CS6135 HW2 Report

109062509 熊 磊

I. Abstract

Let *C* be a set of cells and *N* be a set of nets. Each net connects a subset of cells. The two-way min-cut partitioning problem is to partition the cell set into two groups A and B. The cost of a two-way partitioning is measured by the cut size, which is the number of nets having cells in both groups. In this homework, I implement **FM ALGORITHM** to solve the problem of two-way min-cut partitioning.

II. How to compile and execute?

- Compilation:
  One can go to directory, `HW2/src`, and execute `make compile`, the executable file, named as `hw2`, will be generated in directory, `HW2/bin`.

```
[g109062509@ic23 src]$ make compile
```

- Execution
  In directory, `HW2/bin`, execute `hw2` follow by the supporting arguments, one can change the input or output file address. It's necessary to provide `.nets`, `.cells`, and the output file, otherwise, it will have error.

```
[g109062509@ic23 bin]$ ../bin/hw2 -n ../testcases/p2-1.nets -c ../testcases/p2-1.cells -o ../output/p2-1.out -m
```

III. The final cut size and the runtime of each testcase

	p2-1	p2-2	p2-3	p2-4	p2-5
I/O time (s)	0.002	0.041	0.837	1.606	4.674
Computation time (s)	0.015	1.056	330.211	537.496	390.207
Runtime (s)	0.02	1.12	331.31	539.28	395.02
Initial Cut size	78	894	26,797	63,941	148,218
Final Cut size	**8**	**368**	**4,397**	**45,517**	**131,045**

IV. Implementation details

a. Where is the difference between your algorithm and FM Algorithm described in class? Are they exactly the same?

Most of the concept is follow the slides introduced in the class, but the initial partition and the data structure I implemented is slightly different, which I will explain in **e.**

b. Did you implement the bucket list data structure?

In my implementation, bucket lists are rather two gain lists, with `std::map<int, set<int>>` structure. One for partition A, and the other is for partition B. It will save the set of cells which having the same gain in its partition, if the cell is not locked.

If there is no free cell for a given gain in the partition, then the key of the map will be erased.

The reason to use map is because `std::map` will automatically sort the element by key in ascending order. Hence, later we can use *reverse iterator* `.rbegin()` to get the MAX gain cell. Also, the time complexity to get element by key is O(1), this can help us to facilitate `update_gain()` function.

```
467  cell* getMaxGain (vector<cell*> &Cells, map<int, set<int, MyCompare>> &gainA, map<int, set<int, MyCompare>> &gainB,
468                    int &ASize, int &BSize)
469  {
470      map<int, set<int, MyCompare>>::reverse_iterator maxGain, maxGainA, maxGainB;
471      if (gainA.empty() && gainB.empty()) return nullptr;
472      else if (gainA.empty())
473      {
474          maxGain = gainB.rbegin();
475          if (!isBalance(ASize + Cells[*(maxGain->second.begin())]->size, false)) return nullptr;
476      }
477      else if (gainB.empty())
478      {
479          maxGain = gainA.rbegin();
480          if (!isBalance(BSize + Cells[*(maxGain->second.begin())]->size, false)) return nullptr;
481      }
482      else
483      {
484          maxGainA = gainA.rbegin(); maxGainB = gainB.rbegin();
485          if (maxGainA->first >= maxGainB->first)
486          {
487              if (isBalance(BSize + Cells[*(maxGainA->second.begin())]->size, false)) maxGain = maxGainA;
488              else if (isBalance(ASize + Cells[*(maxGainB->second.begin())]->size, false)) maxGain = maxGainB;
489              else return nullptr;
490          }
491          else
492          {
493              if (isBalance(ASize + Cells[*(maxGainB->second.begin())]->size, false)) maxGain = maxGainB;
494              else if (isBalance(BSize + Cells[*(maxGainA->second.begin())]->size, false)) maxGain = maxGainA;
495              else return nullptr;
496          }
497      }
498      return Cells[*(maxGain->second.begin())];
499  }
```

c. How did you find the maximum partial sum and restore the result?

In each iteration of partition, I moved the cell in greedy way under the constraint. If there is no available to move, or each action will cause violating the constraint, this is the end of the current iteration.

At every step of moving the cell, we will plus the gain of the cell to the variable `curPartialSum`, which is initialize with zero; and before moving the next cell, we will compare the `curPartialSum` with `greatestPartialSum`, also initialize with zero in each iteration. If `curPartialSum` > `greatestPartialSum`, then update the `greatestPartialSum` with `curPartialSum`, and record the number of steps so far in this iteration.

Until an iteration is stop, trace back to the step which attains `greatestPartialSum`, to restore the partition.

```
int fm(vector<net*> &Nets, vector<cell*> &Cells, set<int> &A, set<int> &B,
       map<int, set<int, MyCompare>> &gainA, map<int, set<int, MyCompare>> &gainB,
       int &ASize, int &BSize)
{
    int curPartialSum = 0, greatestPartialSum = 0, greatestPartialSumStep = 0;
    bool fromAtoB = false;
    stack<cell*> movedOrder;
    cell* movedCell = getMaxGain(Cells, gainA, gainB, ASize, BSize);
    while (movedCell != nullptr)
    {
        movedOrder.push(movedCell);
        fromAtoB = movedCell->isA;
        movedCell->isLock = true;
        curPartialSum += movedCell->gain;
        updateGain (movedCell, fromAtoB, false, Nets, Cells, A, B, gainA, gainB);
        move(movedCell, fromAtoB, Nets, A, B, ASize, BSize, gainA, gainB);
        updateGain (movedCell, fromAtoB, true, Nets, Cells, A, B,  gainA, gainB);

        movedCell = getMaxGain(Cells, gainA, gainB, ASize, BSize);
        if (greatestPartialSum < curPartialSum)
        {
            greatestPartialSum = curPartialSum;
            greatestPartialSumStep = movedOrder.size();
        }
        if (chrono::duration_cast<chrono::seconds> (chrono::steady_clock::now() - t_begin).count() > 570) break;
    }
    auto reverseStep = movedOrder.size() - greatestPartialSumStep;
    for (auto i = 0; i < reverseStep; ++i)
    {
        movedCell = movedOrder.top();
        movedOrder.pop();
        movedCell->isLock = false;
        fromAtoB = movedCell->isA;
        updateGain (movedCell, fromAtoB, false, Nets, Cells, A, B, gainA, gainB);
        move(movedCell, fromAtoB, Nets, A, B, ASize, BSize, gainA, gainB);
        updateGain (movedCell, fromAtoB, true, Nets, Cells, A, B,  gainA, gainB);
    }
    return greatestPartialSum;
}
```

d. Please compare your results with the top 5 students' results from last year and show your advantage either in runtime or in solution quality. Are your results better than them?

	Runtime (s)					Cutsize				
Rank	p2-1	p2-2	p2-3	p2-4	p2-5	p2-1	p2-2	p2-3	p2-4	p2-5
1	0.001	0.09	8.7	3.13	126.12	5	118	717	39633	124079
2	0.001	0.03	3	8.93	13.16	5	208	1028	41962	122888
3	0.01	0.12	4.17	9.6	27	17	126	1389	44127	125270
4	0.001	0.13	2.49	5.89	20.75	6	272	1449	42950	124661
5	0.001	0.09	2.25	11.01	7.56	6	221	1630	46323	125272
Me	0.02	1.12	**331.31**	**539.28**	**395.02**	**8**	368	4397	**45517**	131045

■ loss them by large gap
■ win at least one of them

In small testsets, I have similar time consuming with previous heros, while in larger testcases, I lose them by a lot. I think it is because, I did not prune some routes that having no better result.

e. What else did you do to enhance your solution quality or to speed up your program?

- **Initial Partition**

  In my implementation, I tried two ways for initial partition.

```
184     // Large Cell first
185     for (auto i = 0; i < cellsCnt; ++i)
186     {
187         if (isBalance(ASize + Cells[i]->size, true))
188         {
189             Cells[i]->isA = true;
190             for (auto p: Cells[i]->connectNets) Nets[p]->cellsInA++;
191             A.insert(i);
192             ACnt++;
193             ASize += Cells[i]->size;
194         }
195         else
196         {
197             Cells[i]->isA = false;
198             B.insert(i);
199             BCnt++;
200             BSize += Cells[i]->size;
201         }
202     }
```

  First, I tried to evenly split the cells, hence I first sort the cells list by its size, and partition them depends on current total size in A and B.

  We can use a custom comparator to compare in descending order. While the result does not look good.

  The second way is Net first, I tried to move the cells in the same net to the same side. Which is a pretty good idea to sharply reduce the initial cut size.

- **Cell Dictionaries**

  Since the cells' name id not continuously, I use `std::map<int, int>` to create the cell Dictionaries. I save the cell in the `std::vector<cell*>`, I can use Idx to RealName, and RealName to Idx both dictionaries to get the cell pointer. Since it takes only O(1) time to find in dictionary, helping me to implement the code more clearly.

- **getMaxGain**

  Since I implement the bucket list with `std::map`, I can quickly locate the cell with largest gain. Also, it can help me to quickly check if the bucket list is empty and break the iteration.

f. What have you learned from this homework? What problem(s) have you encountered in this homework?

In this homework, I again familiar with the data structure and algorithms, and I enjoy in figuring out the better solution. But I start the homework too late, hence I did not have the satisfied result. I will start early in the next homework.

g. If you implement parallelization, please describe the implementation details and provide some experimental results

N/A