FPGA final project report

徐祈 112065504

Details of my approach

1. Public testcases results

	Testcase1	Testcase2	Testcase3	Testcase4
Runtime(s)	595	595	595	595
HPWL	11715	10198964	73796	62614459.5
improved	25.391%	8.915%	71.33%	38.878%

2. Data structure

```
class Slot
{
public:
    Slot(){};
    int Rid, stored;
    float x, y;
};

class Instance
{
public:
    Instance(){};
    int Iid, type, rsrc;
    float x, y;
    string name;
    vector<int> net;
};

class Net
{
public:
    Net() {}
    string name;
    vector<int> insts;
};
```

Slot is the resource vector unit, Instance is the instance vector unit, Net is the net vector unit.

3. Algorithm

- Main function

  First, parser will eat all data from input files. Then, I do initial placement, which will generate a legalized initial solution. After that, apply simulated annealing on the initial solution to find a better solution with lower cost (HPWL).

- Initial placement

  I tried 3 types of initial placement methods: the first one randomly picks suitable resource for each instance. The second one traverse all resource for each instance, to find the nearest one. The third one is the smarter second version, it doesn't traverse all resources, but just check its neighbor's resource column. I find out that the first one gives a bad total HPWL, the second one and the third one gives good initial HPWL but the third one uses much less time to find the same solution as second one. Therefore, I chose the third method as my initial placement.

- Simulated annealing

```
1.   void SA()
2.       double reject, reduceRatio = 0.9999;
3.       int nAns, uphill, T = 1000000000, N = 100; // N: # answer in T
4.       bestHPWL = originWL;
5.       do
6.           reject = nAns = uphill = 0.;
7.           while (uphill < N and nAns < 2 * N)
8.               ++nAns;
9.               double deltaCost = perturb();
10.              if (deltaCost <= 0)
11.                  if (deltaCost > 0)
12.                      ++uphill;
13.                  originWL = originWL + deltaCost;
14.                  if (originWL < bestHPWL)
15.                      bestInstances = instances;
16.                      bestHPWL = originWL;
17.              else
18.                  ++reject;
19.                  Swap(oldtype, oldr2, oldr1, true);
20.          T *= reduceRatio;
21.          if(T<15) T = 1000000;
22.      while (T > 10);
```

Above is my simulated annealing (SA in short) pseudo code.

I think there are two different between my SA and traditional SA method. First, I always pick the best solution, means there's no way to go uphill. The reason is that by several experiments, it gives better results. That's why I finally chose to use this purely greedy method.

Secondly, in line 21, I increase the temperature once it's too low. Since I find for most testcases, keeping exploring the solution space usually gives better results, so I set a timer to stop the program before 10 minutes. Before program stop, SA will keep looking for solutions.

- Perturb

  Randomly pick one instance and one resource slot which's type must match the instance. Then swap the instance on that resource slot with the randomly picked instance if that slot is already occupied. Else, if it is empty, just put the instance there.

- Swap

  After swapping, I calculate the net HPWL which affected by swapping only. Because there's no need to re-calculate total HPWL, which waste lots of time. Here is a simple pseudo code showing how I do.

```
beforeSwap = netHPWL(inst1) + netHPWL(inst2);
swapping inst1, inst2
return netHPWL(inst1) + netHPWL(inst2) - beforeSwap;
```

  the return value will be passed back to SA function as the value of deltacost. If it's negative, a better solution is found, record as best solution so far.

4. Lesson learned & takeaways

- SA is not always better than greedy approach.
- Optimizing the bottleneck of SA, HPWL calculation in this case, will improve the result a lot.
- Use timer to stop from timeout.