IIT Bombay

Technology Mapping

Project Documentation

Project Members:

Abhishek Bhowmick – 09007018
Sumit Gupta – 09D07006

Project Guide:

Prof. Sachin Patkar
Dilawar Singh

Project Abstract:

For the VLSI CAD project, we worked on technology mapping with a cost function such as area or power consumption as the optimization parameter. The main approach to be followed is similar to mapping for area and delay optimization, that is, to model the circuit in a canonical form (the subject DAG) and then to optimally cover the graph using basis function representation of gates from our technology library. For our implementation, the cost function is just a numeric value which can represent the area or power cost. An important difference from the area-delay optimization case is that for power minimization, the cost function is dependent on the Boolean function to be implemented (related through switching probability at the output of a gate).

For the purpose of the project, we will make the following assumptions:

1. Nodes don't have multiple fanout; hence the circuits are always trees.
2. The maximum number of inputs in a circuit is 10. Hence, the maximum number of leaves in the corresponding DAG is 10.
3. The maximum number of vertices in a graph is 100.
4. Since many subject DAGs are possible for a given circuit, we will arbitrarily choose one DAG.

Methodology:

i. The Boolean function is first cast into a subject DAG.
ii. This graph is then partitioned into a forest of trees (absent in our current implementation)
iii. Exact algorithms for optimal covering of the trees are run (dynamic programming – bottom-up approach).
iv. The covering with the minimum cost function is chosen.

Explanation

Technology mapping is generally used after technology independent optimization and logic decomposition. The role of technology mapping in this course is to assign logic functions to gates from custom library; and thus optimize of Area, delay, power etc. Thus we also know it as "library binding".

Terms &Definitions:

Base function set: is a set of gates which is universal and is used to implement the gates in the technology library (AND, OR and NOT in our case).
Technology library: a collection of well-designed gates having accurate cost estimates. Contains only AND, OR and NOT gates in our library, more can be added easily.
Subject graph: is the graph representation of a logic function using only gates from a given base function set. (i.e., the nodes are restricted to base functions).

Code Snippets with Explanation

Description: This program maps a technology independent logic optimized circuit onto a given technology and optimizes the implementation for minimum cost. This code takes a graph file as graphml input and uses a specified technology library to implement the given function.

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation=
"http://graphml.graphdrawing.org/xmlns http://graphml.graph drawing.org/xmlns/1.0/graphml.xsd">
<key id="key0" for="node" attr.name="name" attr.type="string" />
<graph id="G" edgedefault="directed" parse.nodeids="canonical" parse.edgeids="canonical"
parse.order="nodesfirst">
<node id="n0">
<data key="key0">&amp;amp; </data>
</node>
<node id="n1">
<data key="key0">i</data>
</node>
<node id="n2">
<data key="key0">i</data>
</node>
<edge id="e1" source="n0" target="n1">
</edge>
```

```
<edge id="e2" source="n0" target="n2">
</edge>
</graph>
</graphml>
```

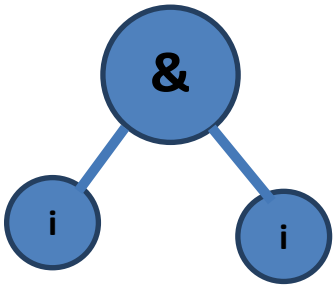


Figure 1

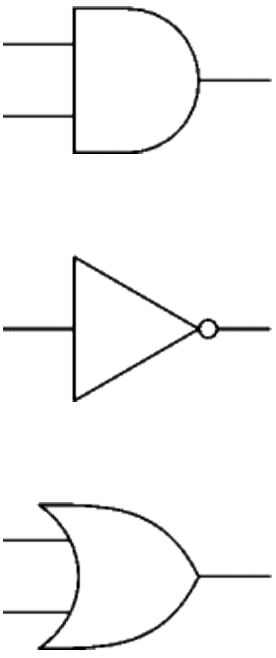
```
// all lines have to end with a tab and all words in a line are separated by tab

Library_name    65nm_technology_node
Nbr_gates       3

GATE
Gate_name       AND
Nbrpatterns     1
Pattern_files   AND_1.graphml
Gate_cost       2

GATE
Gate_name       NOT
Nbrpatterns     1
Pattern_files   NOT_1.graphml
Gate_cost       1

GATE
Gate_name       OR
Nbrpatterns     1
Pattern_files   OR_1.graphml
Gate_cost       3
```



This is the library file which contains information about all the gates in the library, their graphml files and cost. Nbrpatterns indicates the number of pattern graphs possible for that particular gate, we have to read all of them.

Techmapping.cpp file - Code Snippet

Find optimal Area Covering

1. The final output is to be given as an optimum 'covering' of the output node, which is a 'list' of gates and the total cost
2. A gate consists of a type, the input nodes, output node, covered nodes and cost
3. The covering is to be gradually built
4. Each node will have its own optimum covering
5. For each node, find all matches (same as type 'covering' but with one gate only)
6. For each match, the total cost is sum of its own cost and the costs of all the optimum coverings at its inputs
7. Choose the match with minimum total cost and create the optimum covering for the node
8. Covering also will have only one gate with its inputs, where each input has its own covering, thus avoid redundancy, and its own cost
9. The problem is finding the match for a node
10. Each library gate has a 'complete' list of linked lists (strings) - these linked lists of strings represent all the possible paths from the output to each of the input nodes.
11. If all the strings lie in the subject graph starting at a node, then we have a match
12. Create the lists starting with leaves
13. Each list will have a matched indicator (bool)
14. Store the list of linked lists for the subject graph and also the library gates
15. We use subsequence matching between the lists of library gates and and those of the subject DAG to find a match at a particular node.
16. Assumption : each graph pattern has at most**10** linked lists, so a gate having **n** patterns can have at most**10n** linked lists

template<class Graph>structexercise_vertex { }

This is a functor (special type of mapping between categories) and it does operations on a vertex. Input argument needed is vertex descriptor i.e. *"void operator()(const Vertex& v) const".*

Now to list the out-edges of the vertex, the code used to do this is (this is just an example for familiarity with the boost code) -

```
std::cout<< "out-edges: ";
        typenameGraphTraits::out_edge_iteratorout_i, out_end;
        typenameGraphTraits::edge_descriptor e;
        for (tie(out_i, out_end) = out_edges(v, g);
        out_i != out_end; ++out_i) {
        e = *out_i;
        Vertex src = source(e, g), targ = target(e, g);
        std::cout<< "(" <<index[src] << ","
<<index[targ] << ") ";
```

The 'tie' function unpacks the element in a tuple (returned by in-edges) into the variables 'in_i' and 'in_end'.

```
for (tie(in_i, in_end) = in_edges(v,g); // tie function unpacks the elements in a ...
in_i != in_end; ++in_i) {          // ... tuple(returned by in_edges) into the variables in_i, in_end
        e = *in_i;
        Vertex src = source(e, g), targ = target(e, g);
        std::cout<<index[src] << ", " ;
```

template<class Graph>structextract_lists { }

extract_lists will list out all lists of a graph. Again, the input argument needed is vertex descriptor.

```
template<class Graph>structextract_lists {                                    extract_lists(Graph&
g_) : g(g_) {}
typedeftypenamegraph_traits<Graph>::vertex_descriptor Vertex;
void operator()(const Vertex& v) const                                          {
typedefgraph_traits<Graph>GraphTraits;
typenameproperty_map<Graph, vertex_index_t>::type
index = get(vertex_index, g);
typenameGraphTraits::out_edge_iteratorout_i, out_end;
typenameGraphTraits::edge_descriptor e;
typenameGraphTraits::in_edge_iteratorin_i, in_end;
```

void findMatch( )

This function finds all matches to node v and pushes them to ‘tempMatches’.

```
std::vector<int>coveredVertices;
std::vector<int>inputVertices;
std::vector<int>tempVertices;
```

The above part of the code stored the node ID’s of the vertices covered; node ID’s of the input vertices; and also creates a new temporary list of vertices respectively.

The string input “I” is for the node which is an input. The current match will be stored in the list of temporary matches in the vector named “matching_t”.

Now, cover the graph node by node, and for each node i, find all the patterns of the libgate; then creating a temporary list of strings of the subject DAG graph

```
std::vector<list_node*>subjectDAGlists;
```

Till now we have got all the strings in the pattern graph & all the strings in the subject graph as well. Basically, we have two lists now-

- subjectsDAGlists[n]
- lists[m]

```
int m = gates.at(i).gr_list_id[j];
while(lists[m]!=NULL) {
intstringmatchflag = 0;
for(int n=0;n<nbrDAGlists;n++)
```

Our next task is to match them, i.e. match strings of pattern graph to subject graph lists (also add the vertices to the temporary list of vertices to constitute the group of covered vertices).

```
int flag = 1;
list_node *travelnode1, *travelnode2;
travelnode1 = subjectDAGlists.at(n);
```

Set flag = 1, if a match is found (we need to verify if this holds true till the end of checking).

- travelnode1 is for DAG
- travelnode2 is for pattern graph

Starting from the point in DAG list that equals the given node, check if the DAG string contains the vertex V or not, if it does then check for other patterns also. If it doesn't, skip it and move to the next node.

```
while((travelnode2 ->nodeattr)!=input) {
if((travelnode2 ->nodeattr)==(travelnode1 ->nodeattr)){
tempVertices.push_back((travelnode1 ->nodeid));
travelnode1 = travelnode1 -> next;
travelnode2 = travelnode2 -> next;
}
```

Traverse till the input of the string of the pattern graph and string comparison was done. If we found a match, then add the vertices to the temporary list of vertices. If no match was found, BREAK the check and move ahead.

```
for(int p=0;p < tempVertices.size();p++)coveredVertices.push_back(tempVertices.at(p));
inputVertices.push_back((travelnode1 ->nodeid));
subjectDAGlists.erase(subjectDAGlists.begin()+n);
```

That's not all, if a match was found, flag is set to 1. In that case, add the temporary list to the covered vertices list, add the end of the string to the input vertices list, and ultimately removing string from the temporary list of strings of subject graph, assigning "stringmatchflag=1"

```
if(matchflag == 1){
for(int r=0;r<10;r++)    currentMatch.gate_type[r] = gates.at(i).name[r];
for(int s=0;s<coveredVertices.size();s++)
currentMatch.coveredVertices.push_back(coveredVertices.at(s));
for(intt=0;t<inputVertices.size();t++)
currentMatch.inputVertices.push_back(inputVertices.at(t));
currentMatch.cost = gates.at(i).cost;
tempMatches.push_back(currentMatch);
```

If stringmatchflag is set to 1, then our job is to add this match properties i.e. Gate type, covered vertices, input vertices, cost of the gate etc. to the temporary list of matches. The code used data structures to store the list of matches for a particular node.

void optimalCovering( )

This is a dynamic algorithm (recursive as well as iterative) that calculates “Minimum Cost Cover”.

We used the bottom to top approach; and used the Principle of Optimal Substructure for each node in optimal covering analysis.

```
while(out_i!=out_end) {
Vertex targ;
e = *out_i;
targ = target(e, g);
optimalCovering(g, targ);
out_i++;
}
```

The above code basically lists out all the input vertices of a node and runs the function optimalCovering() on each of them – a sequence of recursive calls.

```
if(out_i == out_end) {
if(get(get(&NodeProperty::nattr, g), v) == "i") {
char* pin = "input_pin";
for(int i=0;i<10;i++)     matches[index[v]].gate_type[i] = pin[i] ;
matches[index[v]].cost = 0 ;
matches[index[v]].coveredVertices.push_back(index[v]);
matches[index[v]].inputVertices.push_back(index[v]);
}
```

If there is only a single input and the property of the node is ‘i’, it means the node is an input pin and the cost of its covering is 0. This is the base condition for the recursive function.

```
for(intj=0;j<tempMatches.size();j++) {
tempcost = tempMatches.at(j).cost;
for(int k=0;k<tempMatches.at(j).inputVertices.size();k++) {
tempcost += matches[tempMatches.at(j).inputVertices.at(k)].cost;
}
if(tempcost<mincost) {
```

```
minindex = j;
mincost = tempcost;
}
}
```

Whereas for a normal node, our approach is changed a bit.

Here, findMatch() function gives all the matches (those gates that fit) and put them in an vector of tempMatches. In order to find the best matching, calculate the temporary cost of all matches in tempMatches. The combination giving the least cost will be our best possible match with cost as an optimization parameter.
Now we have the best match, write this to the matches array; add the covered vertices and the input vertices for each match.

Finally, according to the code,

The node is given by -> index[v]
The match for a node is given by -> matches[index[v]].gate_type
The match for a node is given by -> matches[index[v]].cost
-

void readGraphMLFile( )

```
void readGraphMLFile ( Graph&designG, string fileName ) {
boost::dynamic_propertiesdp;
        dp.property("name", get(&NodeProperty::nattr, designG));
std::ifstreamgmlStream;
gmlStream.open(fileName.c_str(), std::ifstream::in);
boost::read_graphml(gmlStream, designG, dp);
gmlStream.close();
}
```

This function used the boost library to read the graph properties, which are the logic functions at nodes. “nattr” is the property name and get( ) function gives us the property map. To see the format of the graphml file, you can find it in the starting of the code documentation. We have implemented an AND date in graphml file using the same format of the file (provided by Dilawar).

int main(intargc, char* argv[])

Read graphml file of the subject DAG (Directed Acyclic Graph). As stated earlier, we have assumed it to be already technology independent optimized.

Then we get a sequence container of gates after reading the graphml file, as shown in figure 3(a). Then readGraphMLFile( ) is called to make the graph g from the graphml file, which is a tree.

```
typedefproperty_map<Graph, vertex_index_t>::type IndexMap;
IndexMap index = get(vertex_index, g);
```

This is the property map type for the property vertex_index_t. Get( ) function was used to get a property map. Similarly, read graphml files of all graph patterns for each gate in the library.

To illustrate the below example, we have created our own graphml file for the implementation. Also written method to read properties of the gates given in the technology library.

We can easily read the library.txt file here, provided all lines have to end with a tab and all words in a line are separated by tab. The process is

- Identify the number of gates by string comparison, it's pre-stored in library.txt file
- Then, read the line up to first 100 characters
- Find the values corresponding to each parameter
- i.e. Read file name, Pattern_files, Gate_cost etc from the txt file

After the library file reading has been done -

```
for(int i=0;i<100;i++)   lists[i] = NULL;
listcount=0;
std::for_each(vertices(g).first, vertices(g).second, extract_lists<Graph>(g));
for(int i=0;i<nbr_library_gates;i++) {
for(int j=0;j<gates.at(i).nbrpatterns;j++) {
listcount = gates.at(i).gr_list_id[j];
std::for_each(vertices(gates.at(i).gr[j]).first, vertices(gates.at(i).gr[j]).second,
extract_lists<Graph>(gates.at(i).gr[j]));
```

```
} }
```

The above code extracts the linked lists of all the graph patterns.

lists is the data structure which store all the lists (branches) of the subject DAG and the library gates.
Since, every graph will have a list_id, the linked lists in the 'lists' array with indices (list_id) to (list_id+9)belong to that particular graph. list_id for the subject graph will be 0 always, so it owns indices 0-9.

```
list_node *travelnode;
for(int i=0;i<listcount;i++) {
travelnode = lists[i];
std::cout<< "string "<< i << " is " ;
while(travelnode!=NULL) {
std::cout<<travelnode ->nodeattr ;
travelnode = travelnode -> next;
}
std::cout<<std::endl;
}
```

The above block just prints out the linked lists (strings) (uncomment it in the main code to see how the lists how are stored).

And finally, find the match at the node, and ultimately find optimal covering of the graph g starting at the first node i.e. from the output node

```
optimalCovering(g, *vertices(g).first);
```

Compile Instructions:

Open the command terminal, go to the project file path; type the following command to compile the program. Here the file name is “techmapping.cpp”.
The folder should contain the files library.txt, AND_1.graphml, OR_1.graphml, NOT_1.grraphml and simple.graphml (the subject DAG).

To compile:

$ g++ techmapping.cpp -lboost_graph

To Debug:

$ g++ -g techmapping.cpp -lboost_graph

$ gdba.out

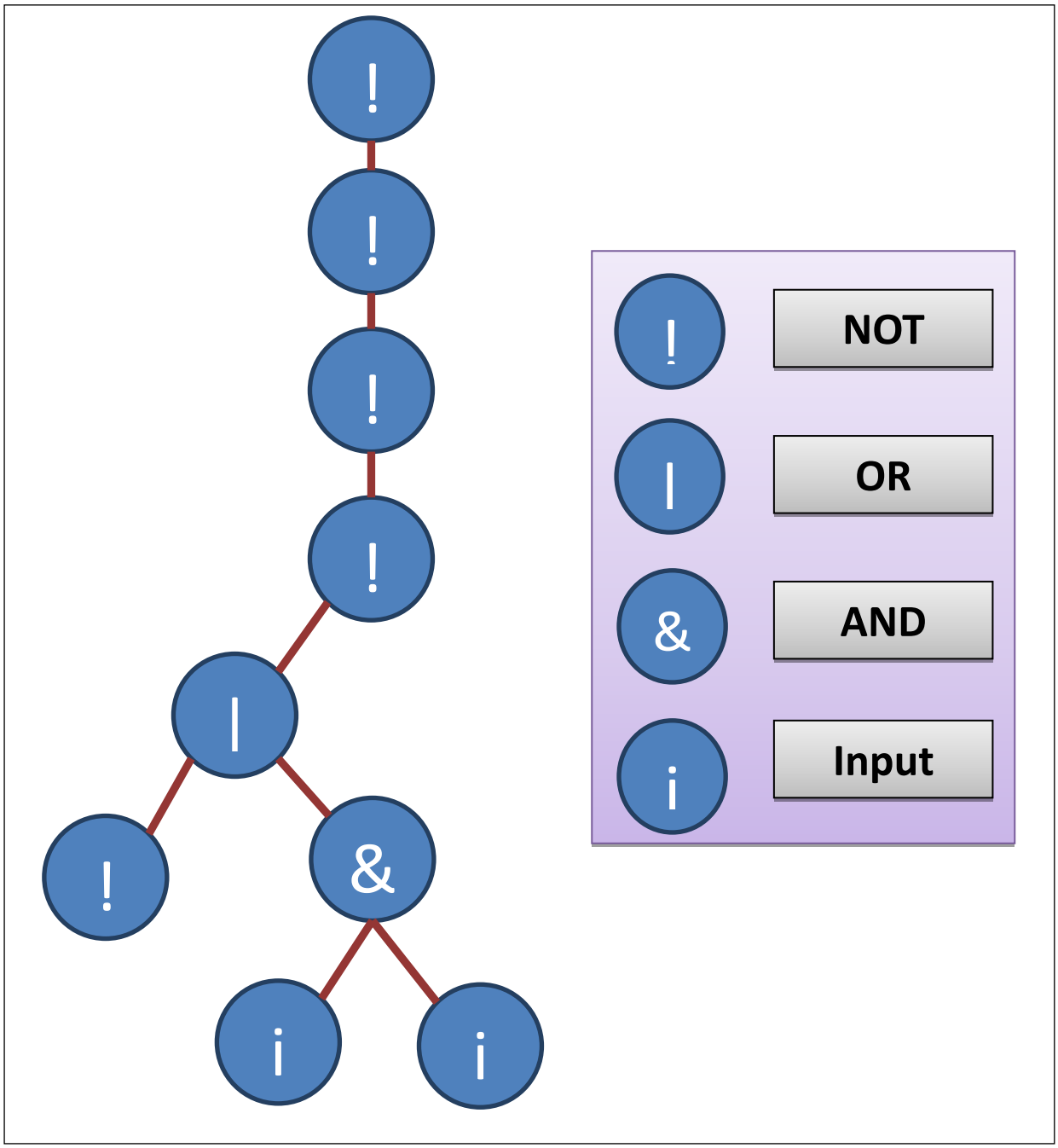
<gdb> run

<gdb>backtrace

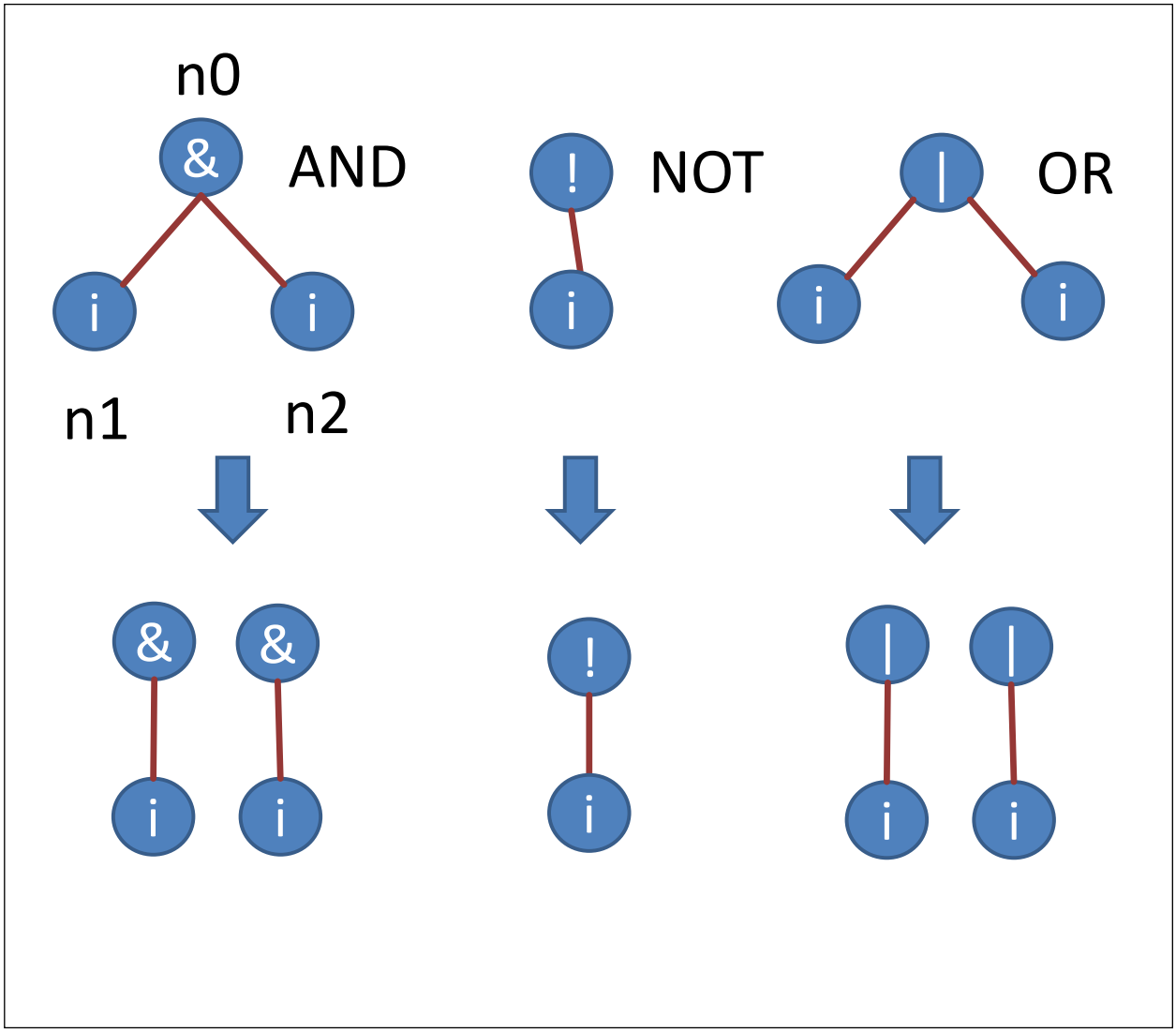
<gdb> frame 3,2,1,0

<gdb> quit

Algorithm Example/Illustration



In this illustration, we will use our algorithm to do the technology mapping and optimizing for the cost i.e. Delay, Area or Power. Also analyze the technology mapping problem for the given graph.



Here, we have splitted the gates in the technology library into strings to be used for comparison with the strings generated from subject graph, thus to find if there is a possibility of replacing the combination of gates with the Standard gate given in the technology library.

In this case, the nodes we have chosen for this example are –> AND, NOT, OR

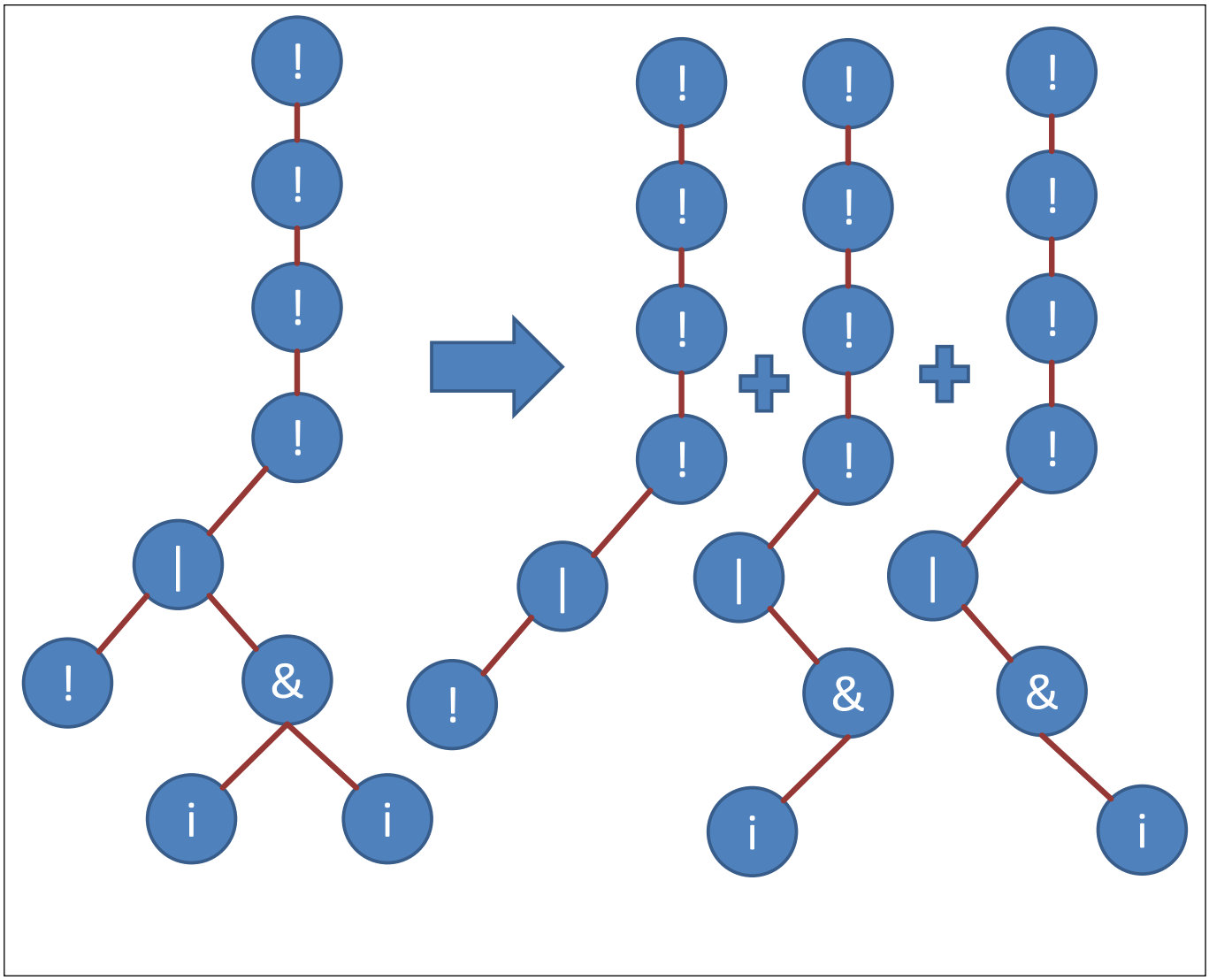


Figure 2 (b)

The above figure 2(b) is generated by generating possible strings from the given graph into linear graphs to be used for comparison i.e. comparing strings in the PATTERN GRAPH and strings in the SUBJECT GRAPH.

The left most graph i.e. Subject graph, can be splitted into 3 linear strings consisting of nodes, covering all possible combinations to be used in comparison. Similarly, we will also generate strings from the gates given in the technology library and ultimately comparing the string obtained from both the graphs to see if there can be a replacements to optimize the cost (power, delay, area etc.)

Note : The graph is already assumed to be decomposed into a pattern of AND, OR and NOT gates.

Outputs:

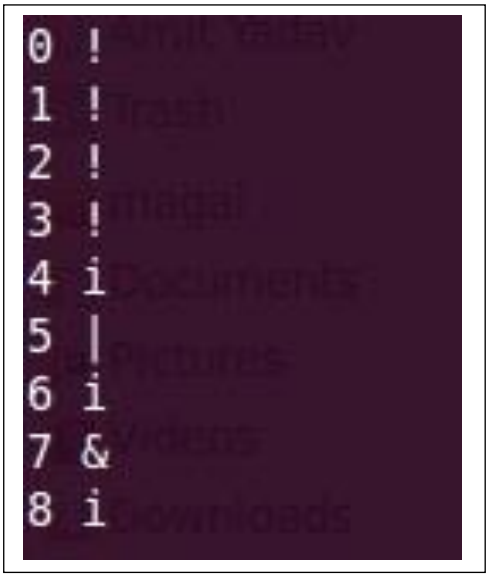


Figure 3(a)

Figure 3(a) shows the nodes and the corresponding logic gates associated with it. The type of gates are obtained after reading the given graphml file.

Figure 3(b) shows the all string combinations which we used in the code to match to see if there is a replacement possible or not using the given gates from the technology library.

For example, the first 3 lines in figure 3(b), shows the possible string combinations which can be verified from the Figure 2 having 6 nodes, 7 nodes and 7 nodes respectively (moving from left to right).

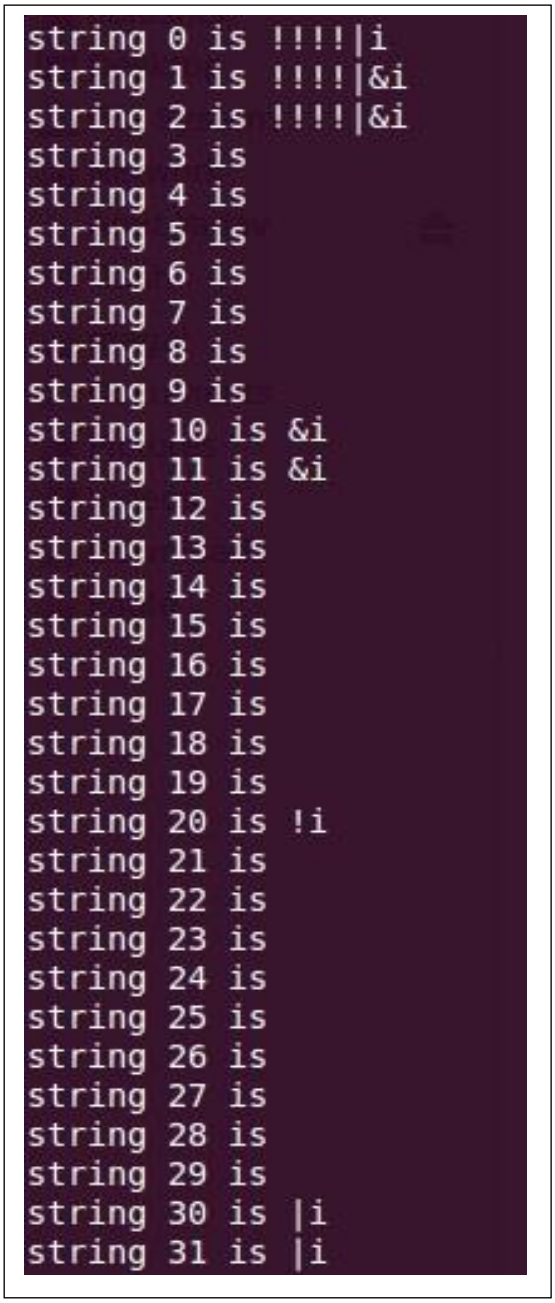


Figure 3(b)

Command Prompt Result:

```
The match for node 4 is input_pin and its cost is 0

The match for node 6 is input_pin and its cost is 0

The match for node 8 is input_pin and its cost is 0

One match for the node 7 is AND
The inputs for this match are nodes 6 8
The match for node 7 is AND and its cost is 2

One match for the node 5 is OR
The inputs for this match are nodes 4 7
The match for node 5 is OR and its cost is 5

One match for the node 3 is NOT
The inputs for this match are nodes 5
The match for node 3 is NOT and its cost is 6

One match for the node 2 is NOT
The inputs for this match are nodes 3
The match for node 2 is NOT and its cost is 7

One match for the node 1 is NOT
The inputs for this match are nodes 2
The match for node 1 is NOT and its cost is 8

One match for the node 0 is NOT
The inputs for this match are nodes 1
The match for node 0 is NOT and its cost is 9
```

Figure 3(c)

For the given example, figure 3(b) is the output generated in the command prompt.

It shows the match for the node with the gates present in the technology library, input pins to the node chosen for matching and also the cost associated with it.

We can see that for the node 7, AND gate is a match and cost is 2. Gradually we will cover other nodes also i.e. from node 7 to node 0 and the cost increases subsequently. We finally choose the best possible combination of gates from technology library in order to optimize for the cost. In our simple graph, there is only a single possible matching.