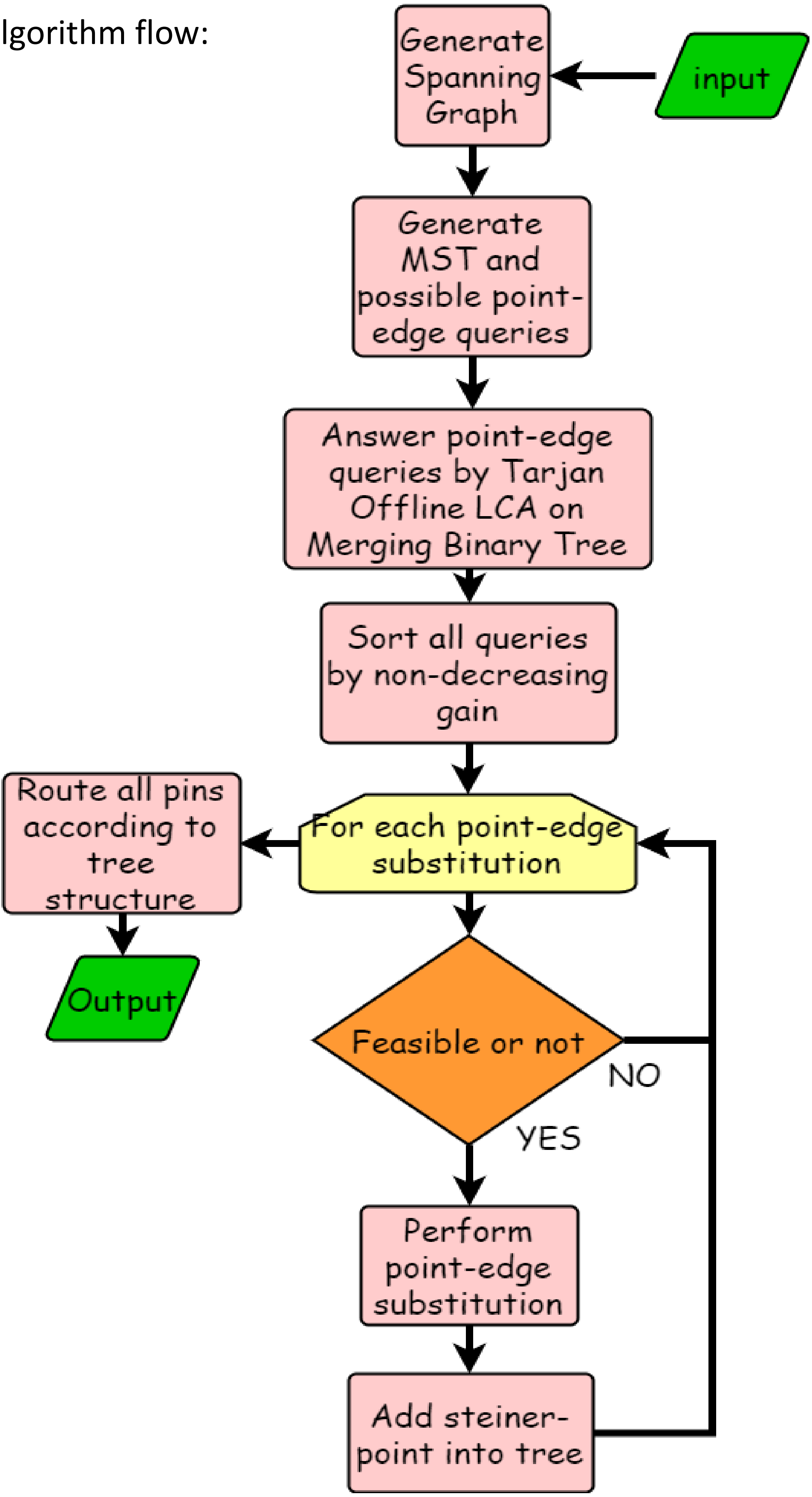
Physical Design for Nanometers ICs, PA 4

B03901056，電機三，孫凡耕

(1) Algorithm flow:


Generate Spanning Graph
input
Generate MST and possible point-edge queries
Answer point-edge queries by Tarjan Offline LCA on Merging Binary Tree
Sort all queries by non-decreasing gain
Route all pins according to tree structure
For each point-edge substitution
Output
Feasible or not
NO
YES
Perform point-edge substitution
Add steiner-point into tree

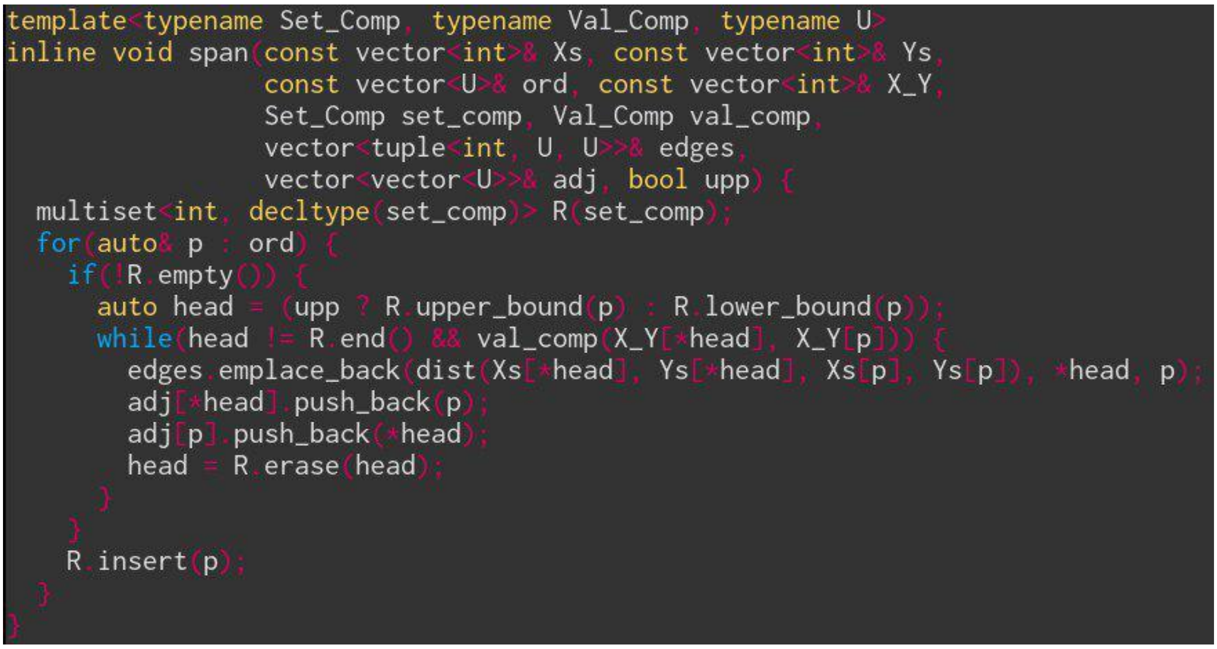
(2) Detailed description:

Generate Spanning Graph → Based on the paper : “Efficient spanning tree construction without delaney triangulation” by H. Zhou, N. Shenoy, and W. Nicholls, which has a complexity of O(n*log(n)).

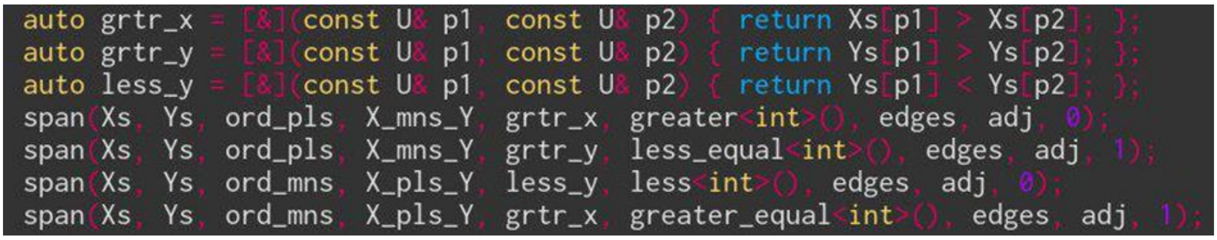
Determine Type→ Since the quantity of the pins is pretty small compare to how large a standard integer can store. Thus for storing pin IDs, char, short or int may be used for different cases.

Dynamic Iterations→ During testing, I observed that a larger test case needs more iterations before convergence. Thus, I determined to let my program runs different numbers of iteration on different size of test cases. The minimum iteration is 2, and it reaches 7 for test cases with more than 200000 pins.

Overall→ My program follows the algorithm in the paper : “Efficient Steiner Tree Construction Based on Spanning Graphs” by Hai Zhou with suitable data structure, template function and optimized routine. For example, the process of generating spanning graph in shown as below (I insert every possible element instead of the shortest because I found that this improves the solution with minimal effects on runtime.

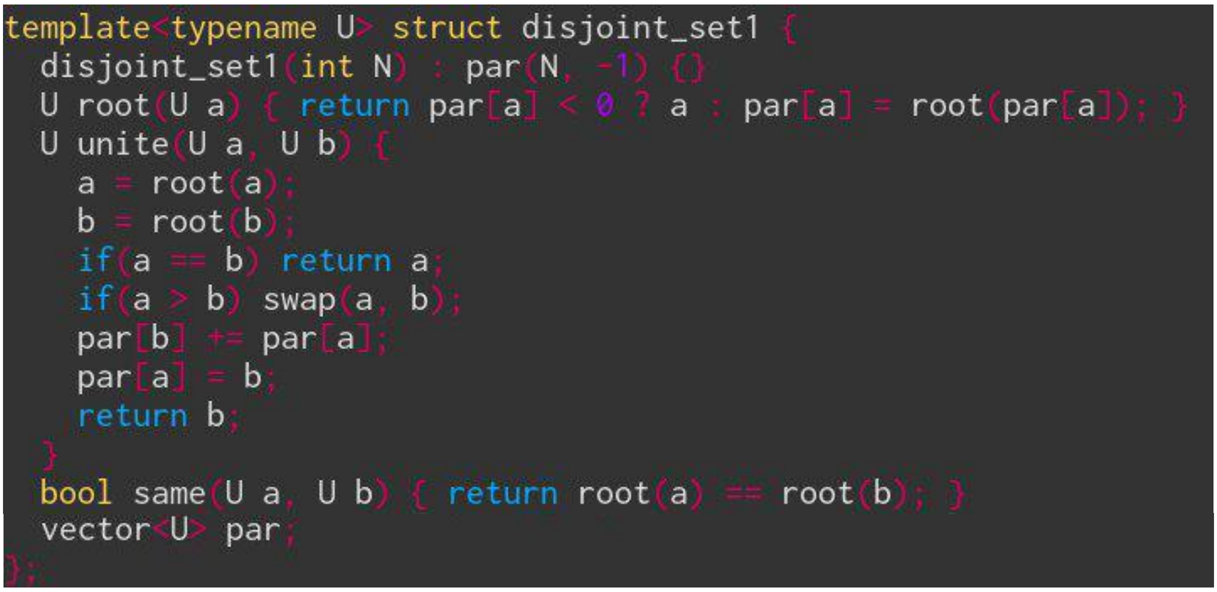


```
template<typename Set_Comp, typename Val_Comp, typename U>
inline void span(const vector<int>& Xs, const vector<int>& Ys,
                 const vector<U>& ord, const vector<int>& X_Y,
                 Set_Comp set_comp, Val_Comp val_comp,
                 vector<tuple<int, U, U>>& edges,
                 vector<vector<U>>& adj, bool upp) {
  multiset<int, decltype(set_comp)> R(set_comp);
  for(auto& p : ord) {
    if(!R.empty()) {
      auto head = (upp ? R.upper_bound(p) : R.lower_bound(p));
      while(head != R.end() && val_comp(X_Y[*head], X_Y[p])) {
        edges.emplace_back(dist(Xs[*head], Ys[*head], Xs[p], Ys[p]), *head, p);
        adj[*head].push_back(p);
        adj[p].push_back(*head);
        head = R.erase(head);
      }
    }
    R.insert(p);
  }
}
```

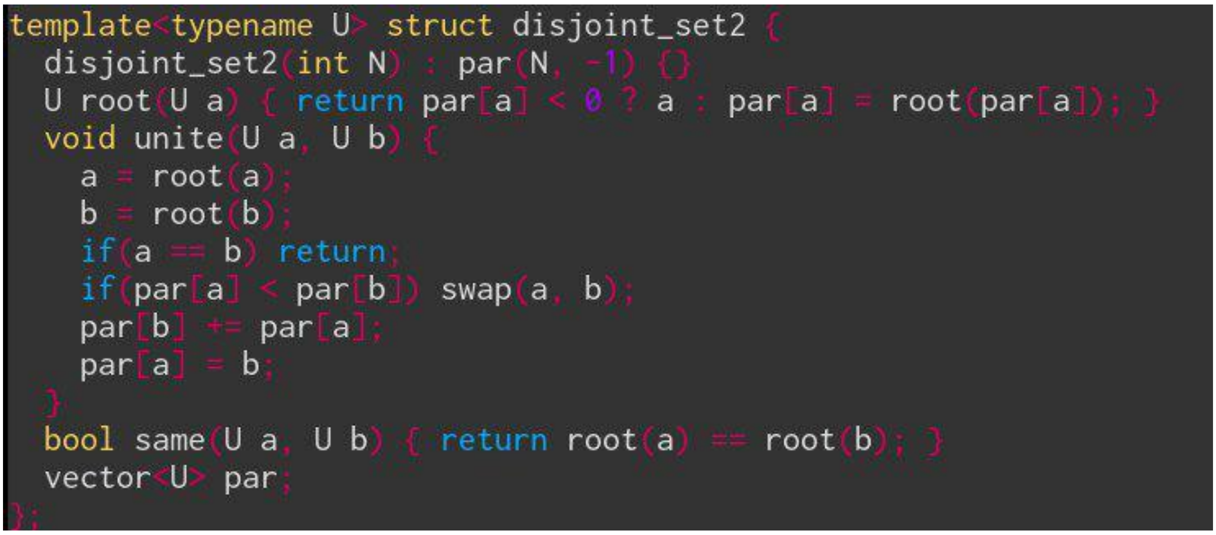


```
auto grtr_x = [&](const U& p1, const U& p2) { return Xs[p1] > Xs[p2]; };
auto grtr_y = [&](const U& p1, const U& p2) { return Ys[p1] > Ys[p2]; };
auto less_y = [&](const U& p1, const U& p2) { return Ys[p1] < Ys[p2]; };
span(Xs, Ys, ord_pls, X_mns_Y, grtr_x, greater<int>(), edges, adj, 0);
span(Xs, Ys, ord_pls, X_mns_Y, grtr_y, less_equal<int>(), edges, adj, 1);
span(Xs, Ys, ord_mns, X_pls_Y, less_y, less<int>(), edges, adj, 0);
span(Xs, Ys, ord_mns, X_pls_Y, grtr_x, greater_equal<int>(), edges, adj, 1);
```

Data Structure:



```
template<typename U> struct disjoint_set1 {
  disjoint_set1(int N) : par(N, -1) {}
  U root(U a) { return par[a] < 0 ? a : par[a] = root(par[a]); }
  U unite(U a, U b) {
    a = root(a);
    b = root(b);
    if(a == b) return a;
    if(a > b) swap(a, b);
    par[b] += par[a];
    par[a] = b;
    return b;
  }
  bool same(U a, U b) { return root(a) == root(b); }
  vector<U> par;
};
```



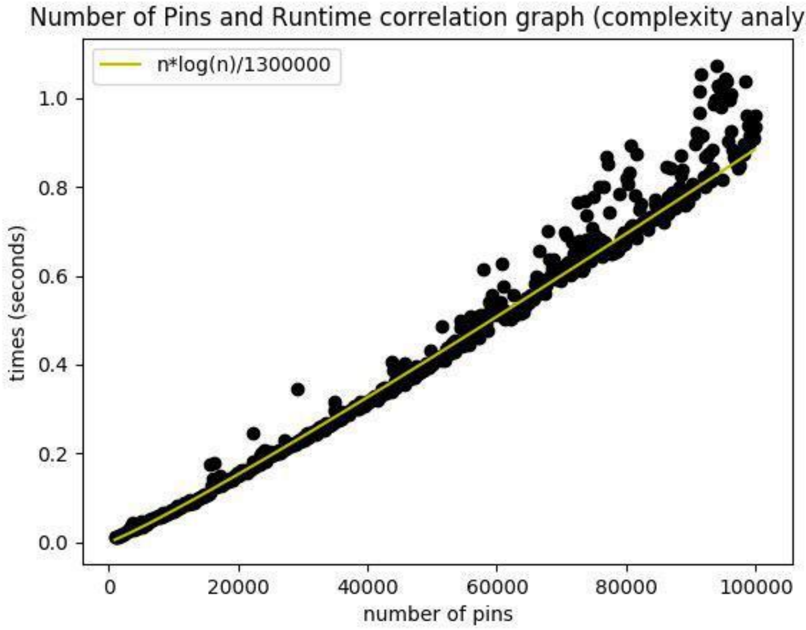
```
template<typename U> struct disjoint_set2 {
  disjoint_set2(int N) : par(N, -1) {}
  U root(U a) { return par[a] < 0 ? a : par[a] = root(par[a]); }
  void unite(U a, U b) {
    a = root(a);
    b = root(b);
    if(a == b) return;
    if(par[a] < par[b]) swap(a, b);
    par[b] += par[a];
    par[a] = b;
  }
  bool same(U a, U b) { return root(a) == root(b); }
  vector<U> par;
};
```

In the whole algorithm, there are two processes that need the help of disjoint set : the first is Kruskal's Algorithm to build minimum spanning tree, and the second is Tarjan's off-line lowest common ancestors algorithm. Disjoint sets are utilized variously in this two operations, thus I use two different variant of disjoint set to optimize my program.

(3) Problem and Discussion:

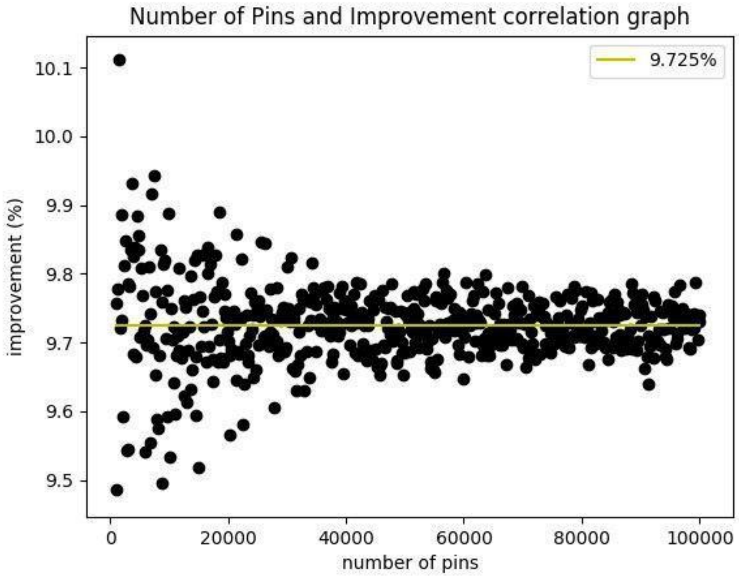
Complexity of my program→

Theoretically, the overall complexity of my program is O(n*log(n)). Experimentally, I tested my program for different sizes of randomly generated test cases. In this experiment, I constrained my program to run only a single iteration instead of dynamically choosing the number of iterations. The results are shown below, where the yellow line indicates the function n*log(n)/1300000. Generally speaking, the programs did follows the yellow line. However, for larger cases, there are higher probability that the programs runs a bit longer than expected. I think this is due to the fact that some times the distribution of the pins effects the numbers of possible point-edge substitution.



Analysis of Improvement→

After several iterations, my programs normally achieves 10.9% of improvement, which is about the same as the paper. Since the first iteration is definitely the most improved and important one, I tested my program on different size of input and visualize the improvement. The results are shown below. We can observed that for smaller cases, the improvements fluctuated more than those larger cases. This is expected since lesser pins implies higher variance of overall topology.


Number of Pins and Improvement correlation graph
improvement (%)
number of pins