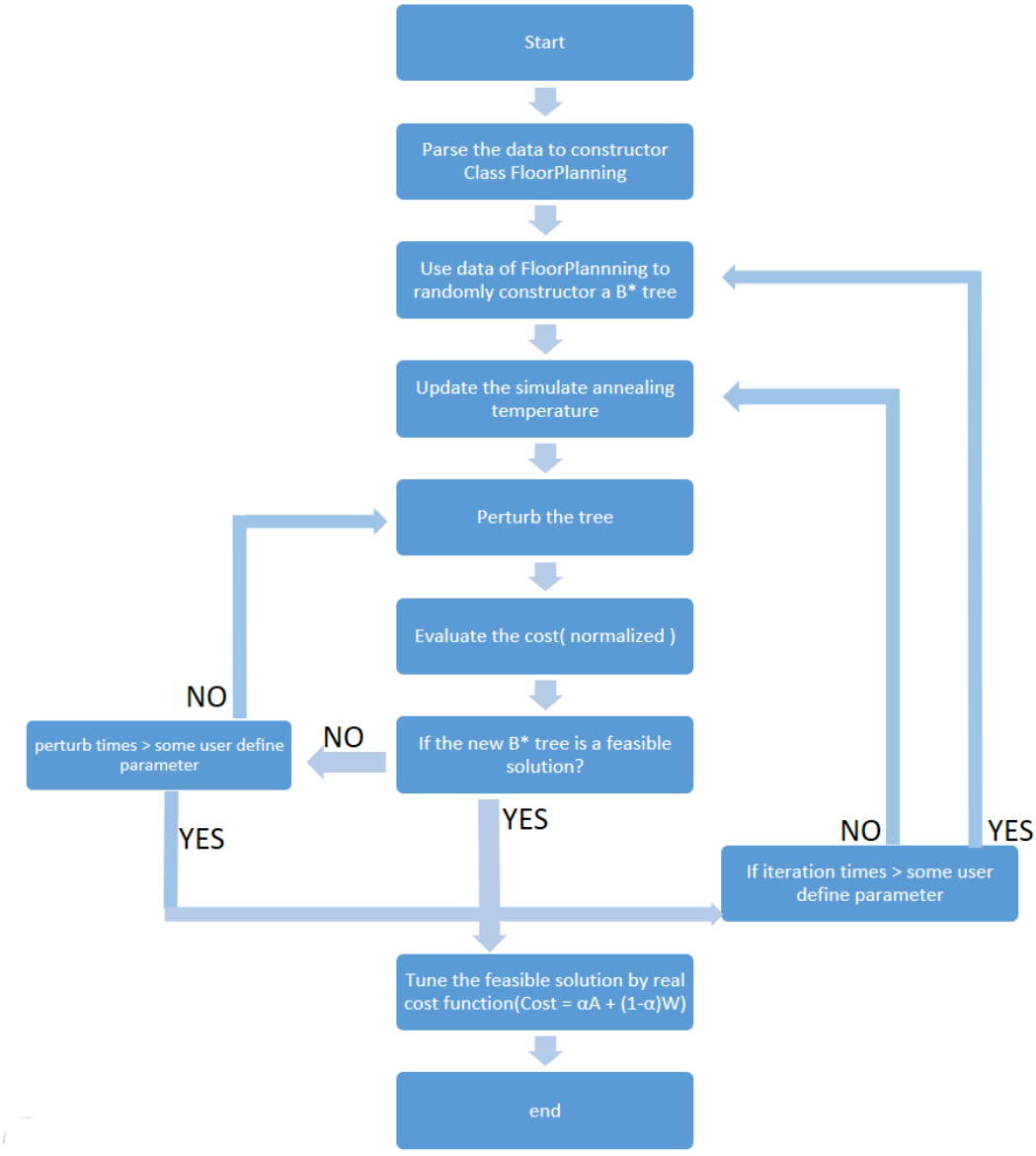
HW2 report

NTUEE

B03901152

陳景由

1. 演算法流程 (Algorithm Flow) (6pt)



演算法流程圖

配合流程圖，將程式運作流程再加上文字敘述如下：

1. 一開始 parse 後建立 FloorPlanning
2. 先隨機的打亂 FloorPlanning 中每個 Block 的次序，再利用 FloorPlanning 建立 B* tree，確保每一次重新建立 B* tree 都是不同的初始狀態。
3. 本次作業是使用 Fast simulated annealing，溫度依照以下規則更新:

$$T_n = \begin{cases} \frac{\Delta_{avg}}{\ln P} & n = 1 \\ \frac{T_1\langle\Delta_{cost}\rangle}{nc} & 2 \le n \le k \\ \frac{T_1\langle\Delta_{cost}\rangle}{n} & n > k. \end{cases}$$

每一次 perturb 完後先確認是否是全部的 block 都落在 outline 之內，若不是則繼續 perterb，直到 perturb 到一定次數後才更新溫度，如果 iteration 足夠多次數後仍無法得到解，就回到 2.重新創造一個 B* tree。

4. 再得到一個 feasible 解之後，就用真正的 cost function(Cost = αA + (1-α)W) 去做最後優化，讓解稍稍變得更佳。
5. 其中 alpha 值是採取 adaptive SA 的實作，因此每次計算 cost 時都會 update。

2. 資料結構 (Data Structure) (6pt)

Data Structure 將分成兩大部分討論:

1.B* tree 2. Fast SA

B* tree part

YContour: 本身是 double link list 結構，代表著一個 y contour 的其中一小段，高度為 y，x 範圍:(x, post->x)，由於會需要常常更新，因此用 insert,delete 複雜度都是 constant 的結構。

```
class YContour
{
    YContour*       pre;
    YContour*       post;
    unsigned        x; // left point of the range
    unsigned        y;
}
```

Node: 為 B* tree 中的 Node,裡面儲存的大多資訊都是 pointer,如果做 swap 只要把兩個 block 的 pointer 互換即可，不用更改整個值，yContour 代表著該 Node 所對應到的 y contour 範圍，只有在建立 tree 時有用，當該 Node 上方被覆蓋滿其他 Node 時，yContour 就不再有存在的意義。

```
class Node
{
    Block*          block;
    Node*           child[2]; // child[0] is left child, child[1] is right child
    Node*           parent;
    YContour*       yContour;
};
```

perturbAction: 這是一個比較特殊的結構，我將每次 perturbAction 記錄成一個物件，如果做 FSA 時結果是不接受一個 cost 較高的結果，就必須要 undo 該次 perturb，因此可以直接呼叫 B* tree 中紀錄的 perturbAction 中的 member function: undo()，藉由這個方式，程式將具有較大的可變性和可維護性，我可以將 perturb 儲存在未來使用，或是 debug 時也可以重新使用某一個 pertrubAction。Perturb 有三個種類，每一種都會儲存 undo 時所需要的全部資訊。

```
class perturbAction {
    void undo();
    // for type1
    Node*           swapNode1;
    Node*           swapNode2;

    // for type2, move bLeaf as aLeaf'new leaf
    vector<Node>    Nodes;
    vector<Node*>   NodePointers;

    // for type3
    Node*           rotateNode;
    unsigned        type;// 1 for swap ,2 for move, 3 for rotate
    BStarTree*      tree;
};
```

BStarTree: 包含所有對於 B* tree 的操作，三種 pertrub，呼叫時會自己產生亂數隨機選擇 perturb 種類以及目標的 block，會使用 FSA 來呼叫 BStarTree 中的 function，並且可以計算 Cost，以及實作 adaptive simulated annealing 的更新 alpha

值。

```
BStarTree
 {
   public:
     BStarTree();
     ~BStarTree() {}
     void             randonConstructTree();
     void             recursiveBuildTree( Node* );
     void             randomRotate();
     void             randomMove();
     void             move( Node*, Node* );
     void             randomSwap();
     void             swap( Node*, Node* );
     void             rotate( Node* );
     void             printBST();
     double           cost();
     unsigned         calcArea();
     unsigned         calcHPWL();
     void             randomAddAsChild( Node*, Node* );
     double           runningCost();
     double           realCost();
     void             updateIsFeasibleQueue();


     double           alphaBase;
     double           alpha;
     double           beta;
     Node*            root;
     // FSA*             fsa;
     vector<Node*>    Nodes;
     YContour*        yContourBegin;
     perturbAction*   action;
     double           runningAlpha;
     unsigned         area;
     unsigned         HPWL;
     unsigned         MIN_W;
     unsigned         MIN_H;
```

```
    unsigned        Anorm;
    unsigned        Wnorm;
    unsigned        feasibleSol;
    bool            isFeasible;
    queue<bool>     isFeasibleQueue;
    unsigned        nTrueInQueue;
};
```

FSA

FloorPlanning 中儲存了所有需要建成一個 B* tree 的資訊。

```
FloorPlanning
{
    unsigned          netID;
    vector<Terminal*> terminals;
    bool              isNet;
};


class Terminal
{
    vector<Net*>      nets;
    string            name;
    bool              isTerminal;
};


class Block : public Terminal
{
    unsigned          W;
    unsigned          H;
};

class FloorPlanning
{
    unsigned          W;
    unsigned          H;
    unsigned          nBlocks;
```

```
    unsigned          nTerminals;
    unsigned          nNets;
    vector<Block*>    Blocks;
    vector<Net*>      Nets;
    vector<Terminal*> Terminals;
};
```

FSA

包含所有 FSA 的操作，更新溫度，紀錄 best case，輸出檔案，以及具有可以呼叫 B* tree 中 perturb 行為的 function，還有 user define 的參數也會在這裡做處理。

```
class FSA
{
  public:
    FSA( BStarTree* t )
    {
      tree = t;
      init();
      totalIter = 0;
    }
    ~FSA(){}
    void PseudoGreedyLocalSearch();
    void HillClimbingSearch();
    void HighTempratureRandomSearch();
    void SA();
    void perturb();
    void randomPerturb( bool );
    void init();
    void output( string& );
    void setStartTime( clock_t t ) { startTime = t; };
  private:
    BStarTree*        tree;
    vector<Node*>     bestTreeNodesPointer;
    vector<Node>      bestTreeNodes;
    vector<unsigned>  bestBlockW;
    double            bestCost;
    double            temp; // temperature
    double            feasibleCounter;
    unsigned          iter; // iteration times
```

```
    unsigned          totalIter; // iteration times
    // unsigned          realCostNorm;
    double            c; // User-specified parameters, for update temperature
    double            k; // User-specified parameters, for update temperature
    double            prob; // uphill climb probability
    double            totalCost;
    double            T1;
    double            newCost;
    double            originCost;
    double            initCost;
    double            totalDeltaCost;
    double            costAfterPerturb;
    double            costBeforePerturb;
    clock_t           startTime;
    unsigned          bestIter;
};
```

3. 問題與討論 (8pt)

y contour**:**

B* tree 的實作並不是完全的 linear time，在某一部分會取決於 y contour 的數量，因此每一次加入新的 block 時，都要確認新的 block 將會覆蓋到那些 y contour，並將不再使用的 y contour 移除。

Perturb:

有三種 operation，swap, rotate, move。

1. Swap:
   在 perturbAction 之中紀錄被 swap 的兩個 block, undo 的時後再將兩個 block 互相交換一次即可。
2. Rotate:
   紀錄 rotate 的 block,undo 時再 rotate 一次。
3. Move:
   這個動作和一般的 binary tree movement 不一樣，我做 perturb 希望 tree 有連續性的小幅度改變，因此不能像平常做 movement 一樣將要移動的 node 的 predecessor 和自己做交換，而是要一層一層的隨機把 child 往上拉，對應到 B* tree 的 Floor planning 就會變成某個 block 移走，而旁邊的 block 則因為缺少一個 block 而層層遞補上來，這樣才不會造成過多的變動。而因為層層遞補的關係，如果實作每一步的 undo 可能會讓 operation 次數過多，因此直接把所有 Node 的資訊 Copy 下來，undo 時直接貼上，也省下寫 code 與 debug 的時間。

Cost function : $\Phi(F) = \alpha A + \beta W + (1 - \alpha - \beta)(R - R^*)^2$

如果實作 adaptive SA，要考慮到 alpha 值和(1-alpha-beta)是不是都落在(0,1)內，因此我將 cost funciton 改成:Cost = alpha*A + beta* W + (1-alpha)(1-beta)(R-R*)^2，確保面積比的 penalty 不會消失。

FSA:

每個溫度下的 iteration 設置是 5*(# of block)，不能夠太多，因為超過五倍太多就會使 SA 達到一個動態平衡，溫度不夠低也無法穩定存在一個很好的解。如果太少次又不夠達到平衡，所以就是一直條參數直到時間大致上能夠到達一個最小值。而 iteration 次數的上限也很重要，如果太少次很難達到平衡，如果太多次但是又陷在某一個 local minimun 的話又會花太多時間，也是必須一直嘗試才能找到最好的情況。最後一個是初始溫度要求除以 ln(P)，對於 ami33 的測資來說，不好的初始溫度(與最後試出來的最好的初始溫度僅差兩倍)可以造成程式執行時間差 2~5 倍。

Calculate HPWL:

這個 operation 非常的花時間，如果說很在意效率的話，可以放棄一些 cost，在 SA 進行時只計算 cost = alpha*A + (a-alpha)*(R-R*)^2，也可以達到一個可行解，而在有提供的測資中，可行解的 cost 差異並不會到非常巨大，因此不計算 HPWL，僅在得到 feasible sol 之後用 real cost 去進行 tunning 也不失為一個方法，效率大概可以達到 2.5 倍左右。