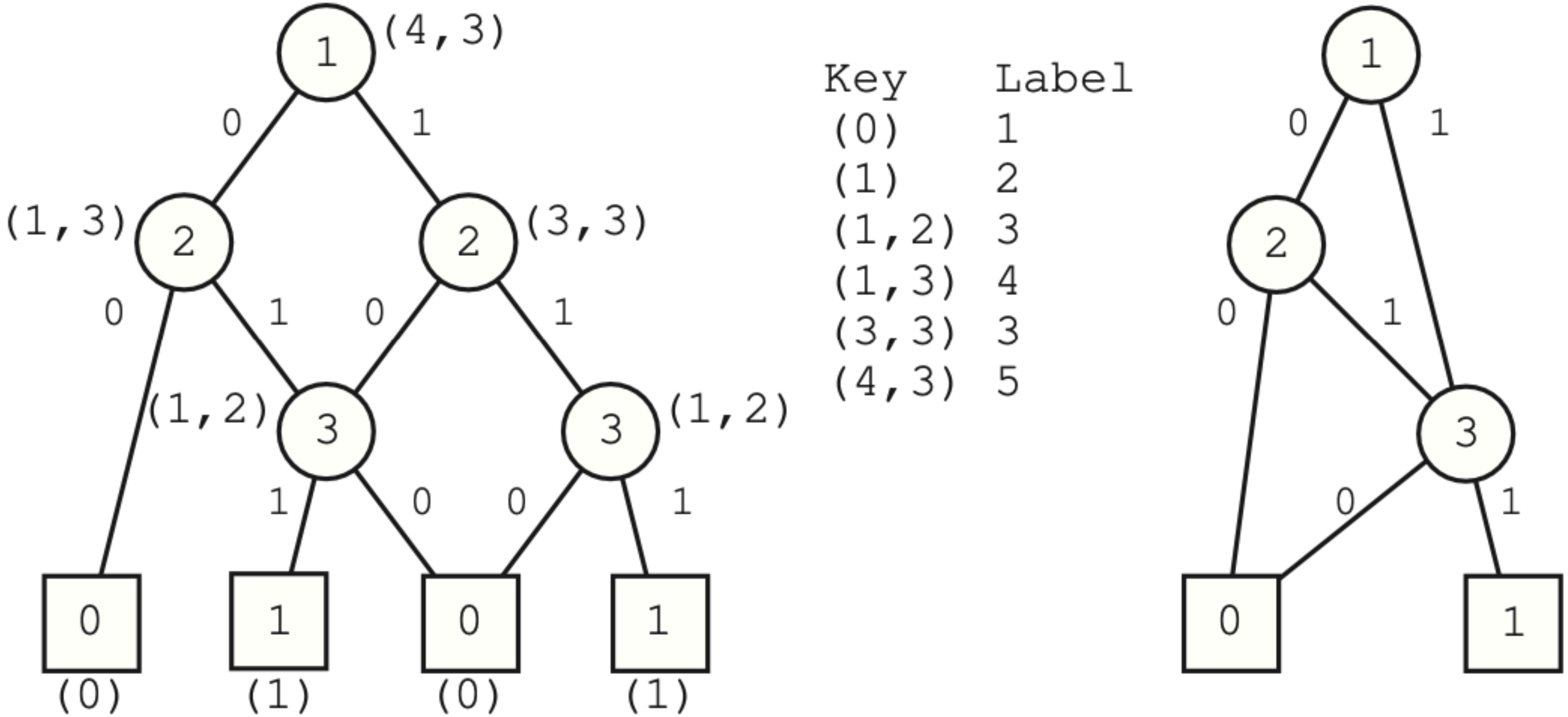


Figure 5.Reduction Algorithm Example

Apply

- It provides **ROBDD** of $f_1 < op > f_2$ ($< op >$ is a logical operator)
- It can do complement of function f by $f \oplus 1$
- This algorithm is based on Shannon expansion :
  $f_1 < op > f_2 = \bar{x} \cdot (f_1|_{x_i=0} < op > f_2|_{x_i=0}) \; + \; x_i \cdot (f_1|_{x_i=1} < op > f_2|_{x_i=1})$
- We have to handle several cases

A brief view of Implementation

- If both v and u are terminal vertices, than do $value(v) < op > value(u)$
- If both v and u are nonterminal vertices and their indices are same, do $apply(low(v), low(u))$ or $apply(high(v), high(u))$
- If v is nonterminal vertex and u is terminal vertex or it's index is greater than v, do $apply(low(v), u)$ or $apply(high(v), u)$
- But this algorithm's time complexity is $O(2^n)$

A brief view of Implementation(con't)

- We're gonna improve this algorithm by two refinements
- First, **we create a table** containing entries of the form (v_1, v_2, u) indicating that the result of applying the algorithm to subgraphs with roots v_1 and v_2 was a subgraph with root u
- Second, check if either v_1 or v_2 is a **controlling value** of boolean operation

```
function Apply(v1, v2: vertex; <op>: operator): vertex
    var T: array[1..|G1|, 1..|G2|] of vertex;

    {Recursive routine to implement Apply}
    function Apply-step(v1, v2: vertex): vertex;
    begin
        u := T[v1.id, v2.id];
        if u ≠ null then return(u); {have already evaluated}
        u := new vertex record; u.mark := false;
        T[v1.id, v2.id] := u; {add vertex to table}
        u.value := v1.value <op> v2.value;
        if u.value ≠ X
        then begin {create terminal vertex}
            u.index := n+1; u.low := null; u.high := null;
        end
        else begin {create nonterminal and evaluate further down}
            u.index := Min(v1.index, v2.index);
            if v1.index = u.index
                then begin vlow1 := v1.low; vhigh1 := v1.high end
                else begin vlow1 := v1; vhigh1 := v1 end;
            if v2.index = u.index
                then begin vlow2 := v2.low; vhigh2 := v2.high end
                else begin vlow2 := v2; vhigh2 := v2 end;
            u.low := Apply-step(vlow1, vlow2);
            u.high := Apply-step(vhigh1, vhigh2);
        end;
        return(u);
    end;

begin   {Main routine}
    Initialize all elements of T to null;
    u := Apply-step(v1, v2);
    return(Reduce(u));
end;
```

Figure 6.Implementation of *Apply*

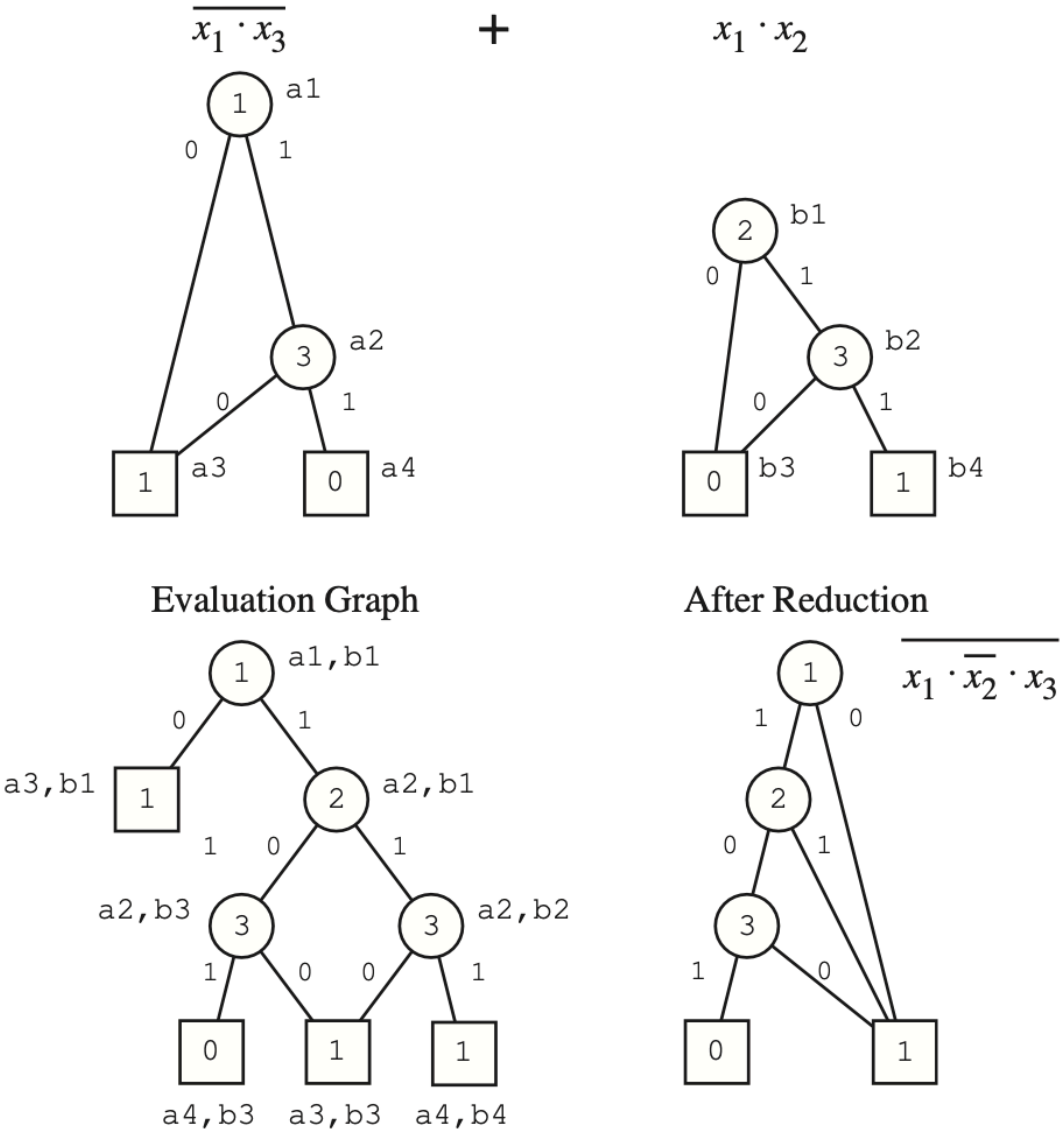


Figure 7.Example of *Apply*