10/12/2015

Project 1

VLSI Circuit Partitioning

-by Fiduccia-Matthesyses Algorithm

Varunsrivathsa Venkatesha
SBU ID: 110457727
EMAIL: varunsrivathsa.venkatesha@stonybrook.edu

Table of Contents

List of Figures

Problem Statement:

To implement and experiment the Fiduccia-Mattheyses partitioning algorithm implemented for gate-level designs. Objective is to minimize the cutset-size while meeting given area constraints fixed for the partitions.

Introduction:

A problem can be solved efficiently by dividing it into number of smaller problems. In modern circuit there exists millions of transistors to deal with. Partitioning is such a technique to divide a circuit or a system in collection of subsystems. Each subsystem can be designed individually and then combined to speed up the design process.
There are three types of partitioning:

- System level partitioning: Whole system is divided in to subsystems. And the subsystems can be designed independently.
- Board level partitioning: Circuit assigned to each board is divided in to smaller sub circuits.
- Chip level partitioning: Circuits assigned to each chip can be further divided in to smaller units.

Partitioning of the circuit can be done using many Algorithms. Fiduccia-Mattheyses is one such algorithm which is a classical approach that operates on Hypergraph model.

Related Work:

A circuit can be partitioned using many Algorithms. Few of them are Fiduccia-Mattheyses algorithm, Kernighan–Lin Algorithm, Simulated annealing Algorithm. Kernighan–Lin Algorithm solves the NP-hard Balanced Bi-Partitioning Problem, where the given gate-level circuit is divided into two equal-sized partitions. It follows "gain-based cell swap". Simulated annealing is a randomized local search algorithm. This algorithm is a generic probabilistic metaheuristic for the global optimization problem of locating a good approximation to the global optimum of a given function in a large search space. It is often used when the search space is discrete. Fiduccia-Mattheyses algorithm, improvement over Kernighan-Lin algorithm is a classical approach that operates directly on the Hypergraph model.

Fiduccia-Mattheyses is a modified version of Kernighan- Lin algorithm. The first modification is that only a single vertex is moved across the cut in a single move. This permits the handling of unbalanced partitions and non-uniform vertex weights. The other modification is the extension of the concept of cutsize to hypergraphs. Finally, the vertices to be moved across the cut are selected in such a way so that the algorithm runs much faster at O(n). As in Kernighan-Lin algorithm, a vertex is locked when it is tentatively moved. When no moves are possible, only those moves which give the best cutsize are actually carried out.

Flowchart of FM algorithm:

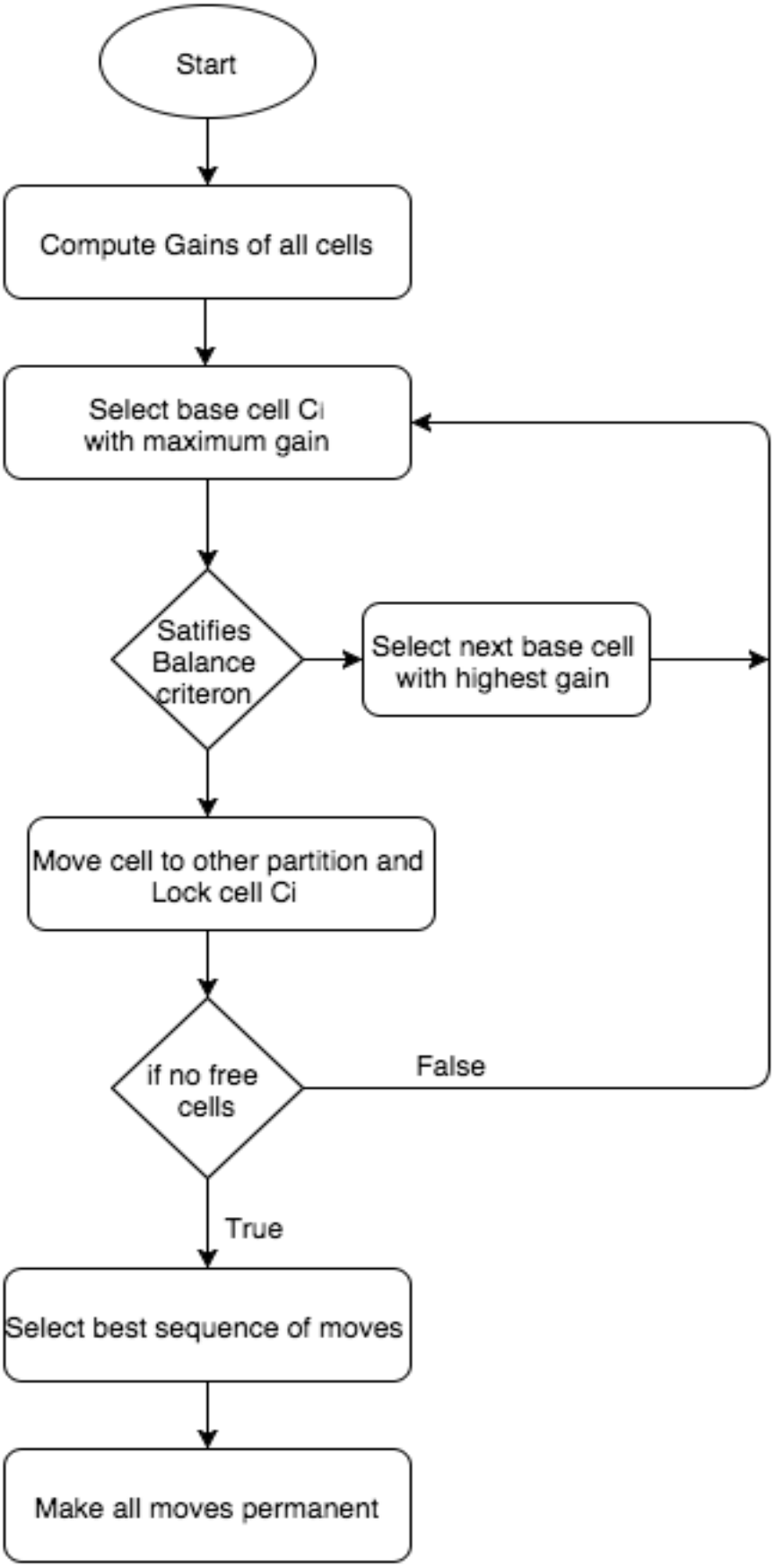

Figure 1: Flowchart of FM Algorithm

Steps in detail:

Step1: Gain of all a cell *x* is calculated by using the formula
$gain = FS(x) - TE(x)$
where F(s) = Moving force
T(s) = Retention force

Step2: Add all the to gain bucket structure, which helps to select the cell with maximum gain to move from P1 to P2.

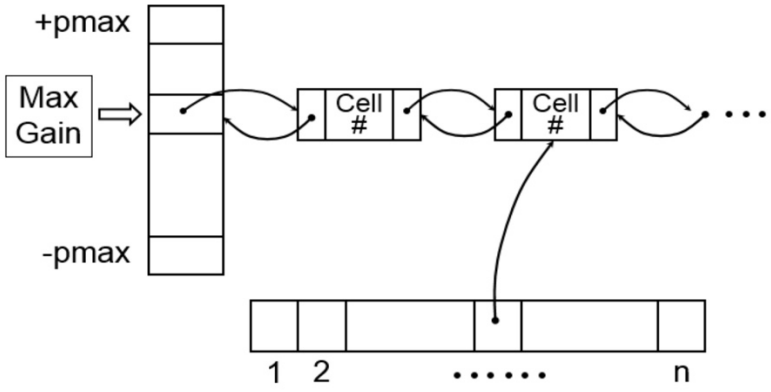

Figure 2: Gain bucket data structure

Step3: Select the cell with maximum gain and if it satisfies balance criterion move the cell to other partition. Lock the moved cell.

- Balance criterion:
  $[r.area(G) - areamax(G)] \leq area(A) \leq [r.area(G) + areamax(G)]$

  where *1)* $r = \frac{area(A)}{area(A)+area(B)}$

  2) *area(A)* and *area(B)* are areas partition A and B
  3) *area(G)* is the total area of graph
  4) *areamax(G)* is the maximum cell area in G

Step4: Update gains values of all the cells which are connected to the moved cell and calculate cutset size.

Step5: Repeat steps 3 and 4 until are the cells are locked.

Proposed Solution and Implementation:

Proposed Solution:

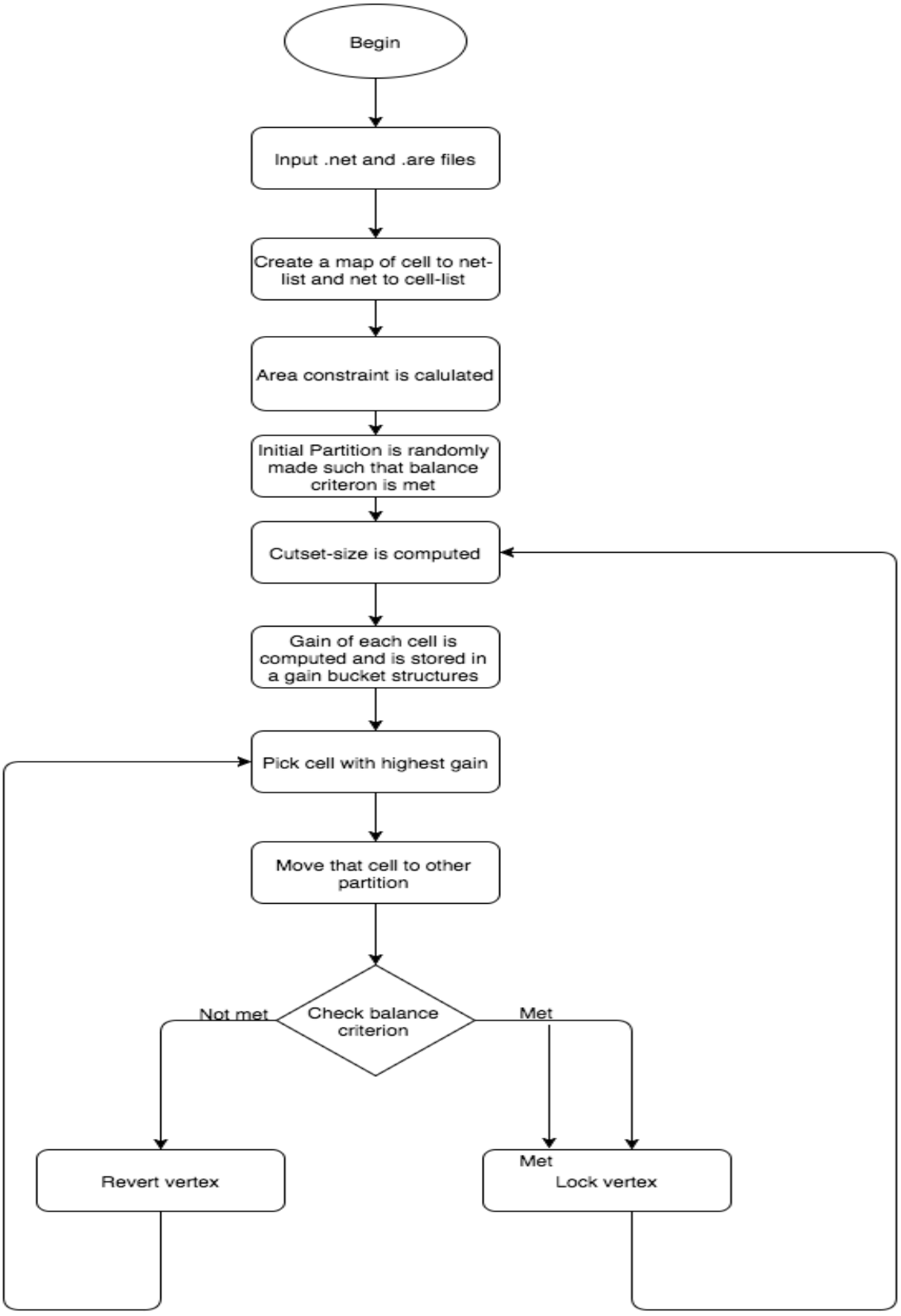

Figure 3: Flowchart of proposed solution

Implementation:

Code implemented has two files 'main.cpp' and 'cell.h'

1) The file 'vertex.h' has a class cell which has methods
   a) 'retPartition' – T get the partition of cell.
   b) 'chckLocked' – Returns if the cell the is locked or not.
   c) 'retGain' – Returns the gain of cell.
   d) 'changePartition' – Changes the partition of cell.
2) The file 'main.cpp' has functions
   a) 'rdAreFile' and 'rdNetFile' – Reads the given .net and .are file.
   b) 'CellNumToCellMap' – Prints cell number to cell map.
   c) 'NetToCellMap' – Prints net to cell map.
   d) 'givePartition' – Assigns partition to each cell randomly.
   e) 'calcFs' – Returns F(s) value of each cell.
   f) 'calcTs' – Returns T(s) value of each cell.
   g) 'retGain' – Computes gain and returns gain.
   h) 'cutSize' – Returns cutset size of the graph.
   i) 'gainBucket' – Creates a map of gains to list of cells.
   j) 'retAreaPartition' – Returns area of the first partition.
   k) 'calcTotalArea' – Return area of total graph.
   l) 'calcMaxCell' – Returns the cell with maximum area and its area.
   m) 'chkBalanceCriterion' – Checks the Balance criterion and returns a Boolean value 0 or 1.
   n) 'updateGain' – Updates the gain cells after each move.
   o) 'MoveCell' – Starts to move cell from one partition to other by toggling partition bits.

Experimental Results:

Benchmark files	Number of nodes	Ratio cut	Initial Cut set size	Min Cut size Achieved	% reduction in cutset size
File1.hgr	8	0.3	3	2	33
File1.hgr	8	0.6	6	3	50
File2.hgr	18	0.3	3	2	33
File2.hgr	18	0.6	4	2	50
File3.hgr	20	0.4	3	1	66
File3.hgr	20	0.6	4	1	75

Conclusion:

Fiduccia-Mattheyses algorithm is implemented in 'C++' for a given .are and .net file. Code was tested on three input files and the outputs are tabulated.

Bibliography:

[1] B.W Kernighan and S. Lin –An Efficient Heuristic procedure for partitioning Graphs, The Bell system technical journal, Vol.49 ,Feb 1970,pp.291-307.

[2] A linear-time heuristic for improving Networt Partitions by C.M. Fiduccia and R.M Metheyeses.

[3] Text Book: Algorithms for VLSI physical design automation, third edition by Naveed A. Sherwani.

[4] Prof. Alexa Doboli Notes.

[5] http://users.ece.gatech.edu/limsk/book/slides/pdf/FM-partitioning.pdf

Appendix:

1)‘vertex.h’

```
class vertex
{
 public:
 int  vertexID;
 int partition;
 bool chkLocked;
 int  gain;
 int  area;
 std::list<int> netList;
 bool retPartition();
 bool chkLocked()
  {
    return isLocked;
  }
 void setIsLocked()
   {
            isLocked = true;
  }
 bool retGain();
 void changePartition()
  {
    if(partition == 0)
            partition = 1;
    else //if(partition ==1)
            partition = 0;
  }
};
```

2)‘main.cpp’

```
#include <iostream>
#include<iostream>
#include<fstream>
#include<vector>
#include<map>
#include<list>
#include <sstream>
#include <cstdlib>
#include<string>
#include<queue>
#include<algorithm>
#include "cell.h"

using namespace std;

int num_cells = 8;
int num_nets;
int cutSetSize = 0;
float ratioCut = 0.6;
```

```
int areaOfPartitionA = 0;
int areaOfPartitionB = 0;
int totalArea = 0;
int maxAreaCell = 0;
map<int,Cell> cellIdToCellMap;
map<int, list<Cell> > netToCellListMap;
map<int, list<int> > gainToCellIdListMap;
vector<int> gainVector;

void rdAreFile();
void rdNetFile();
std::list<Cell> split(std::string str,char delimiter);
void CellNumToCellMap (/*map<int,Cell> cellIdToCellMap*/);
void NetToCellMap (/*map<int, list<int> > netToCellListMap*/);
void givePartition();
int calcFs(int A);
int calcTs(int A);
int retGain(int A);
int cutSize();
int gainBucket();
int retAreaPartition(int partition);
int calcTotalArea();
int calcMaxCell();
bool chkBalanceCriterion();
void updateGain(int cellId);
void MoveCell();

int main ()
{
rdAreFile();
givePartition();
rdNetFile();

CellNumToCellMap (/*cellIdToCellMap*/);
NetToCellMap(/*netToCellListMap*/);
//int s = calcFs(3);
int g = retGain(3);
//cout<<"FofS is  "<<s<<endl;
cout<<"Gain of  is   "<<g<<endl;
cutSetSize = cutSize();
cout<<"Cutset size "<<cutSetSize<<endl;
gainBucket();
int a = retAreaPartition(0);
cout<<"area of partition 0 is "<<a<<endl;
int b = retAreaPartition(1);
cout<<"area of partition 1 is "<<b<<endl;
totalArea = calcTotalArea();
cout<<"Total area is "<<totalArea<<endl;
maxAreaCell = calcMaxCell();
cout<<"Max area among Cells "<<maxAreaCell<<endl;
bool isAreaConstraint = chkBalanceCriterion();
cout<< " area constraint is "<<isAreaConstraint<<endl;
MoveCell();
return 0;
```

```
}

std::list<Cell>& split(std::string str,char delimiter,int i,std::list<Cell>& cellList)
{
   std::stringstream sp(str);
  // std::list <Cell> data;
   std::string t;
   int t1;
   while(getline(sp,t,delimiter))
   {

      t1=atoi(t.c_str());
      Cell temp = cellIdToCellMap[t1];
                        //temp.netList.push_back(i);
                        cellIdToCellMap[t1].netList.push_back(i);
                        //cout<<"t1  "<<t1<<endl;
                        //cout<<"temp.CellID "<<temp.cellID<<endl;
      cellList.push_back(temp);
   }
   return cellList;
}
void rdNetFile()
{
            string line;
                        ifstream myfile("XX.hgr");
            if(!myfile)
            {
            cout<<"File cannot be opened" << endl;
            }
            else
            {
                        int a=0;
            while(getline(myfile, line))
            {
                                    list<Cell> cellList;
            if(a >= 1)
            {

                                                cellList = split(line,' ',a,cellList);
                                                netToCellListMap.insert(pair<int,list<Cell> >(a,cellList));
                                    }
                                    else
                                    {
                                                std::stringstream sp(line);
                        std::string t;
                                                int b[3];
                                                int i=0;
                        while(getline(sp,t,' '))
                                                {
                                                            b[i] = atoi(t.c_str());
                                                            i++;
                                                }
                                                num_nets = b[0];
                                                num_cells = b[1];
```

```
                                        }

                                        list<Cell>::iterator i;
                                for( i =cellList.begin(); i != cellList.end(); ++i)
                                        cout << (*i).cellID << " ";
                                cout << endl;
                        a++;
                        }
                        }
   cout<<"num_nets  "<<num_nets<<endl;
   cout<<"num_cells "<<num_cells <<endl;
}
void rdAreFile()
{
                string line;
                int cellCount = 1;

                ifstream myfile("ibm01.are");
                if(!myfile)
                {
                cout<<"File cannot be opened" << endl;
                }
                else
                {
                                while(getline(myfile,line)&& (cellCount<=num_cells) )
                {
                std::stringstream sp(line);
                std::string t;
                                        Cell temp;
                                        int b[2];
                                   int i=0;
                while(getline(sp,t,' '))
                                        {
                                                b[i] = atoi(t.c_str());
                                                i++;
                                        }
                                        //cout<<b[0]<<" "<<endl;
                                        cout<<b[1]<<" "<<endl;
                                        temp.cellID = cellCount;
                                        temp.area = b[1];
                                        cellIdToCellMap.insert(pair<int,Cell>(cellCount,temp));
                                        cellCount ++;
      }
      cout<<"Cell Count is "<<cellCount-1<<endl;
   }

}
void CellNumToCellMap (/*map<int,Cell> cellIdToCellMap*/)
{

 cout<<"CellID      Cell    area            partition   netlist      "<<endl;
 for(map<int,Cell>::const_iterator it1 = cellIdToCellMap.begin();
   it1!= cellIdToCellMap.end(); ++it1)
   {
```

```
                        cout<<it1->first<<"  "<<(it1->second).cellID<<"      "<<(it1->second).area<<"       "<<(it1-
>second).partition<<"                                ";
                        for(list<int>::const_iterator it2 = (it1->second).netList.begin(); it2!=(it1-
>second).netList.end();++it2)
                        cout<<(*it2)<<" ";
                        cout<<endl;
                }


}
void NetToCellMap (/*map<int, list<Cell> > netToCellListMap*/)
{

   //cout<<"printing values of maps"<<endl;
   cout<<endl<<"net                Cell List "<<endl;
                for(map<int, list<Cell> >::const_iterator it1 = netToCellListMap.begin();
   it1!= netToCellListMap.end(); ++it1)
   {
     cout<<it1->first<<"           ";
     for(list<Cell>::const_iterator it2 =  it1->second.begin(); it2 != it1->second.end(); ++it2)
     {
                    cout<<(*it2).cellID<<" ";
               }
     cout<<endl;
   }
}
void givePartition()
{
                cellIdToCellMap[1].partition = 0;
                cellIdToCellMap[2].partition = 1;
                cellIdToCellMap[3].partition = 0;
                cellIdToCellMap[4].partition = 0;
                cellIdToCellMap[5].partition = 0;
                cellIdToCellMap[6].partition = 1;
                cellIdToCellMap[7].partition = 0;
                cellIdToCellMap[8].partition = 1;
}


int calcFs(int A)
{
  int FofS = 0;

  Cell temp =           cellIdToCellMap[A];
  //cout<<"temp.partition = "<< temp.partition<<endl;
  for(list<int> ::const_iterator it1 = cellIdToCellMap[A].netList.begin(); it1!=cellIdToCellMap[A].netList.end();++it1)
                {
                        int FofS_net =0;
                for(list<Cell>::const_iterator it2 = netToCellListMap[*it1].begin(); it2!= netToCellListMap[*it1].end();++it2)
                                        {

                                                if(temp.partition == it2->partition)
                                                {
                                                        FofS_net++;
```

```
                                        }
                                    cout<<"current Cell ID "<<it2->cellID<<" "<< "partition " << it2->partition
<<endl;
                                    }
                if(FofS_net<=1)
                            FofS++;
                }
  return FofS;

}
int calcTs(int A)
{
            int TofS = 0;
             Cell temp = cellIdToCellMap[A];
            for(list<int> ::const_iterator it1 = cellIdToCellMap[A].netList.begin();
it1!=cellIdToCellMap[A].netList.end();++it1)
                        {

                        int TofS_net =0;
            for(list<Cell>::const_iterator it2 = netToCellListMap[*it1].begin(); it2!= netToCellListMap[*it1].end();++it2)
                                    {
                                                if(temp.partition != it2->partition)
                                                {
                                                            TofS_net++;
                                                }
                                    //          cout<<"current Cell ID "<<it2->cellID<<" ";//<< "partition " << it2->partition
<<endl;

                                    }
                 if(TofS_net==0)
                            TofS++;
                        }

  return TofS;
}

int retGain(int A)
{
   int FofS=calcFs(A);
   int TofS=calcTs(A) ;
            return (FofS-TofS);
}

int cutSize()
{
            int cutsize = 0;
   for(map<int, list<Cell> >::const_iterator it1 = netToCellListMap.begin();
   it1!= netToCellListMap.end(); ++it1)
   {
                Cell temp = it1->second.front();
     for(list<Cell>::const_iterator it2 =  it1->second.begin(); it2 != it1->second.end(); ++it2)
     {

                        if(temp.partition != it2->partition)
```

```
                        {
                                cutsize++;
                                break;
                        }
                }
        }
        return cutsize;
}

int gainBucket()
{
        int gain =0;
        for(map<int,Cell>::iterator it1 = cellIdToCellMap.begin();it1!= cellIdToCellMap.end(); ++it1)
        {
                gain = retGain(it1->first);
                (it1->second).gain = gain;
        //      gainVector.push_back(gain);
        //      cout<<"it1->first "<< it1->first<<endl;
                map<int, std::list<int> >::iterator finder;
                finder = gainToCellIdListMap.find(gain);
                if(finder==gainToCellIdListMap.end())
                {
                        list<int> celllist;
                        celllist.push_back(it1->first);
                        gainToCellIdListMap.insert(pair<int,list<int> >(gain,celllist));

                }
                else
                finder->second.push_back(it1->first);
        }
        sort(gainVector.begin(), gainVector.end());
        reverse(gainVector.begin(), gainVector.end());
        for(vector<int>::iterator it1 = gainVector.begin(); it1!= gainVector.end(); ++it1)
                cout<< *it1 <<endl;

                        cout<<"gain             Cell list"<<endl;
   for(map<int,list<int> >::const_reverse_iterator it1 = gainToCellIdListMap.rbegin(); it1 != gainToCellIdListMap.rend();
++it1)
                {
                        cout<<it1->first<<"  ";
                        for(list<int>::const_iterator it2 = (it1->second).begin(); it2!= (it1->second).end(); ++it2)
                        {
                                cout<<*it2<<" ";
                        }
                        cout<<endl;
                }

}
int retAreaPartition(int partition)
{
        int sumArea = 0;
        for(map<int,Cell>::const_iterator it1 = cellIdToCellMap.begin();
   it1!= cellIdToCellMap.end(); ++it1)
   {
```

```
                        if(it1->second.partition == partition)
                        {
                                sumArea+=it1->second.area;
                        }
                }
                return sumArea;
}

int calcTotalArea()
{
                return (retAreaPartition(0) + retAreaPartition(1));
}
int calcMaxCell()
{
                int maxArea = cellIdToCellMap[1].area ;
                for(map<int,Cell>::const_iterator it1 = cellIdToCellMap.begin();
   it1!= cellIdToCellMap.end(); ++it1)
   {
                        if(maxArea < (it1->second).area)
                        {
                                maxArea=it1->second.area;
                        }
                }
                return maxArea;
}

bool chkBalanceCriterion()
{
   int areaPartitionA = retAreaPartition(0);
                int areaPartitionB = retAreaPartition(1);
                //cout<<"ratio cut is "<<ratioCut<<endl;
                //cout<<"total area is "<<totalArea<<endl;
                //cout<<"range 1 is "<<(ratioCut*totalArea - maxAreaCell)<<endl;
                //cout<<"partition A is "<<areaPartitionA<<endl;
                //cout<<"range 2 is "<<
                if (((ratioCut*totalArea - maxAreaCell)<= areaPartitionA)&&(areaPartitionA<=(ratioCut*totalArea +
maxAreaCell)))
                        return true;
                else
                        return false;
}
void MoveCell()
{
   int initialCutset = cutSize();
                int cutSet = 0;
                int i=0;
                cout<<"initial Cutset = "<<initialCutset<<endl;
                int cellId =0;
                bool stop = false;

                        for(map<int,list<int> >::reverse_iterator it1 = gainToCellIdListMap.rbegin(); it1 !=
gainToCellIdListMap.rend(); ++it1)
                        {
                                cout<<"for cell of gain "<<it1->first<<endl;
```

```
                        do{
                                for(list<int>::iterator it2 = (it1->second).begin(); it2!= (it1->second).end();
++it2)
                                {
                                        if(cellIdToCellMap[*it2].getIsLocked()== false)
                                                {
                                                        cellIdToCellMap[*it2].changePartition();
                                                        if(chkBalanceCriterion() == false)
                                                        {

        cellIdToCellMap[*it2].changePartition();
                                                                continue;
                                                        }
                                                        cellIdToCellMap[*it2].setIsLocked();
                                                        cout<<"updating gain bucket"<<endl;
                                                        int temp = *it2;
                                                        list<int>::iterator it3 = it2;

                                                        //(it1->second).remove(*it3);
                                                        updateGain(temp);
                        cutSet =  cutSize();
                                                i++;
                        //cout<<"cutset size after "<< i <<" move =  "<<cutSet<<endl;
                                                }
                                }

                        }while(it1->first > 0);
                }

}
void updateGain(int cellId)
{
        int gain =0;
        for(list<int> ::const_iterator it1 = cellIdToCellMap[cellId].netList.begin();
it1!=cellIdToCellMap[cellId].netList.end();++it1)
        {

        for(list<Cell>::const_iterator it2 = netToCellListMap[*it1].begin(); it2!= netToCellListMap[*it1].end();++it2)
                {
                        //if(cellId == it2->cellID)
                        //continue;
                        gain =      retGain(it2->cellID);
                        map<int, std::list<int> >::iterator finder;
                        finder = gainToCellIdListMap.find(gain);
                        if(finder==gainToCellIdListMap.end())
                        {
                                list<int> celllist;
                                celllist.push_back(it2->cellID);
                                gainToCellIdListMap.insert(pair<int,list<int> >(gain,celllist));

                        }
                        else
                        finder->second.push_back(it2->cellID);
```

```
                }
        }
        cout<<"updated gain bucket is"<<endl;
        cout<<"gain         Cell list"<<endl;
  for(map<int,list<int> >::const_reverse_iterator it1 = gainToCellIdListMap.rbegin(); it1 != gainToCellIdListMap.rend();
++it1)
                {
                        cout<<it1->first<<"  ";
                        for(list<int>::const_iterator it2 = (it1->second).begin(); it2!= (it1->second).end(); ++it2)
                        {
                                cout<<*it2<<" ";
                        }
                        cout<<endl;
                }
}
```