Physical Design PA1: FM Partition

M11215084 吳尚陽

Collaborate: M11215036 許桓菘

Algorithms

Pseudo code

```
/* Initiation of all cell gains */                                // O(P) – P: #pin terminals
  for each $Cell_i$ in all cells                                   // O(c) - c: #cell
    gain($Cell_i$) ← 0
    F ← “From block” of $Cell_i$
    T ← “To block” of $Cell_i$
    for each $Net_n$ on $Cell_i$                                   // O(p) - p: pins for each $Cell_i$
      if F($Net_n$) = 1, then gain($Cell_i$) ← gain($Cell_i$) + 1
      if T($Net_n$) = 0, then gain($Cell_i$) ← gain($Cell_i$) – 1
    end
  end
  // Find $Cell_{MaxGain}$                                         // O(1) - supported by *special data structure
  $Cell_{base}$ ← $Cell_{MaxGain}$
/* Iterate until gain($Cell_{MaxGain}$) ≤ 0 */                     // O(P) for each iteration – P: #pin terminals
  For each $Cell_i$ has one chance to change block                 // O(c) - c: #cell
    // Update gain($Cell_{free}$)
    F ← “From Block” of the $Cell_{base}$
    T ← “To Block” of the $Cell_{base}$
    Lock $Cell_{base}$
    For each $Net_n$ on $Cell_{base}$                              // O(p) - #$Cell_{free}$ reduces by time
      // check all $Cell_i$ on critical nets $Net_n$ before moving $Cell_{base}$
      if T($Net_n$) = 0, then all $Cell_{free}$ on $Net_n$: gain($Cell_{free}$) ← gain($Cell_{free}$) + 1
      else if T($Net_n$) = 1, then $Cell_{free}$ on $Net_n$ only on T side: gain($Cell_{free}$) ← gain($Cell_{free}$) - 1
      // moving $Cell_{base}$ changes F($Net_n$) and T($Net_n$)
      F($Net_n$) ← F($Net_n$) - 1
      T($Net_n$) ← T($Net_n$) + 1
      // check all $Cell_i$ on critical nets $Net_n$ after moving $Cell_{base}$
      if F($Net_n$) = 0, then all $Cell_{free}$ on $Net_n$: gain($Cell_{free}$) ← gain($Cell_{free}$) – 1
      else if F($Net_n$) = 1, then $Cell_{free}$ on $Net_n$ only on F side: gain($Cell_{free}$) ← gain($Cell_{free}$) + 1
    end
    // Find $Cell_{MaxGain}$                                       // O(1) - supported by *special data structure
    $Cell_{base}$ ← $Cell_{MaxGain}$
  end
  // Recover to the best state with least cut size                 // O(P) - recover #c steps in worst case
  Move $Cell_{base}$ back to its origin block
end
```

Data structure

Finding $\boldsymbol{Cell_{MaxGain}}$ and **gain($\boldsymbol{Cell_i}$) ← gain($\boldsymbol{Cell_i}$) ± 1** should be a heavy-loading progress due to the searching of a specific cell in a huge number of cells; However, here we could **optimize the time complexity to O(1) using a red-black tree index pointing to a series of double linked cell nodes**.
First, we decided the indices are gains possessed by all cell nodes. Thus, all the nodes on the same block with same gain are linked with a double linked list structure with an index represents their gain.
Next, we could observe the movements we do to update one random node's gain. To update the gain of a specific node is to first, search, second, remove it from the original place, last, insert it into a new list with updated gain index.
Searching in index in red-black tree we need amortized time complexity of O($\log n$), and luckily, here n should be small since n represent different gains but not nodes. Searching a random node in an unsorted double linked list takes O(n), but here n is ideally small, since we should split all nodes into 2 blocks, then into several gain indices. We could thus see time complexity of searching as constant time.
Removing a node in double linked list takes O(1). Adding back to a new list, first, we need to search for the index, which is constant time, and insert it to the head of the list takes only O(1).
We could see that updating gain in such structure takes only constant time.
As for Finding $Cell_{MaxGain}$, we only have to find the largest gain index. The index will then point to the node with max gain. In this case, red-black tree with small n refer to gain, O($\log n$) is sufficient to find max. Same reason as above, Finding $Cell_{MaxGain}$ would only take constant time.
To sum up, **Finding $\boldsymbol{Cell_{MaxGain}}$** and **updating gain($\boldsymbol{Cell_i}$)** both **takes only O(1)** to accomplish under our data structure.

Time complexity

It would be quit confusing why **O(P) for each iteration** but not **O($\boldsymbol{P^2}$)**. The reason is that to go through the whole process of iterating, nodes would be locked one after another in each round. As the iteration goes on, the size of the input declines. The other reason is that despite the fact of there are four conditions that may lead to checking through all $Cell_i$ on critical nets Net_n, the process happens at most once on one net. That is, only when [F(Net_n), T(Net_n)] are in four states, [0, n], [1, n-1], [n-1, 1], and [n,0] the cell checking process would be launched. A cell that is moved, will be locked afterwards. It will be impossible that a net occurs to the same state twice.
As for the recovery at the end of each iteration. The time complexity is dramatic if we are not using gain to simplify the counting of cut size. Recovering $Cell_{base}$ the status of related Net_n is also necessary. However, dealing with the status recovery for each Net_n takes only O(1), since cut size checking for each net is dismissed. Thus, O(P) is needed if #c, the worst case, steps have to be recovered. Fortunately, the worst case happens only in the last iteration.
As the result, we could say that the overall time complexity is **O(P) for each iteration.**

Results and findings

The strategy of FM partition algorism is to compromise a little size balancing criteria while trying to minimize cut size. We can find out from the algorithm that the "Simulated annealing stopping criteria" is to "stop when accumulative gain $\leq$ 0". However, I want to take a closer look at the "accumulative gain = 0" condition. Would it be better if we trace back to a “no gain, but more balanced” step in the end of every iteration? Thus, I did some revision on the algo and compared the two results.

FM: recover to the first met best cut size in each iteration

performed better

FM_b: occur with same cut sizes, compare balance factor

	FM partition		FM_b partition		Cut size		Iterations	
	PartA	PartB	PartA	PartB	FM	FM_b	FM	FM_b
Input_0	86469	64281	90096	60654	5543	6084	25	24
Input_1	1485	1515	1500	1500	1239	1248	9	9
Input_2	3528	3472	3552	3448	2199	2207	15	11
Input_3	36666	30000	36659	30007	27656	27559	21	23
Input_4	74622	76128	75375	75375	44801	44666	17	45
Input_5	189333	193156	193140	189349	143784	143546	19	26

We could find out some interesting result from the table:

1. It seems that FM_b has better cut size performance over FM on larger datasets:
   On larger dataset, balancing seems to improve the cut size performance. The stricter balance criteria may lead to more iterations needed for SA to terminate. This may somehow help FM_b to earn more chances to try and seek for better solution.

2. On Input_0, FM has a more balanced result than that of FM_b:
   As FM partition is a heuristic algorism that the final result lies heavily on the result of initial partition, I change the way to do the initial partition for this data set and the result is as follows.

	FM partition		FM_b partition		Cut size		Iterations	
	PartA	PartB	PartA	PartB	FM	FM_b	FM	FM_b
Input_0	60314	90436	90395	60355	9432	9053	24	20

   Totally different result shows that it may be some coincident that FM_b met termination earlier due to the order of the cells.

3. After the experiment, I found that initial partition is one of the keys of getting a better result. Thus, I tried random but balanced initial partitioning and found that the results run more iterations. I think that it’s because in our dataset, a net cluster the nodes, and thus, inputParse arranged _cellList in the way that the neighbors are somehow more related. Therefore, randomly choosing sides in initial partitioning may somehow break the order and lead to a longer process of SA.