Разбиение графа на подграфы
Выделение модулей в деревьях отказа
Бинарные диаграммы решений

Лень Ирина, 371 группа

СПбГУ, 2018

Определение

Разбиение графа на подграфы (разрезание графа) (англ. Graph partition)

— представление исходного графа $G = \langle A, V \rangle$ в виде множества подмножеств вершин $A = \{A_1, A_2, ..., A_n\}, A_i \subseteq V$ по определенным правилам. Обычно по условию задачи требуется: полнота $\bigcup_{i=1}^{n} A_i = A, A_i \neq \varnothing$ и ортогональность разбиения: $\forall i \neq j \ A_i \cap A_j = \varnothing$

Актуальность, применимость разбиения

- Задача раскраски графа.
- Задача определения числа и состава компонент связности графа.
- В задаче трассировки межсоединений печатных плат или микросхем необходимо разбиение исходной схемы на слои.
- Сегментация картинок (умные ножницы).
- Распараллеливание обработки данных.
- и т.д.

Проблемы разбиения

- Вариантов разбиения графа из n вершин на $n/2$ частей:

$$C_n^{n/2} = \frac{n!}{((n/2)!)^2} \approx 2^n \sqrt{2/(\pi n)}$$

- Поиск оптимального разбиения является NP-complete задачей

Методы разбиения графа

- Покоординатное разбиение
- Рекурсивный инерционный метод деления пополам
- Многоуровневое разделение
- Деление с учётом связности (по сути, поиск в ширину)
- Алгоритм Кернигана — Лина

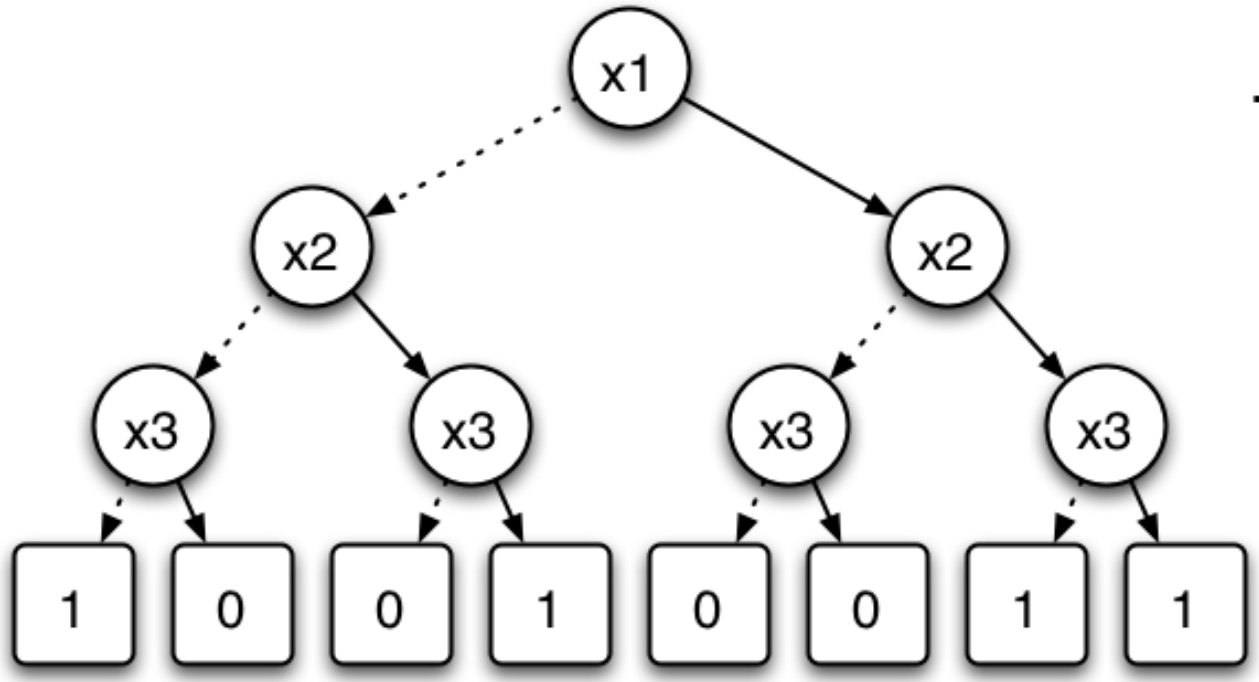
Программное Обеспечение

- METIS/ParMETIS (Kapyris)
- Chaco (Sandia)
- Scotch (INRIA)
- Jostle (now commercialized)
- Zoltan (Sandia)

Binary Decision Diagram (BDD)

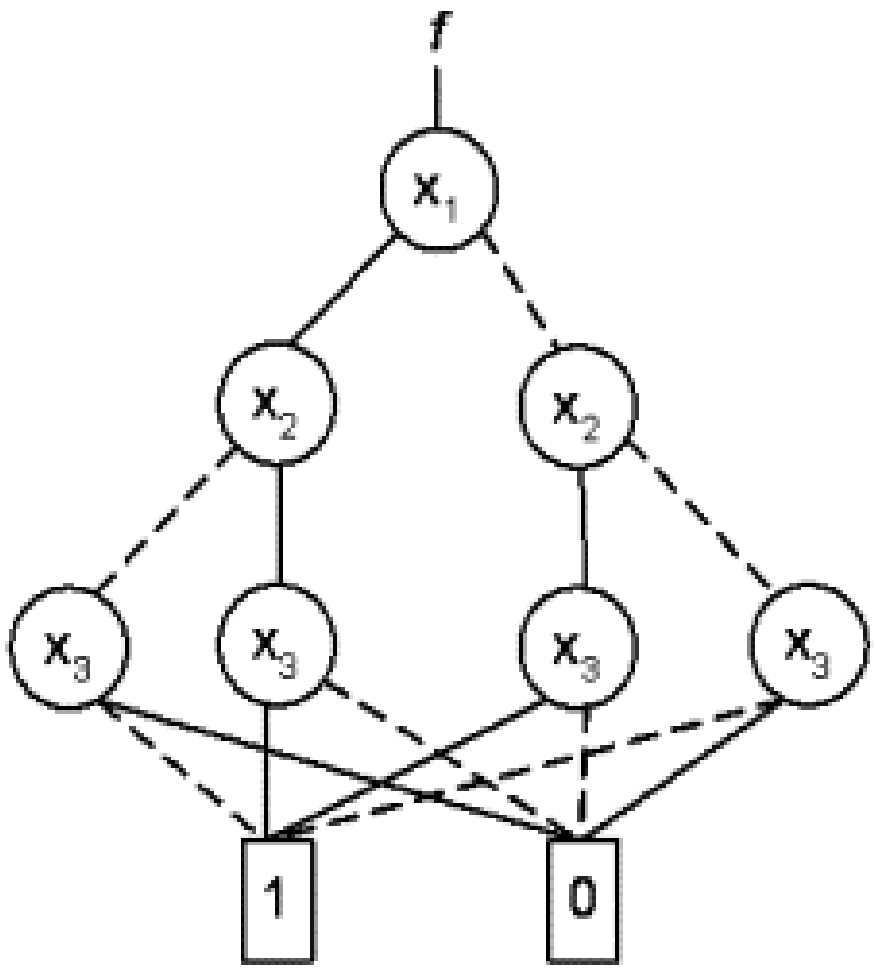
Бинарная диаграмма решений (БДР)

– форма представления булевой функции $f(x_1, x_2, ..., x_n)$ от n переменных в виде направленного ациклического графа, состоящего из внутренних узлов решений (помеченных x_i), каждый из которых имеет по два потомка, и двух терминальных узлов (помеченных 0 и 1), каждый из которых соответствует одному из двух значений булевой функции.

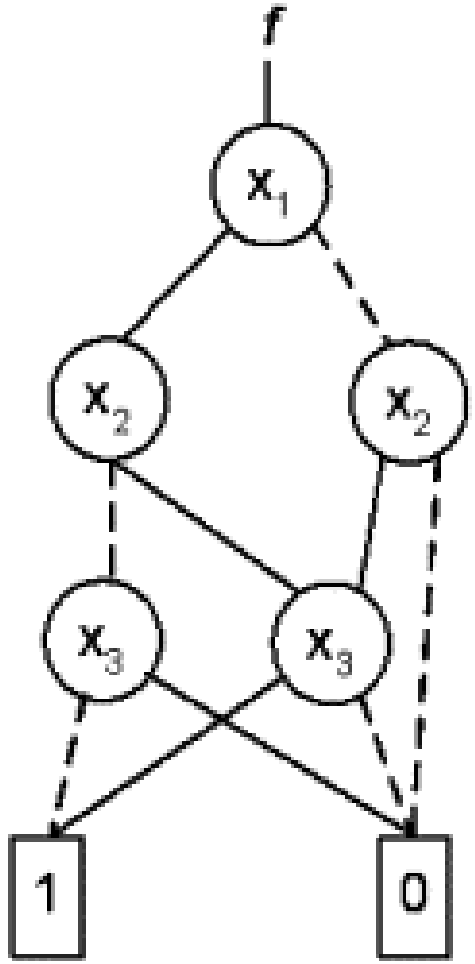


x1	x2	x3	f
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Виды BDD



(a) **OBDD**



(b) **ROBDD**

Актуальность, применимость BDD

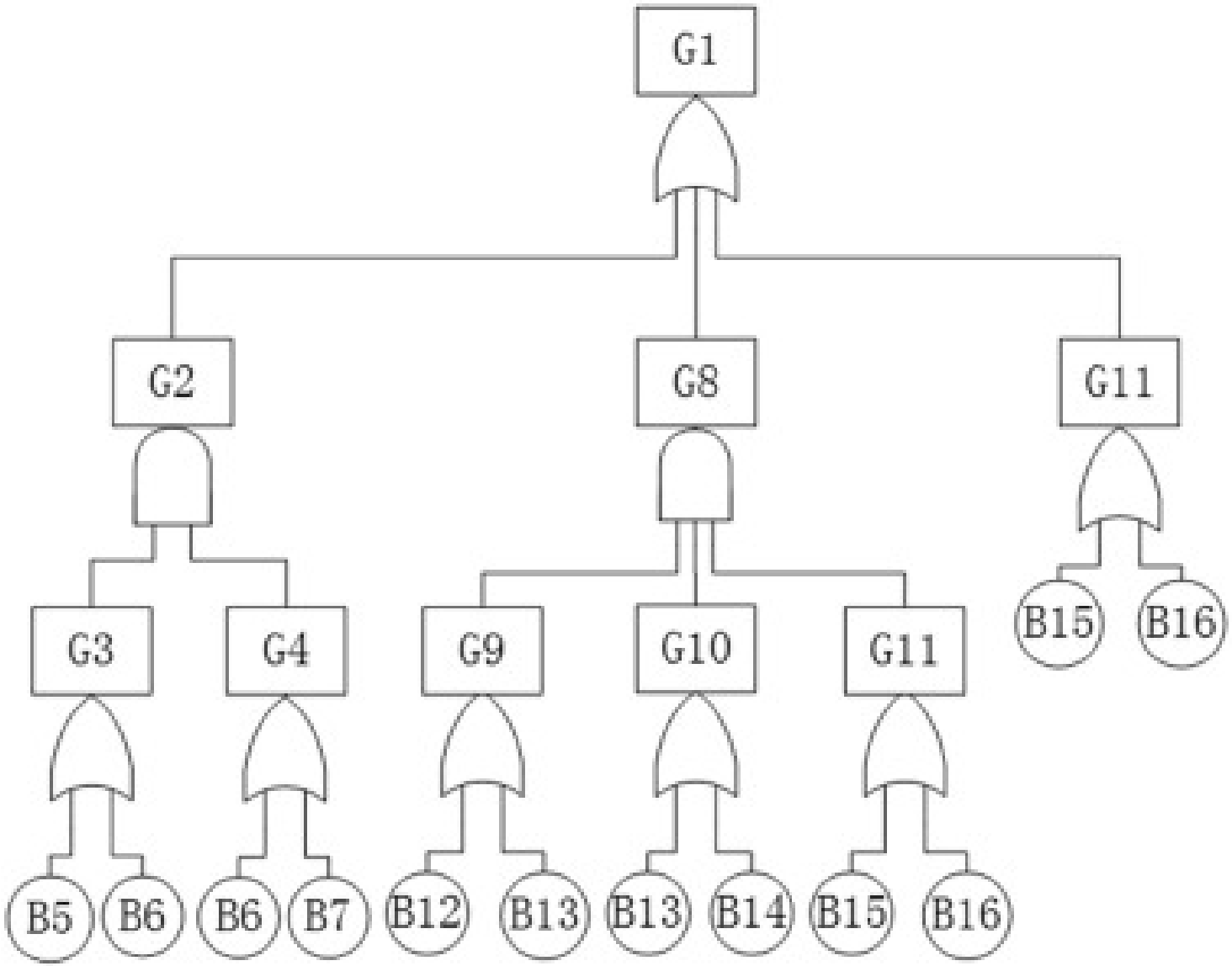
- Булевы базисы Грёбнера (практическую применимость в криптографии, моделирование квантовых вычислений и к задаче «Выполнимость» для конъюнктивной нормальной формы.)
- Упрощение логических схем.
- Поиск путей функционирования/отказа для системы.
- и т.д.

Parallel algorithm for finding modules of large-scale coherent fault trees

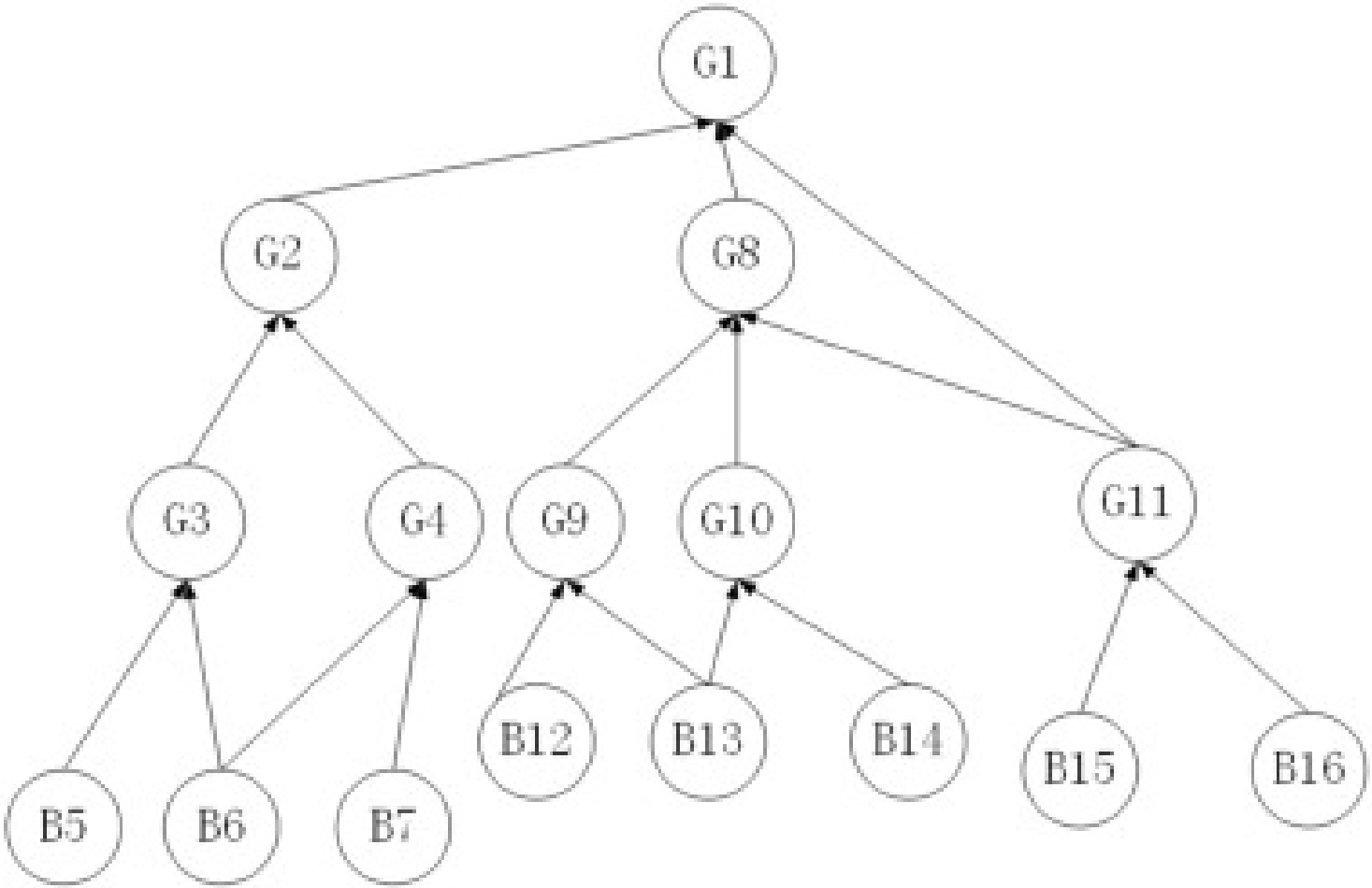
Z.F. Li, Y. Ren, L.L. Liu, Z.L. Wang

School of Reliability and Systems Engineering, Beihang University, Beijing, China

Дерево отказов(FT – fault tree)



Направленный ациклический граф (DAG – Directed Acyclic Graph)



Линейный алгоритм (LTA – Linear time algorithm)

The algorithm has 3 steps:

1. Initialize the counters; traverse the list of nodes.
2. Perform the first 'depth-first left-most' traversal of the graph to set counters; for leaves, dates #1 and #2 are identical.
3. Perform the second 'depth-first left-most' traversal of the graph in which it collects, for each internal event v, the minimum of the first dates and the maximum of the last dates of its children. v is a module if only and if:

   a) the collected minimum is greater than the first date of v.
   b) the collected maximum is less than the last date of v.

Параллельный алгоритм

V_{out} the output edge number of node in the directed acyclic graph

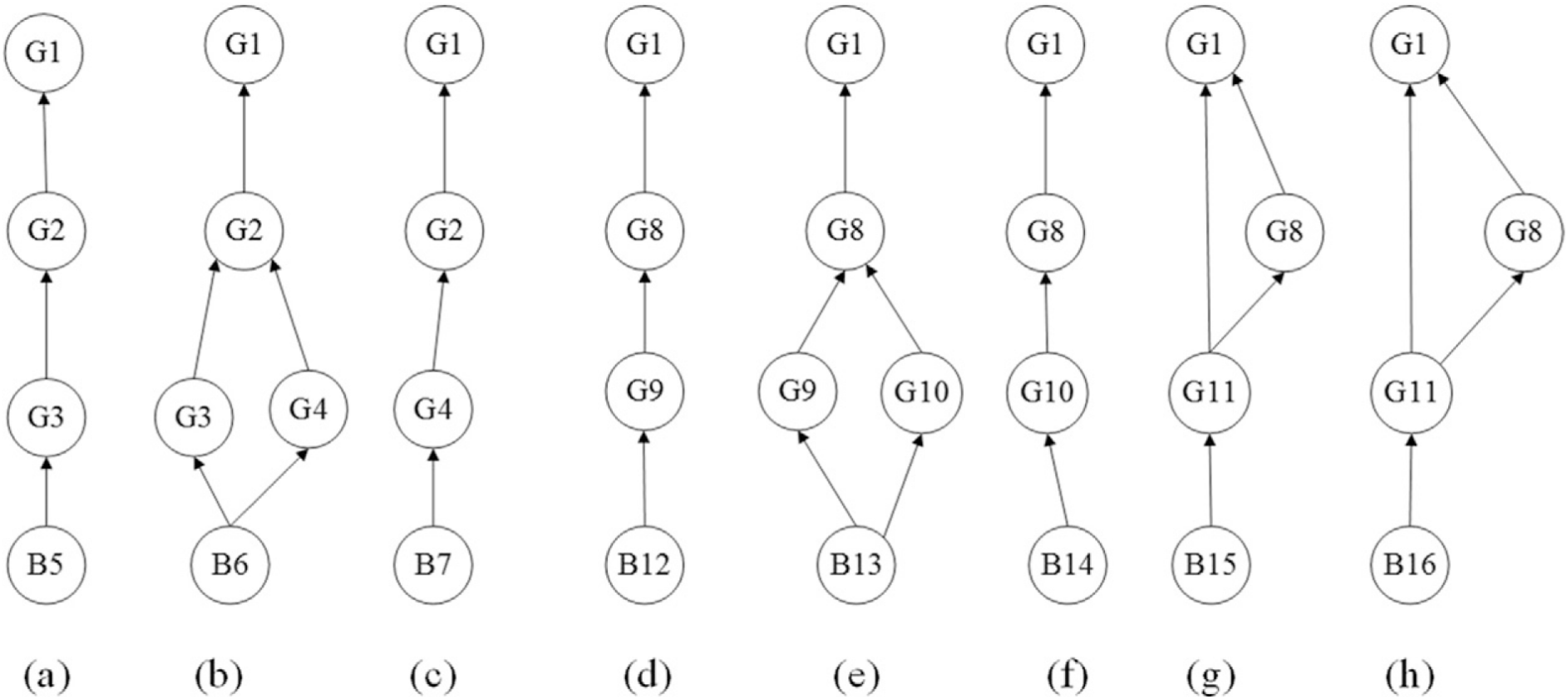
V_M it means whether the node is module top or not, if and only if $V_M \leq 1$, the node is module top.

We set the V_{out} when we construct the directed acyclic graph from fault tree. So we can conclude the following parallel algorithm basing on the above description.

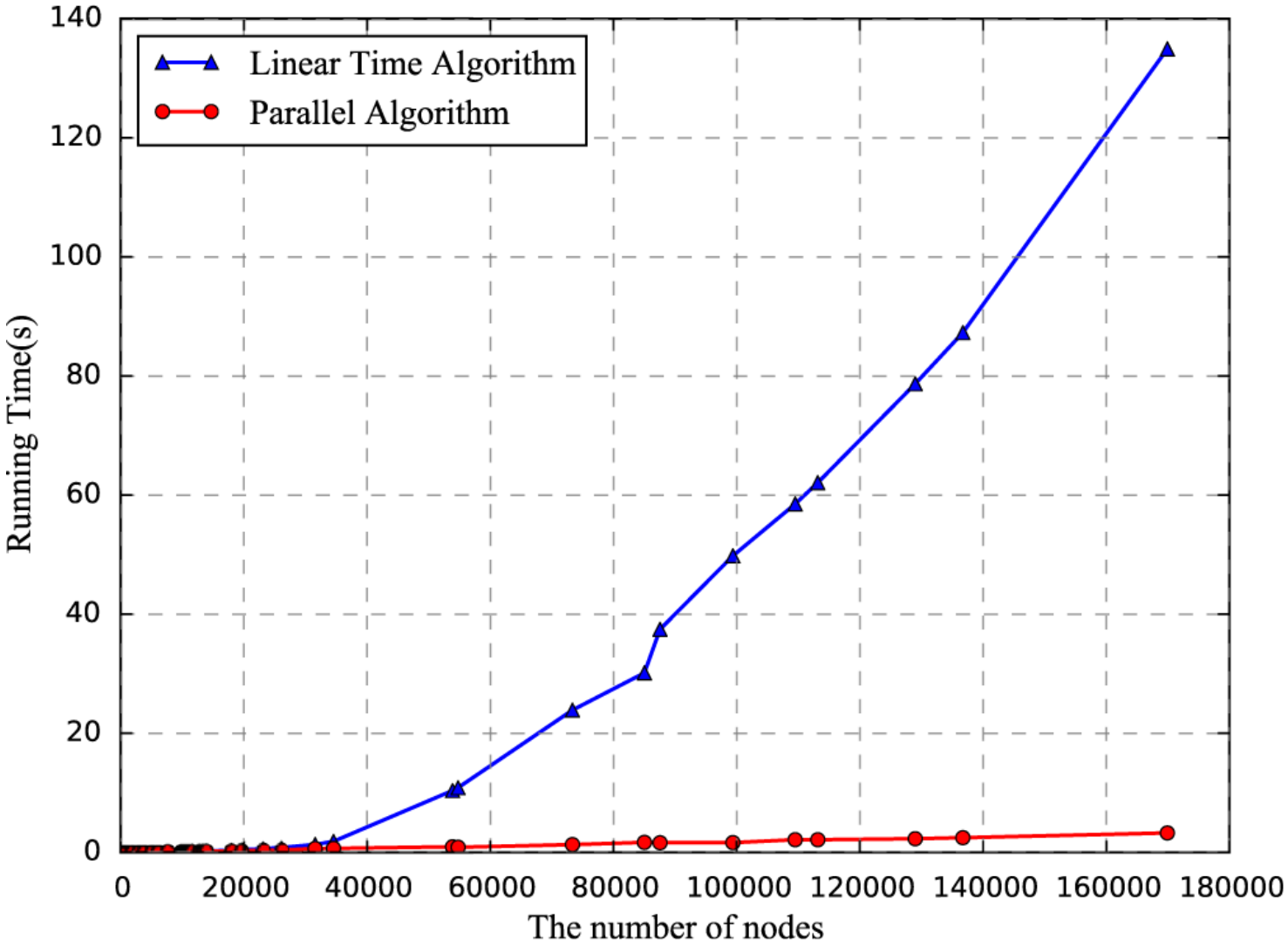
The algorithm consists of 4 steps:

1. Initial variable. For all leave node, $V_M = V_{out}$.
2. Perform the 'depth-first left-most' traversal from the leave nodes, and judge the internal node e whether it is visited or not. If it is not visited, updating its V_M value according to the formula $e_M = \max(s_M, s_{out})$. s is an input of node e. Or determine whether the path $\overline{se}$ is passed or not. If the path is passed, updating its V_M value according to the formula $e_M = \min(s_M, e_M - 1)$.
   Or updating the V_M according to formula $e_M = \min(s_M, e_M) - 1$.
3. Perform the procedure described in step 2 on every subgraph.
4. Union of the internal nodes of all subgraphs marked through above procedures, so we get all module tops. The rule to union of the nodes is described as follows. If any $V_M \leq 1$ for the same node in all subgraphs, we predict that the node is a module top. If the node does not follow the rule, it is not a module top.

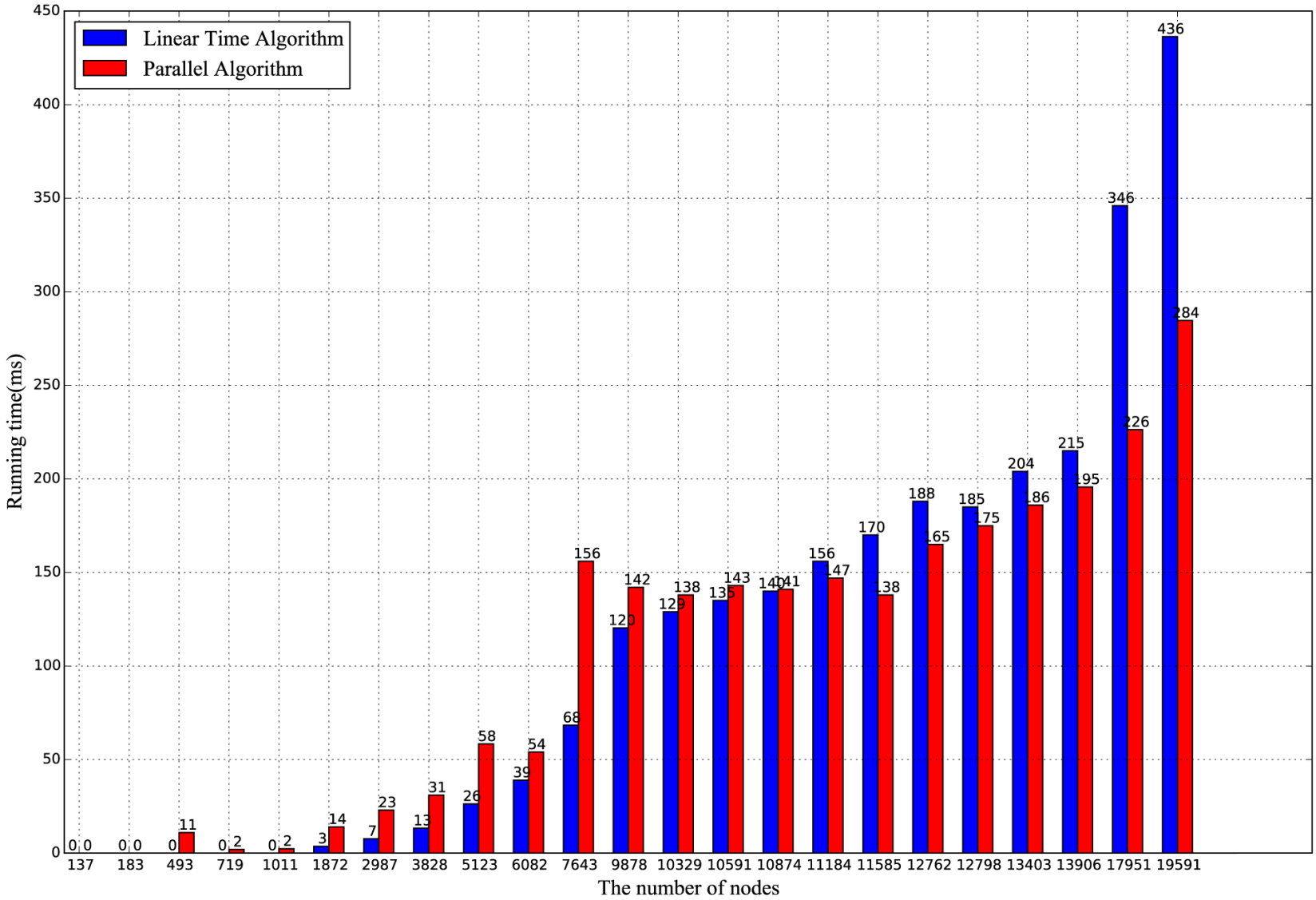
Подграфы для примера



Результаты


Linear Time Algorithm
Parallel Algorithm
Running Time(s)
The number of nodes
0
20
40
60
80
100
120
140
0
20000
40000
60000
80000
100000
120000
140000
160000
180000

Результаты


Linear Time Algorithm
Parallel Algorithm
Running time(ms)
The number of nodes
0
50
100
150
200
250
300
350
400
450
137 183 493 719 1011 1872 2987 3828 5123 6082 7643 9878 10329 10591 10874 11184 11585 12762 12798 13403 13906 17951 19591
0 0 0 0 0 2 0 2 3 14 7 23 13 31 26 58 39 54 68 156 120 142 129 138 135 143 140 141 156 147 170 138 188 165 185 175 204 186 215 195 346 226 436 284

BDD algorithms based on modularization for fault tree analysis

Yunli Deng, He Wang, Biao Guo

Fundamental Science on Nuclear Safety and Simulation Technology Laboratory, College of Nuclear Science and Technology, Harbin Engineering University, PR China

Преобразование FT в BDD

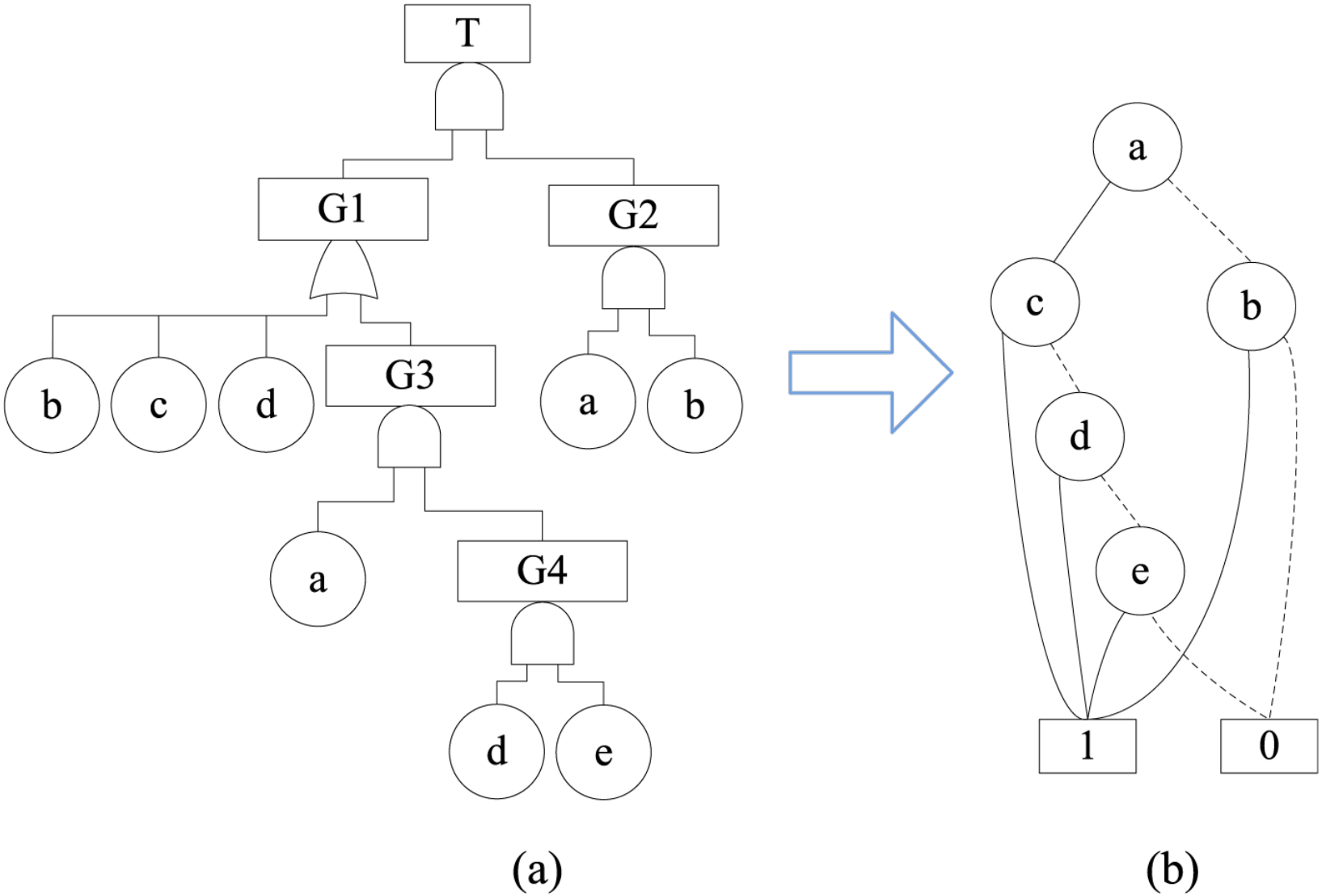


Fig. 2. (a) An example of a FT, (b) BDD graph of the FT after subsuming with the variable ordering a<b < c < d < e.

Рассматриваемый пример FT

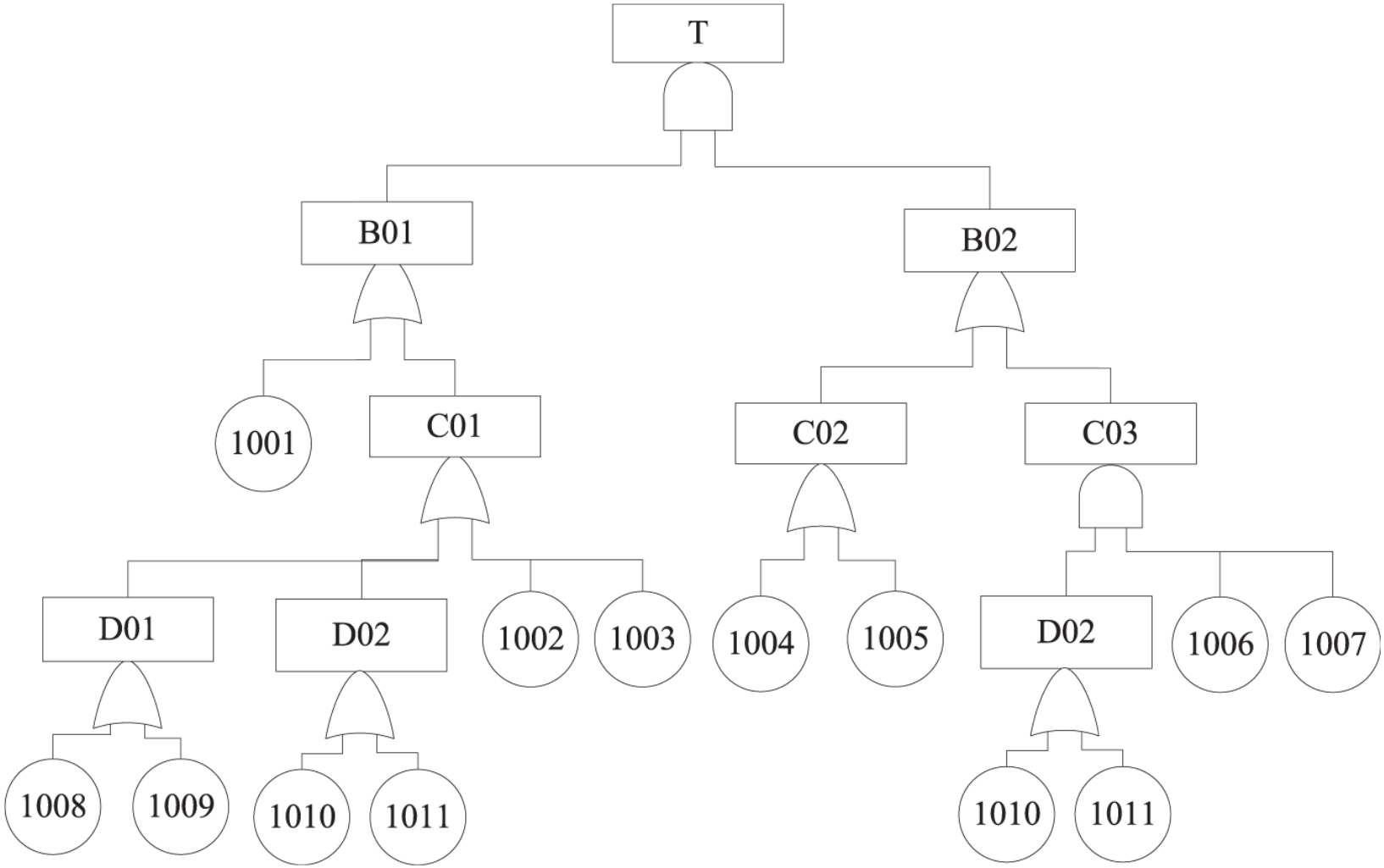


Fig. 3. An example of a fault tree.

Модули

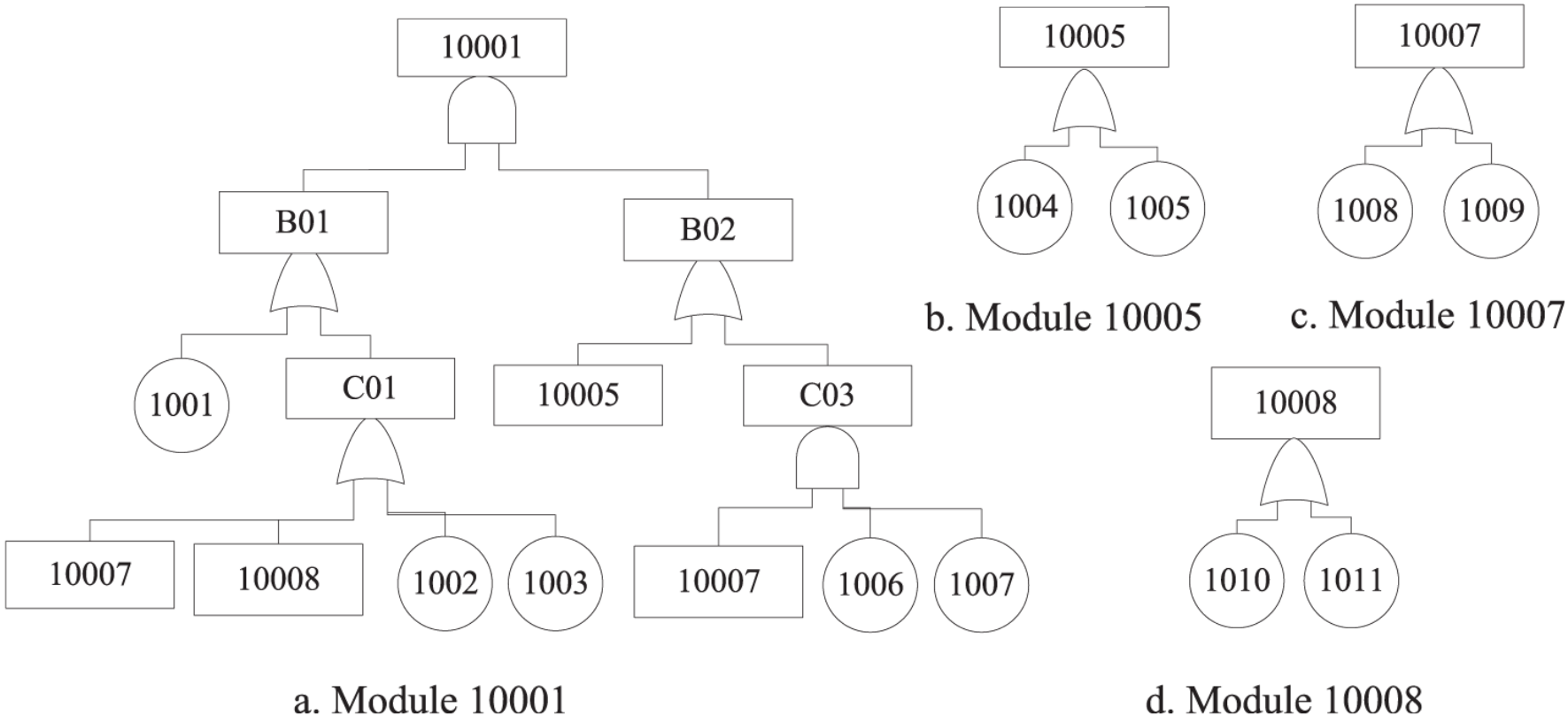
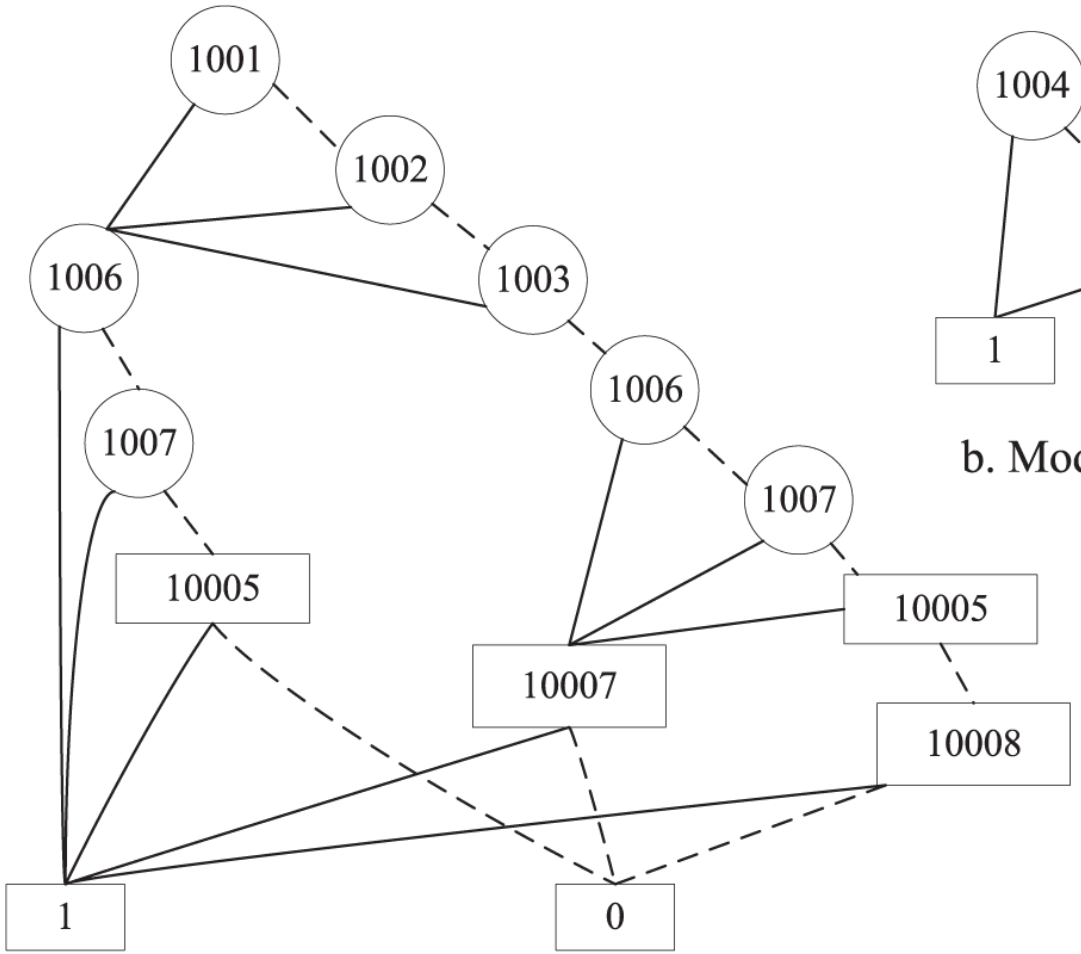
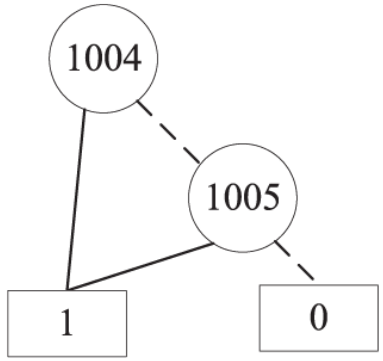


Fig. 4. Modules of the example FT.

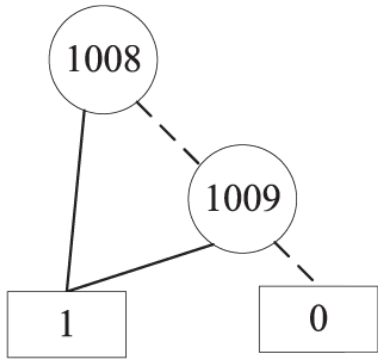
Модули, преобразованные в BDD



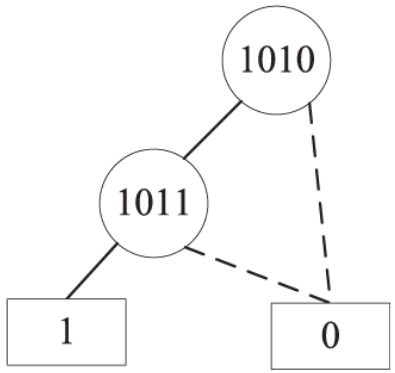
a. Module 10001



b. Module 10005



c. Module 10007



d. Module 10008

Fig. 5. BDDs of each module.

Возможные варианты присоединения модулей

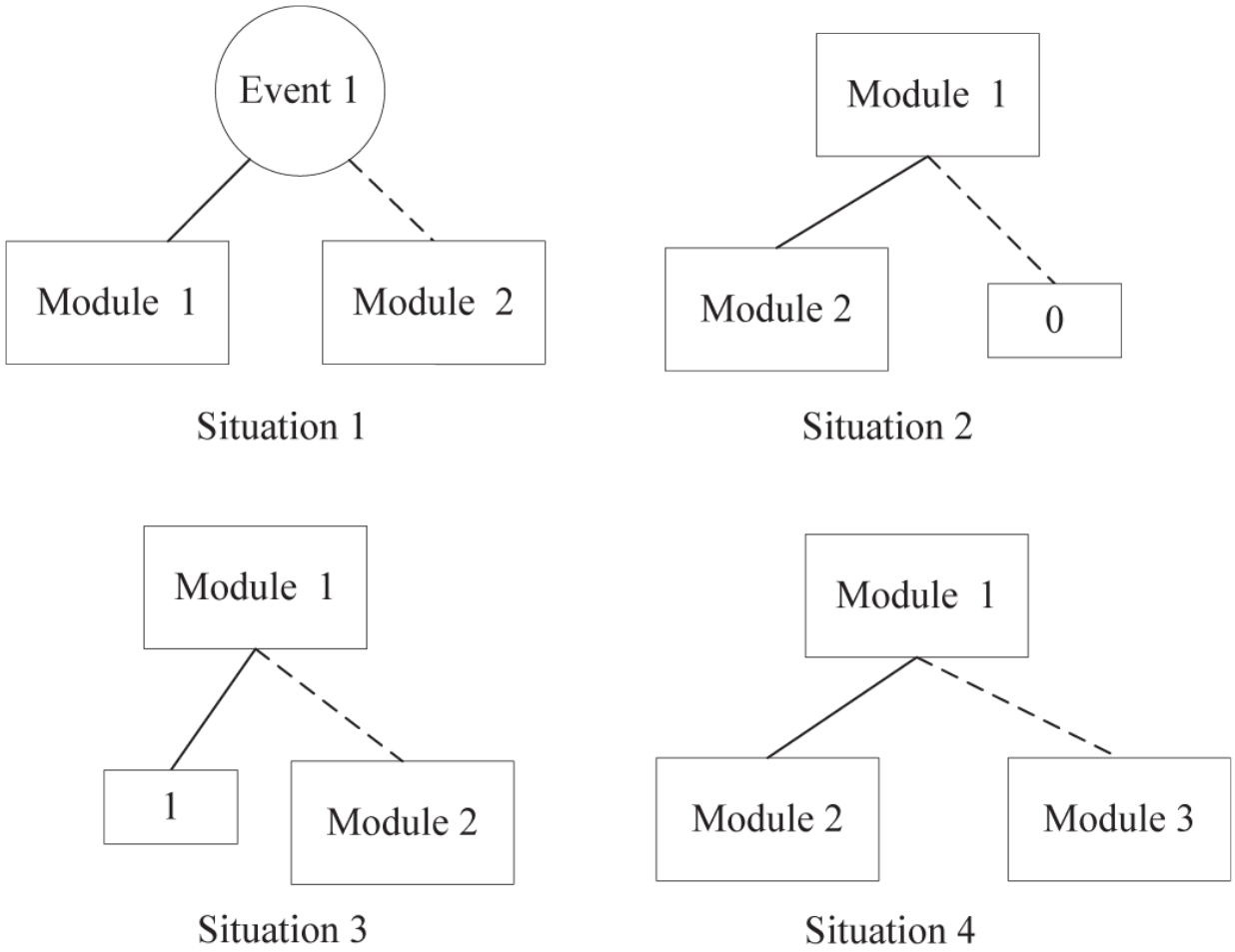


Fig. 7. Four situations in BDD structure with modules.

Алгоритм для случая 1

```
Traversing (T) /* T=ite(x, F, G, p_x)*/
                /* F=ite(y, F_1,G_1,p_y)*/
                /* G=ite(z,F_2,G_2,p_z)*/
    if(y>10000)/*it's a module*/
            MCS ← add(x), p = p_x * p
            Traversing(BDDs[y])/*BDDs is a hashtable to store BDDs
of modules with the key equaling to their number*/
    if(z>10000)
            MCS ← add(-x), p = (1 − p_x) * p
            Traversing(BDDs[z])
```

Fig. 8. Key to deal with situation 1.

Алгоритм для случая 2

```
Module2_Traversing (T, H) /* T=ite(x, F, G, p_x) , where x<10000*/
                          /* H=ite(h, F_h, G_h, p_h)*/
        MCS ← add(x), p = p_x * p
        if(F=1)
            Traversing(H)
            if(G=0)
            else
                MCS ← add(-x), p = (1 − p_x) * p
                Module2_Traversing(G, H)
        else
            Modul2_Traversing(F, H)
            if(G=0)
            else
                MCS ← add(-x), p = (1 − p_x) * p
                Module2_Traversing(G, H)
```

Fig. 9. Recursive function to deal with situation 2.

Алгоритм для случая 3

```
Module3_Traversing (T, H) /* T=ite(x, F, G, p_x) , where x<10000*/
                          /* H=ite(h, F_h, G_h, p_h)*/
        MCS ← add(x), p = p_x * p
        if(F=1)
              MCSs ← MCS, p_mcs ← p, p_top = p + p_top
              MCS=new , p = 1
              MCS ← add(-x), p = (1 − p_x) * p
             if(G=0)
                    Traversing(H)
             else
                    Module3_Traversing(G, H)
        else
              Modul3_Traversing(F, H)
              MCS ← add(-x), p = (1 − p_x) * p
              if(G=0)
                    Traversing(H)
              else
                    Module3_Traversing(G, H)
```

Fig. 10. Recursive function to deal with situation 3.

Алгоритм для случая 4

```
Module4_Traversing (T, H, K) /* T=ite(x, F, G, p_x) , where x<10000*/
                             /* H=ite(h, F_h, G_h, p_h)*/
                             /* K=ite(h, F_k, G_k, p_k)*/
        MCS ← add(x), p = p_x * p
        if(F=1)
            Traversing(H)
            MCS ← add(-x), p = (1 − p_x) * p
            if(G=0)
                Traversing(K)
            else
                Module3_Traversing(G, K)
        else
            Module2_Traversing(F, H)
            MCS ← add(-x), p = (1 − p_x) * p
            if(G=0)
                Module3_Traversing(F, K)
            else
                Module4_Traversing(F, G, H)
```

Fig. 11. Recursive function to deal with situation 4.

Результаты

Table 4
FT test results of BDD and BDD based on modularization.

FT	Size	BDD based on modularization			BDD	
		Modules	Solutions	Time(s)	Solutions	Time(s)
Das9201	160	32	14217	0.97	14217	>300
Das9204	83	13	16704	0.10	16704	1.05
Chinese	61	1	392	0.03	392	0.03
Isp9604	338	76	746574	3.82	–	–
Isp9606	130	17	1776	0.27	1776	106
Isp9607	138	14	150436	0.94	150436	>300
edf9201	314	23	579720	10.6	–	–

Список литературы

- Z.F. Li, Y. Ren, L.L. Liu, Z.L. Wang, Parallel algorithm for finding modules of large-scale coherent fault trees
- Y. Dutuit, A. Rauzy, A linear-time algorithm to find modules in fault trees
- Yunli Deng, He Wang, Biao Guo, BDD algorithms based on modularization for fault tree analysis
- http://www.cs.cornell.edu/ bindel/class/cs5220-f11/slides/lec19.pdf