Practice 12 (2016/12/20)
Task scheduling & Logic simulator

1. **Directed graph → Leveled directed graph by causality**
   A. A graph consists of vertices and edges. A graph is a directed graph if each edge has specific direction. The edge direction implies the causality of something represented in the graph. Consider the causality of a directed graph, we can re-arrange the vertices of a directed graph in a causality order.
   B. A graph is acyclic if it has no cycle. A directed graph is acyclic if it has no directed cycle. If we map a logic gate of a combinational circuit as a vertex in a directed graph and map a connection between two logic gates as a directed edge connecting two associated vertices of these two logic gates, the graph is acyclic.
2. **Problem 1 (75 pts)**

   Task scheduling is to determine a sequence to perform a set of tasks. The input to the task scheduling problem is a set of tasks with some constraints that define the execution ordering for two tasks. For instance, task *A* cannot be performed before task *B* is completed since they have specific relation like that task *B* produces something needed by task *A*. Read all constraints and construct a constraint graph where each vertex represents a task and a directed edge connects vertex *a* to vertex *b* if task *B* (vertex *a*) can be performed only after task *A* is completed. According to the constraint graph, cluster in a group all tasks that can be performed simultaneously. In Fig. 1, a directed edge connect vertex 2 to vertex 6, which implies task 6 must be performed after the completion of task 2. Groups are executed sequentially in increasing order of group number. All tasks in a group are ordered in increasing order of task number and can be performed simultaneously. Notably the constraint graph is acyclic, i.e., there is no cycle in the graph.

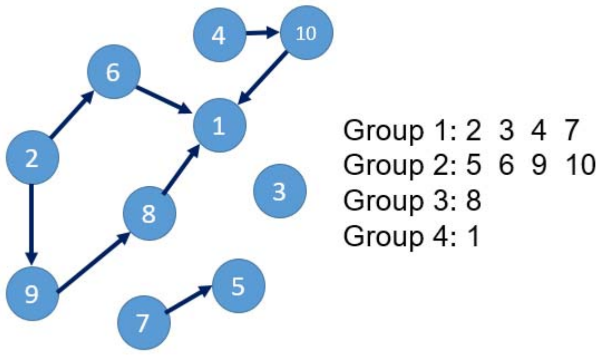


Fig. 1

Input Format:

Each line contains two tasks that are represented in integers and separated by a blank and represents a constraint between two tasks, which means the later task must be performed after the completion of the former one.

Input example 1: the input file of Fig. 1 is as follows.

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

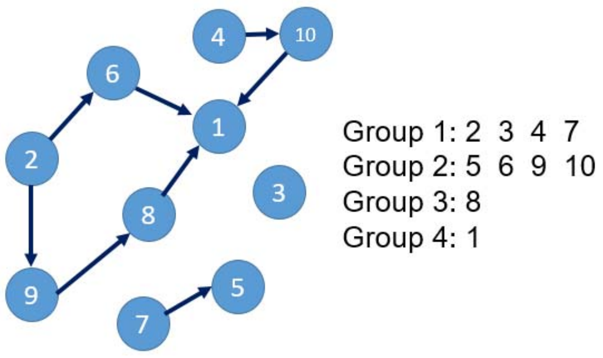
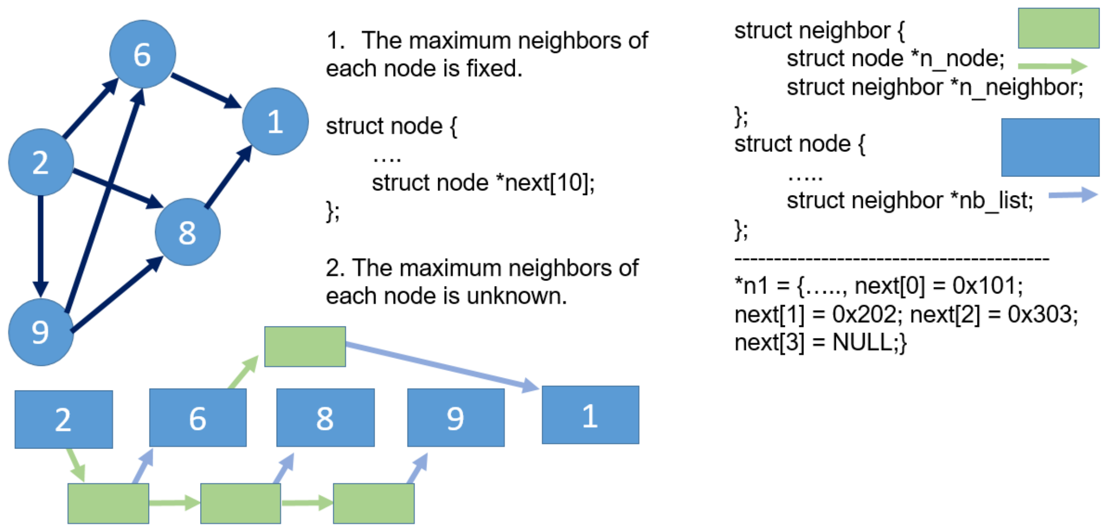


Fig. 2

Output format: write the output into a file named as "task.out". For each group, the group number and the number of tasks in this group are printed at a line and all task numbers in the group are printed at next line.

Output example: The output file of Fig. 1 is as follows

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


1. The maximum neighbors of each node is fixed.

```
struct node {
      ….
      struct node *next[10];
};
```

2. The maximum neighbors of each node is unknown.

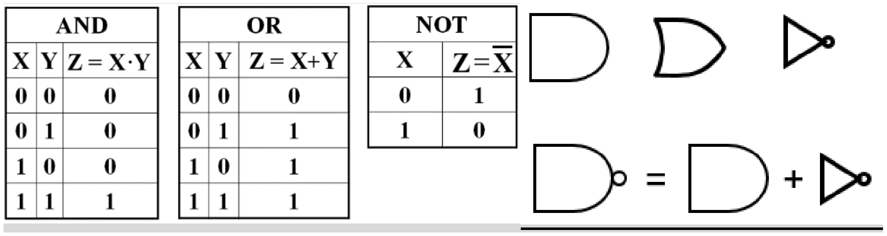
```
struct neighbor {
      struct node *n_node;
      struct neighbor *n_neighbor;
};
struct node {
      …..
      struct neighbor *nb_list;
};
----------------------------------------
*n1 = {….., next[0] = 0x101;
next[1] = 0x202; next[2] = 0x303;
next[3] = NULL;}
```

Fig. 3

Main features:

A. Read all constraints from a constraint file. For each task, construct a vertex to represent it. Use *next* pointer to link all task vertices to form a linked list. As you insert a new task into the linked list, maintain the list in an increasing order of task number. You can reference the codes in the textbook. (due tonight)

B. Schedule tasks and cluster tasks in groups and write the resultant groups in the output file “task.out”. (due in 12/27)



AND		
X	Y	Z = X·Y
0	0	0
0	1	0
1	0	0
1	1	1

OR		
X	Y	Z = X+Y
0	0	0
0	1	1
1	0	1
1	1	1

NOT	
X	$Z=\overline{X}$
0	1
1	0

Fig. 4

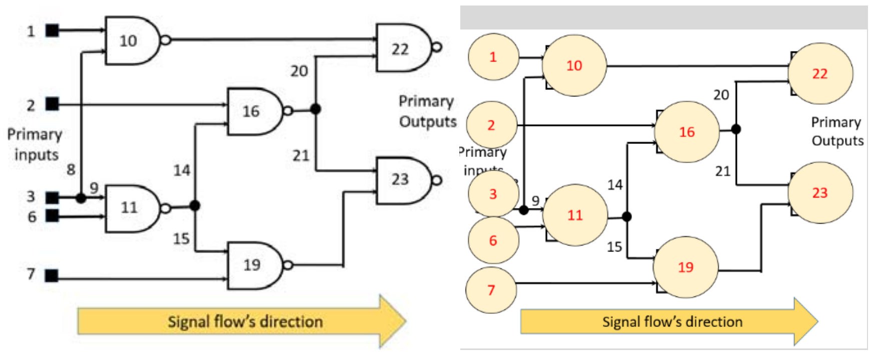


Fig. 5

3

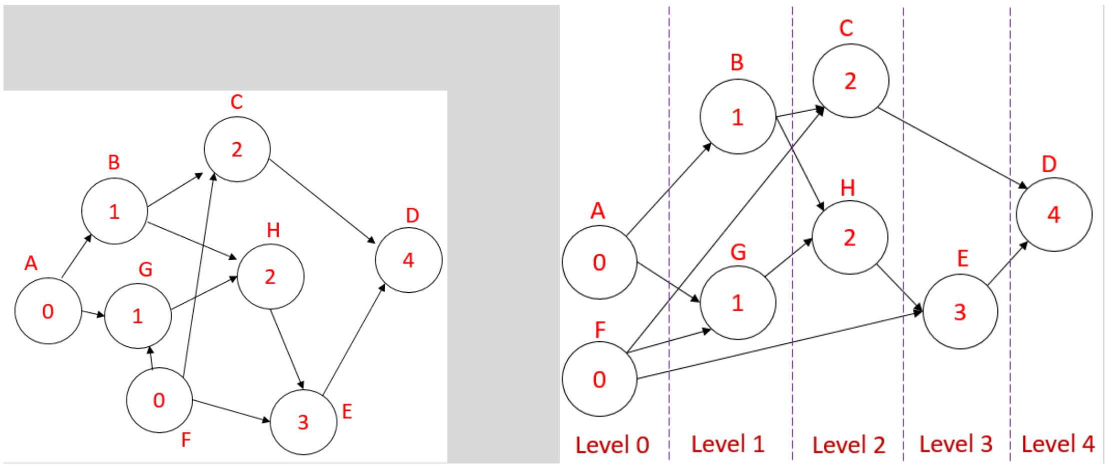
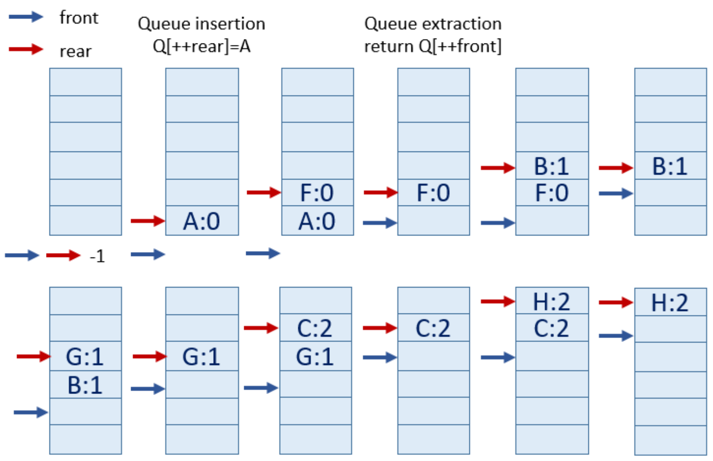


Fig. 6

```
1.  Put all vertices of 0 in-degree into a queue
2.  while (queue is not empty)          {
3.      extract one vertex v from the queue;
4.      for each vertex u pointed by v   {
5.          increase visited number of u by 1;
6.          if (visited number == fanin number)  {
7.              level number = level(v) + 1;
8.              put vertex u into queue;
9.          }  // end if
10.     }  // end for
11. } // end while
```

Fig. 7 Leveling algorithm and queue operation



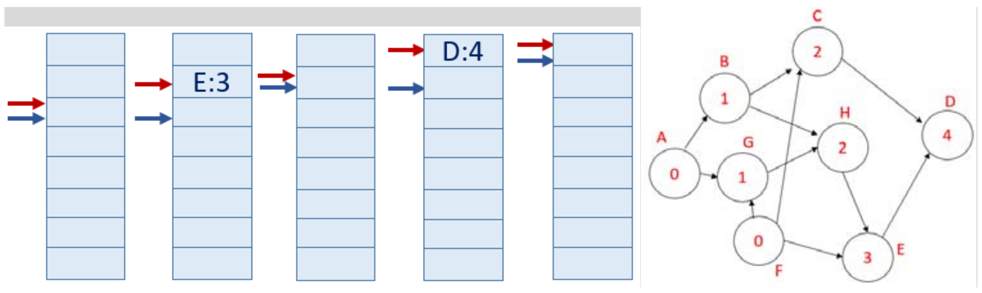


Fig. 8

3. Problem 2 (100 pts)

Logic simulator is a very important simulation technique to verify the correctness of a circuit design. Designers often apply logic simulator to his designs to obtain their output response based on a given set of input patterns. After that the output responses are compared with a golden set of outputs to check if the designs are correct. The problem is pretty simple but huge problem instance size complicates the implementation of methods to realize logic simulation. Since to apply all combinations of input patterns to the circuit under test is impossible, we are usually given a set of input patterns and then apply all input patterns to logic simulator. The way to put an input pattern into an integer is simple for implementation but inefficient for the performance of logic simulator. Thus generally several input patterns are packed into an integer, such as 32 patterns for a machine of 32-bit word length. With this scheme, we can perform logic simulation for 32 input patterns at a time.

Logic gate operations: In this project, there are 7 types of logic gates that are listed below. For each table, the rightmost column is the output value of each logic gate.

AND		
A	B	A&B
0	0	0
0	1	0
1	0	0
1	1	1

NAND		
A	B	~(A&B)
0	0	1
0	1	1
1	0	1
1	1	0

XOR		
A	B	A^B
0	0	0
0	1	1
1	0	1
1	1	0

OR		
A	B	A|B
0	0	0
0	1	1
1	0	1
1	1	1

NOR		
A	B	~(A|B)
0	0	1
0	1	0
1	0	0
1	1	0

NOT	
A	~A
0	1
1	0

BUF	
A	A
0	0
1	1

Fig. 9

```
Bitwise operations

12 & 7 = 4,  12 | 7 = 15
1  1  0  0          1  1  0  0
0  1  1  1  AND     0  1  1  1  OR
---------------     ---------------
0  1  0  0   NOT    1  1  1  1  NOT
--------------      --------------
1  0  1  1 (NAND)  0  0  0  0 (NOR)
```

Fig. 10

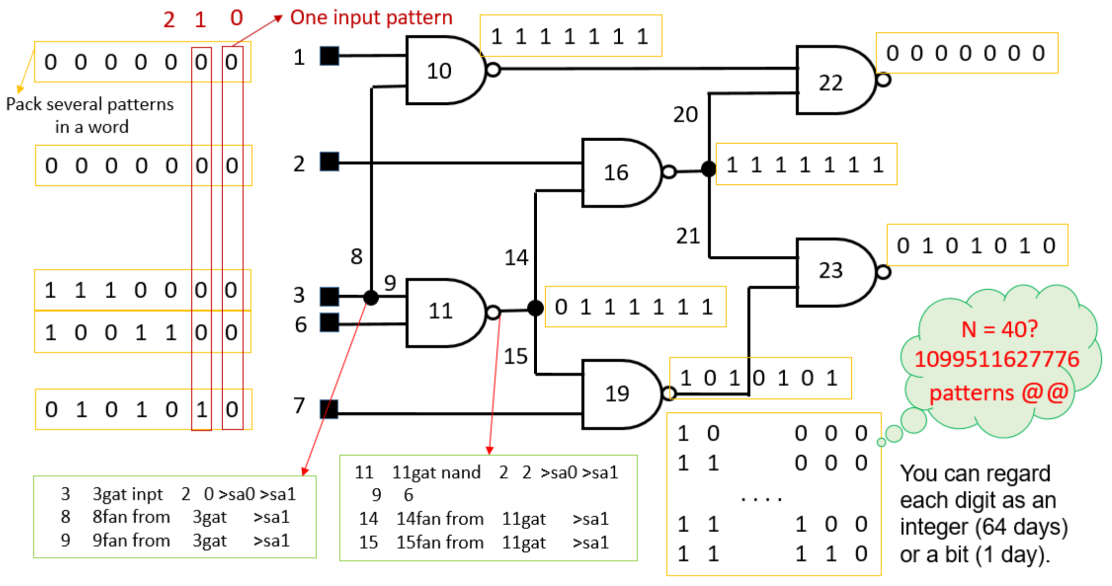


Fig. 11

Input Format:

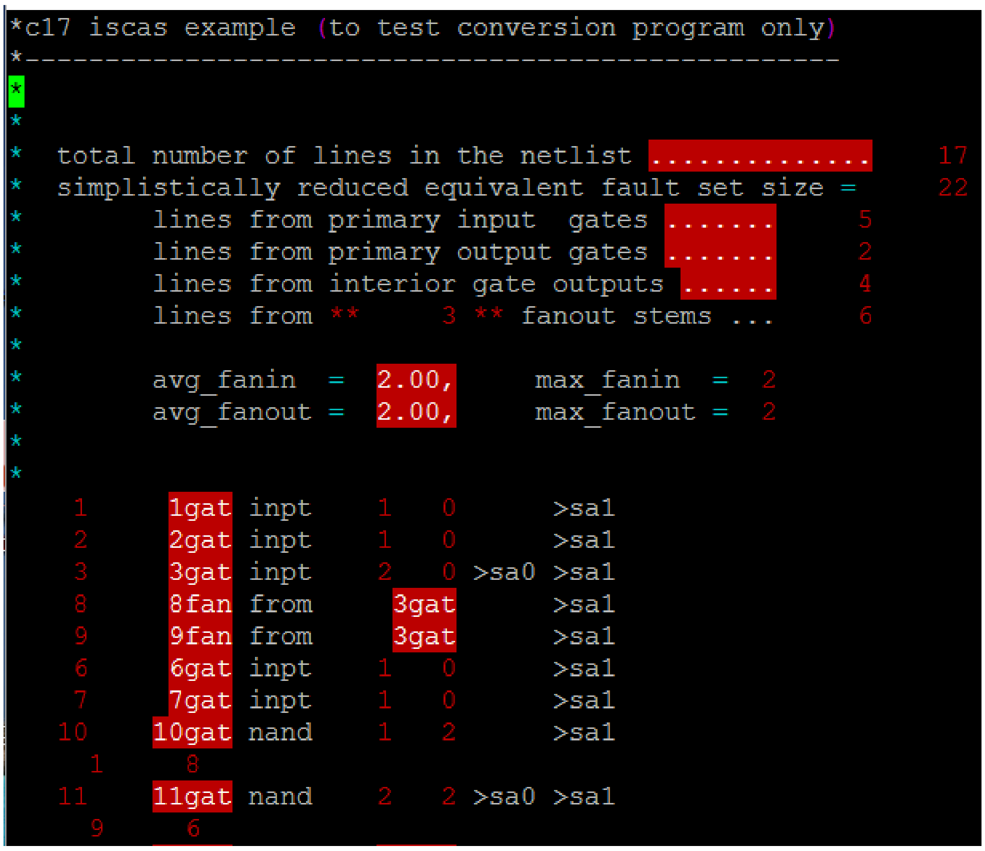
Signal_no   Signal_name   gate_type   fanout_no   fanin_no   other

If fanout_no is larger than 1, the following lines describe fanout

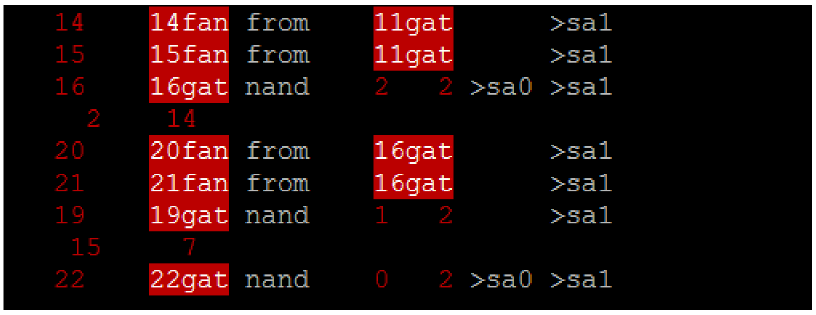
Circuit input example:

```
    3       3gat inpt      2    0   >sa0 >sa1
    8       8fan from       3gat        >sa1
    9       9fan from       3gat        >sa1
   10      10gat nand      1    2        >sa1
     1       8

   11      11gat nand      2    2 >sa0 >sa1
     9       6
   14      14fan from      11gat         >sa1
   15      15fan from      11gat         >sa1
   22      22gat nand      0    2 >sa0 >sa1
      10      20
```



```
*c17 iscas example (to test conversion program only)
*---------------------------------------------------
*
*
*  total number of lines in the netlist ..............     17
*  simplistically reduced equivalent fault set size =      22
*        lines from primary input  gates .......      5
*        lines from primary output gates .......      2
*        lines from interior gate outputs ......      4
*        lines from **     3 ** fanout stems ...      6
*
*        avg_fanin  =  2.00,     max_fanin  =  2
*        avg_fanout =  2.00,     max_fanout =  2
*
*
    1     1gat inpt    1   0        >sa1
    2     2gat inpt    1   0        >sa1
    3     3gat inpt    2   0   >sa0 >sa1
    8     8fan from     3gat        >sa1
    9     9fan from     3gat        >sa1
    6     6gat inpt    1   0        >sa1
    7     7gat inpt    1   0        >sa1
   10    10gat nand    1   2        >sa1
     1     8
   11    11gat nand    2   2   >sa0 >sa1
     9     6
```



```
   14    14fan from    11gat        >sa1
   15    15fan from    11gat        >sa1
   16    16gat nand    2   2   >sa0 >sa1
     2    14
   20    20fan from    16gat        >sa1
   21    21fan from    16gat        >sa1
   19    19gat nand    1   2        >sa1
    15     7
   22    22gat nand    0   2   >sa0 >sa1
```

Input pattern file: For the circuit whose input number is below 20, the input pattern file will not be given. You have to apply every pattern to logic simulation and print the output responses in increasing order of pattern number. For the circuit whose input number is above 20, an input pattern file with 1 million input patterns will be offered

Pattern 1

Pattern 2

…

Each pattern is an unsigned long integer and is printed at a line.

Input pattern example:

8474831

299393213

…

Means binary patterns for 32 input signals:
00000000100000010101000011001111
00010001110110000110000010111101

Output format

In_no1 In_no2 … In_noN || Out_no1 Out_no2 … Out_noK

The first line prints the input signal and output signal name. The input and output signals are printed in decreasing order of their signal number. Two adjacent signals are separated by one space character.

Output example:

```
7 6 3 2 1 || 23 22
0 0 0 0 0 || 0 0
0 0 0 0 1 || 0 0
0 0 0 1 0 || 1 1
```

If the input number is more than 20, you have to read an input pattern file. The output pattern order is the same as that in the input pattern file.

Main features:

(a) **Given a circuit file and an input pattern file (for the circuit with more than 20 inputs). Construct leveled directed graph and the gates of the same level are stored in a level list.**

(b) **Pack every 32 or 64 input patterns in a word and perform 32 or 64 input-pattern logic simulation on the graph at a time (depending on whether your computer is a 32-bit or 64-bit computer). Write input pattern and output responses to output file cxx.out. For instance, you have to generate an output file names as c17.out for the circuit c17.isc (due in 1/3).**