1. Introduction

In this project, I implement a floorplanner which can place a set of hard blocks of a net-list with fixed sizes in a way that it will optimize the total area and wire-length with a reasonable runtime. The floorplanner is based on the B*-tree data structure [1] and the simulated annealing optimization algorithm. 6 benchmarks of block sizes ranging from 10 to 300 have been used to evaluate the performance of the floorplanner.

2. Problem Formulation

The objective of the floorplanner is to produce a position for each block so as to minimize both the total area and total wire-length, with the constraints that there exists no overlap between blocks and terminals are placed along the edge of the chip with minimum pitch distance s=2. The optimization problem can be formulated as follows.

minimize $alpha * Area + (1 - alpha) * Wirelength$
s.t. (i) No overlap between blocks;
(ii) Terminals are located along the edge of the chip
(iii) The minimum distance between terminals is s=2

where alpha ($0 \leq alpha \leq 1$) is the area weight in the cost function, Area is the total area of a floorplanning result which is the smallest rectangle that encloses all blocks, Wirelength is the total wire-length of a floorplanning result which is the sum of the half perimeter wire length of each net.

3. B*-tree data structure

In this project, the representation of a floorplanning solution is based on the B*-tree data structure [1]. B*-tree is an ordered binary tree. An example showing the correspondence is illustrated in Figure 1.

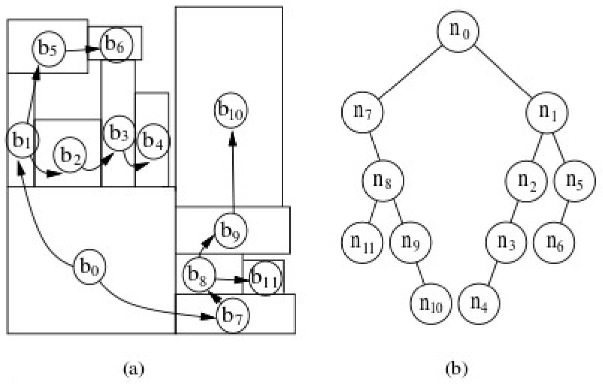


Figure 1: (a) An floorplanning . (b) The B*-tree representing the floorplanning

The B*-tree keeps the geometric relationship between two modules as follows.
(1) The root corresponds to the block on the bottom-left corner, $(x_0, y_0) = (0, 0)$.
(2) If node n_j is the left child of node n_i, block b_j must be located on the right-hand side and adjacent to block b_j, i.e., $x_j = x_i + w_i$.
(3) If node n_j is the right child of n_i, block b_j must be located above and adjacent to block b_i, with x-coordinate of b_j equal to that of b_i, i.e., $x_j = x_i$.
(4) The y-coordinate of a block is determined based on a contour data structure presented in [2]. The contour structure is a doubly linked list which stores the horizontal contour curve for current compact floorplanning, as shown in Figure 2. For example, when we insert a node n_i, since the x-coordinate x_i of the node n_i has been determined by (2) or (3), we can search the linked list until reaching a node n_j in the contour linked list which covers x_i. Then, the y-coordinate $y_i = y_j+h_j$, and a new contour is updated with the top boundary of the new block n_i.

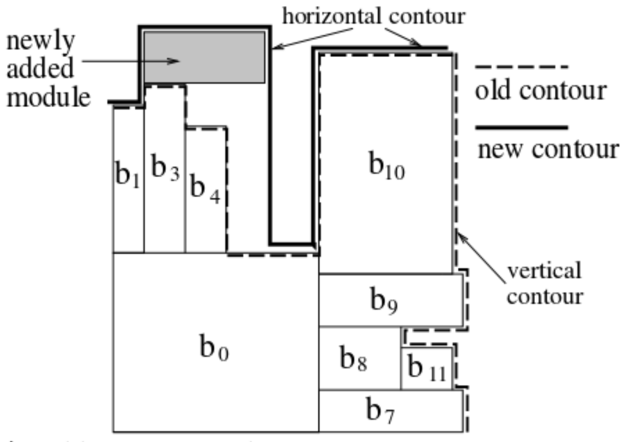


Figure 2: When inserting a new block, the x-coordinate is firstly determined by the geometry relationship (2) or (3), and then the y-coordinate is determined by searching the horizontal contour from left to right. After that, a new contour is updated with the top boundary of the new module.

The correspondence between a compact floorplanning and its induced B*-tree is 1-to-1 [1]. In other words, for a compact floorplanning, we can construct a unique B*-tree, and vice versa. The 1-to-1 correspondence prevents the enlarging of search space with duplicate solutions. Moreover, the B*-tree can perform basis tree operations such as search, insertion, deletion in only O(1), O(1) and O(n) time, where n is the number of blocks, and the transformation between a compact floorplanning and its induced B*-tree takes only O(n) time.

4. Simulated annealing

Since the floorplanning problem is a NP-hard problem, a simulated annealing (SA) optimization algorithm is used to obtain a good solution instead of the best solution with a reasonable runtime.

The basic structure of the simulated annealing algorithm is shown as follows.

Algorithm Simulated-annealing based floorplanning
Input:
(i) Hard blocks B with fixed area and shapes, terminals T, and its net-list NETLIST
(ii) SA parameters: init-temp, term-temp, conv-rate, k, alpha
Output:
positions of each block, total area, total wire-length, runtime

```
temp = init-temp;
btree = RAND-BTREE(B);   //build initial btree
fp = PACK(btree);      //generate a floorplanning
while temp > term-temp && rej-rate < conv-rate do          //rej-rate = #rejection / (K*N)
        for int i=0 to K*N do
                new_btree = PERTURB(btree);        //perturb the btree
                new_fp = PACK(new_btree);           //generate a new floorplanning
                cost_diff = COST(new_fp) – COST(fp);
                if cost_diff < 0 do      //find better solution
                        btree = new_btree;
                        fp = new_fp;
                        best_fp = new_fp;
                else if RANDOM(0,1) < exp(-cost_diff/temp) do
                        btree = new_btree;       //accept bad solution with probability based on temp
                        fp = new _fp;
                endif
        endfor
        temp = SCHEDULE(temp);  //update temp
endwhile
```

4.1 Cost function

The cost function COST() used in the SA algorithm is as follows:

$$alpha * Area + (1 - alpha) * Wirelength$$

where alpha is the area weight between 0 to 1, Area, Wirelength are the total area and wire-length of currect floorplanning solution. Experiments show that alpha = 0.3 can produce good solutions.

4.2 Perturbation

Two perturbations PERTURB() are implemented to generate a new b*-tree as well as a new floorplanning:

(1) moving one random node x to be another random node y's child (delete node x, insert x after y)

(2) swap two random nodes x and y (update the pointers for nodes x, y, their parents and children)

The probabilities for move and swap are equal (p[swap] = p[move] = 0.5). For each temperature before termination condition is satisfied, the SA algorihtm will perturb K*N times (K = 1000 for small benchmarks or 100 for large benchmarks).

4.3 Temperature scheduling and termination condition

A simple temperature scheduling function SCHEDULE() is used in this project: temp = 0.85 * temp. The initial temperature is set to 1000, and the termination temperature is set to 0.1.

When the rejection rate is larger than the pre-defined termination rate (conv-rate = 0.99) or when the temperature is less than the termination temperature (term-temp = 0.1), the program will terminate.

4.4 Terminal assignment

The function PACK() not only generates the position for each block, but also generates position for each terminal based on two given terminal constraints. In this project, a greedy assignment algorithm is used to generate the position for each terminal along the chip edge with minimum pitch s=2. The terminal assignment method is illustrated in Figure 3.

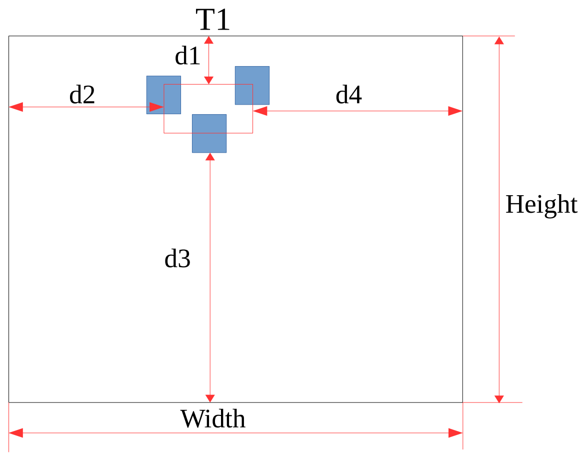


Figure 3: Terminal assignment

The terminal assignment consists of two steps.

The first step is to calculate the closest the edge for the terminals. For example, in the net of terminal T1 (without considering T1), we can calculate the minimum and maximum x-coordinates (x_{min}, x_{max}) and y-coordinates (y_{min}, y_{max}) for the three blocks that are connected to T1. Then, four distances can be calculated: d1 = Height – y_{max}; d2 = x_{min}; d3 = y_{min}; d4 = Width – x_{max}; After that, T1 is assigned to the closest edge with minimum distance among d1 to d4. If T1 is assigned to the horizontal edges (y_{T1}=0 or y_{T1}=Height), its x-coordinate would be x_{T1} = (x_{min} + x_{max})/2; The similar rule can be applied to the case when T1 is assigned to the vertical edges. The first steps takes O(t) time for t terminals since each net has only very few number of blocks.

The second step is to check the overlap among terminals. Each terminal Ti is check with other terminals to ensure that there is not overlap. If overlap is found for Ti, then Ti is moved clock-wise with unit distance along the chip edge. The overlap checking is continued until no overlap exists. The second step takes O(t^2) time on average for t terminals.

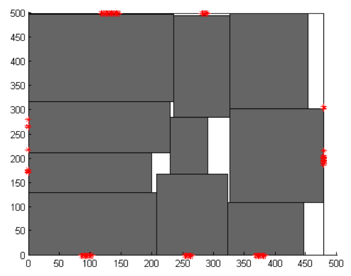
5. Experiments and Results:

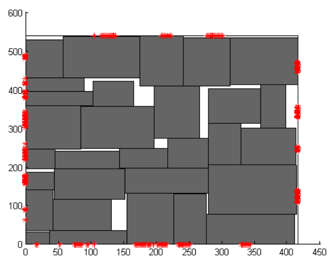
The floorplanner is implemented using C++. Six benchmarks are used to evaluate the performance of the floorplanner. The simulation is run on linux.glue machine at the University of Maryland. The floorplanning results on 6 benchmarks are presented in Table 1. The final parameter settings for SA algorithm are shown in the table, along with three performance metrics: total area, total wirelength (WL) and runtime. All floorplanning results pass the MATLAB script.

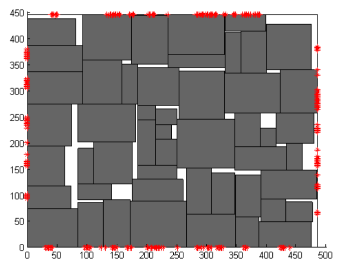
Benchmark	Alpha	Init. Temp	Term. Temp	Conv. Rate	K	Area	WL	Runtime(s)
B10	0.3	1000	100	0.99	1000	237566	19275.5	18.42
B30	0.3	1000	0.1	0.99	1000	223270	55706.5	452.6
B50	0.3	1000	0.1	0.99	1000	215340	93819.5	467.49
B100	0.3	1000	0.1	0.99	1000	202275	139103	2971.93
B200	0.3	1000	0.1	0.99	100	207405	306297	4915.34
B300	0.3	1000	0.1	0.99	100	326040	446736.75	4222.4

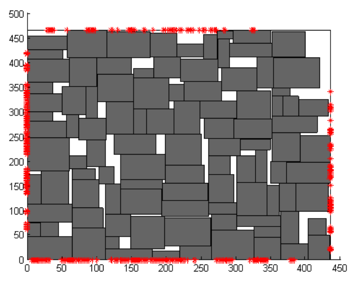
Table 1: The floorplanning results of six benchmarks. Alpha is the area weight in the cost function; Init. Temp and Term. Temp are initial temperature and termination temperature; Conv. Rate is the termination threshold for rejection rate. K is a multiplier which controls the number of moves for each temperature during simulated annealing (K*N = # moves, N = # blocks).

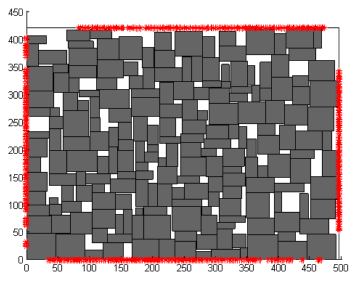
The plots of six floorplanning solutions are illustrated in Figure 4.


(a) B10


(b) B30


(c) B50


(d) B100


(e) B200

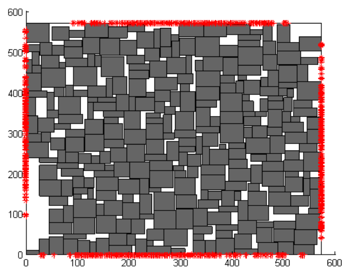

(f) B300

Figure 4. Floorplanning solution plots

6. Discussion

6.1 The SA parameters settings

In general, a longer runtime for SA algorithm will generate a better solution. To increase the runtime, we can change the parameters such as init. temp, term. temp, conv. rate and K. The observation of changing these parameters are discussed as follows:

(1) Increase Init. Temp: the SA with a high init. temp starts with higher probability of accepting bad solutions, which behaves like random walking from one solution to another and thus can avoid stuck at local minimal. However, my observation based on experiments is that the increase of init. Temp will increase the runtime but will not always give a better solution.

(2) Reduce the term. Temp: the SA will run a bit longer but it will stop when the rejection rate is larger than the conv. Rate.

(3) Increase the conv. Rate: this will increase the SA search time but will stop if the temperature condition is violated.

(4) Increase the multiplier K: better solution will be found at a higher probability since we search more times at each temperature. However, it will require a longer run time (K times run time in worst case).

Among all possible settings, I find that increasing conv. Rate and the multiplier K can have a higher probability of producing a better solution than the change of the other two parameters.

6.2 The area wight alpha

I have tried three different setting of area weight (alpha = 0.3, 0.5, 0.7) for 5 benchmarks. Although increasing alpha will decrease the area (on average 1% to 3%), it will increase wire-length more seriously (on average 5% to 14%). Thus I choose alpha = 0.3.

	Alpha = 0.3		Alpha = 0.5		Alpha = 0.7	
Benchmark	Area	WL	Area	WL	Area	WL
B10	1	1	-0.01	0.07	-0.01	0.15
B30	1	1	-0.01	0.06	-0.01	0.09
B50	1	1	-0.01	0.02	-0.03	0.09
B100	1	1	-0.01	0.06	-0.04	0.24
B200	1	1	-0.03	0.06	-0.06	0.16
Average	1	1	-0.014	0.054	-0.03	0.146

Table 2: The impact of area weight alpha

7. Reference

[1] Chang, Yun-Chih, et al. "B*-Trees: a new representation for non-slicing floorplans." *Proceedings of*

the 37th Annual Design Automation Conference. ACM, 2000.
[2] Guo, Pei-Ning, Chung-Kuan Cheng, and Takeshi Yoshimura. "An O-tree representation of non-slicing floorplan and its applications." *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*. ACM, 1999.