EE6094 CAD for VLSI Design

Programming Assignment 3: Floorplanning

Student ID: 108501023
Student Name: 李品賢

Compile, Execute and Verification

1. Pull the source code, i.e., *108501023_PA3.cpp*, *Makefile*, *t10.txt*, *t20.txt*, *t500.txt* and *checker* into the workstation folder.

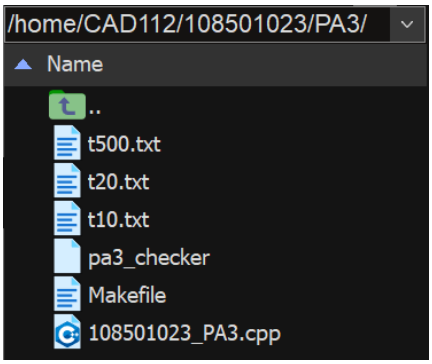

Fig. 1

2. Use *Makefile* as a trigger point to run the *108501023_PA3.cpp* program, and then the output *t10_out.txt* / *t20_out.txt* / *t500_out.txt* are generated.

```
[s108501023@eda359_forclass ~/PA3]$ make all
[s108501023@eda359_forclass ~/PA3]$ make run input=t10.txt output=t10_out.txt
[s108501023@eda359_forclass ~/PA3]$ make run input=t20.txt output=t20_out.txt
[s108501023@eda359_forclass ~/PA3]$ make run input=t500.txt output=t500_out.txt
[s108501023@eda359_forclass ~/PA3]$ make claen
```

Fig. 2

- make all
- make run intput=t10.txt output=t10_out.txt
- make run intput=t20.txt output=t20_out.txt
- make run intput=t500.txt output=t500_out.txt
- make clean

3. Use checker to check whether output files fits the standard output format.

- ./checker t10.txt t10_out.txt
- ./checker t20.txt t20_out.txt
- ./checker t500.txt t500_out.txt

Completion

All three cases are successfully passed the checker, the screen shows three happy “Pepe the Frog”. The following three figure (Fig. 3, Fig. 4, Fig. 5) are the results.

	t10.txt	t20.txt	t500
Completion	O	O	O
Area	57323.05	89318.63	2636155.31

Fig. 3

Fig. 4

Fig. 5

Data structure

I use slicing tree and the corresponding polish expression to represent a single slicing floorplan. Take Fig. 6 as an example, the left-hand side is a slicing floorplan, and the right-hand side is its slicing tree. Additionally, we can do the postorder traversal on the tree, and then we get the sequence 21H67V45VH3HV which is the polish expression of this slicing tree.

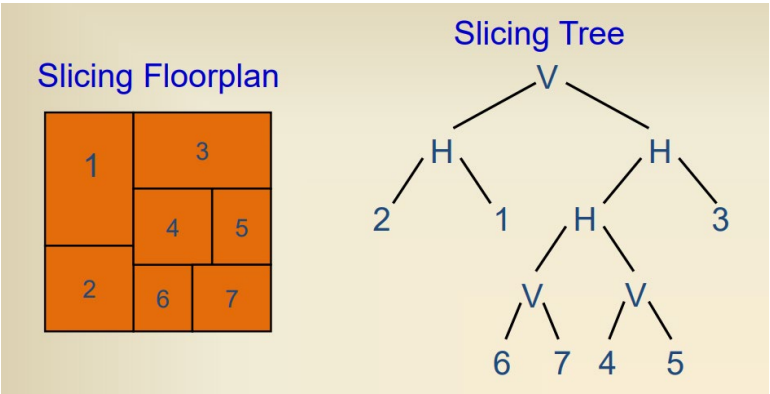

Fig. 6

Algorithm

The basic algorithm I used is Wong-Liu Algorithm (Fig. 7), and it's also a simulated annealing based algorithm. The thesis indicates how to choose the initial temperature and how to set the parameters. Also, the most important part is that it gives three types of move changing current state to its neighborhood state.

In my program, I take a pancake shape as an initial condition. Also, if the single SA have done before 2 hours, it would start another run of SA again and again until the time is up. The newly started SA take the best polish expression as an initial condition.

For the bottom-up procedure, I use Stockmeyer Algorithm (Fig. 8), the time complexity is O(m+n) which is extremely fast. It improves run time a lot.

```
1 begin
 2 E ← 12V3V4V ... nV; /* initial solution */
 3 Best ← E; T0 ← Δavg / ln(P) ; M ← MT ← uphill ← 0; N = kn;
 4 repeat
 5   MT ← uphill ← reject ← 0;
 6   repeat
 7     SelectMove(M);
 8     Case M of
 9     M1: Select two adjacent operands ei and ej; NE ← Swap(E, ei, ej);
10     M2: Select a nonzero length chain C; NE ← Complement(E, C);
11     M3: { done ← FALSE;
12       while not (done) do
13           Select two adjacent operand ei and operator ei+1;
14           if (ei-1 ≠ ei+1) and (2 Ni+1 < i) then done ← TRUE;
13'          Select two adjacent operator ei and operand ei+1;
14'          if (ei ≠ ei+2) then done ← TRUE;
15       NE ← Swap(E, ei, ei+1); }
16   MT ← MT+1; Δcost ← cost(NE) - cost(E);
17   if (Δcost ≤ 0) or (Random <  e^(-Δcost/T)
18    then
19       if (Δcost > 0) then uphill ← uphill + 1;
20       E ← NE;
21       if cost(E) < cost(best) then best ← E;
22     else reject ← reject + 1;
23   until (uphill > N) or (MT > 2N);
24   T ← rT; /* reduce temperature */
25 until (reject/MT > 0.95) or (T < ε) or OutOfTime;
26 end
```

Andy Yu-Guang Chen

Fig. 7

```
Procedure Vertical_Node_Sizing
Input: Sorted lists L = {(a1, b1), ... , (as,bs)}, R = {(x1, y1), ... , (xt, yt)},
       where ai < aj, bi > bj, xi < xj, yi > yj (for all i < j)
Output: A sorted list H = {(c1, d1), ... , (cu,du)},
       where u ≤ s + t - 1, ci < cj, di > dj (for all i < j)
Begin
   H := Ø
   i := 1, j := 1, k = 1
   while (i ≤ s) and (j ≤ t) do
       (ck, dk) := (ai + xj, max(bi, yj))
       H := H U {(ck, dk)}
       k := k + 1
       if max(bi, yj) = bi then i := i + 1
       if max(bi, yj) = yj then j := j + 1
```

Fig. 8

Flow Chart

1. Overall program

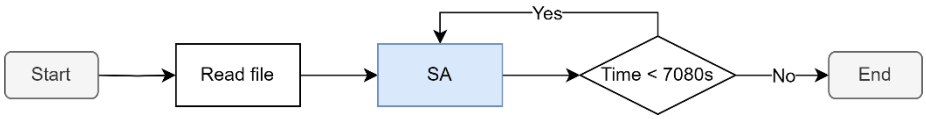

Fig. 9

2. Simulated annealing procedure

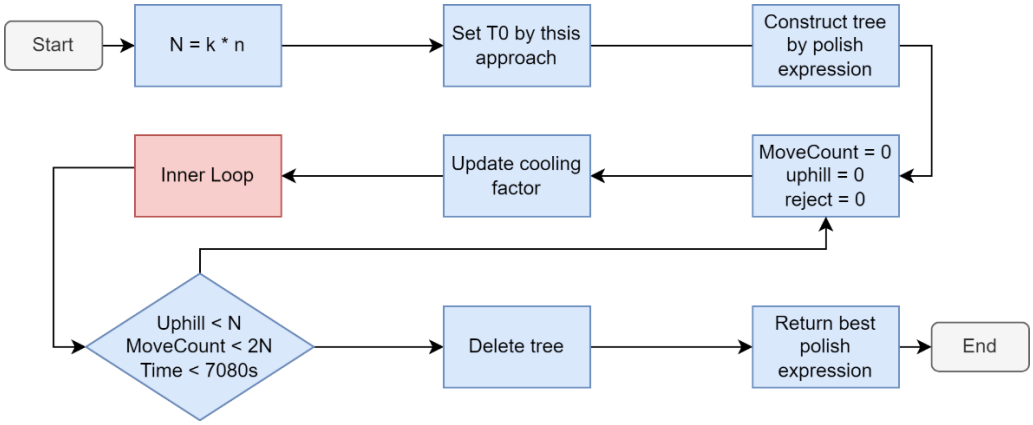

Fig. 10

3. Inner loop of simulated annealing

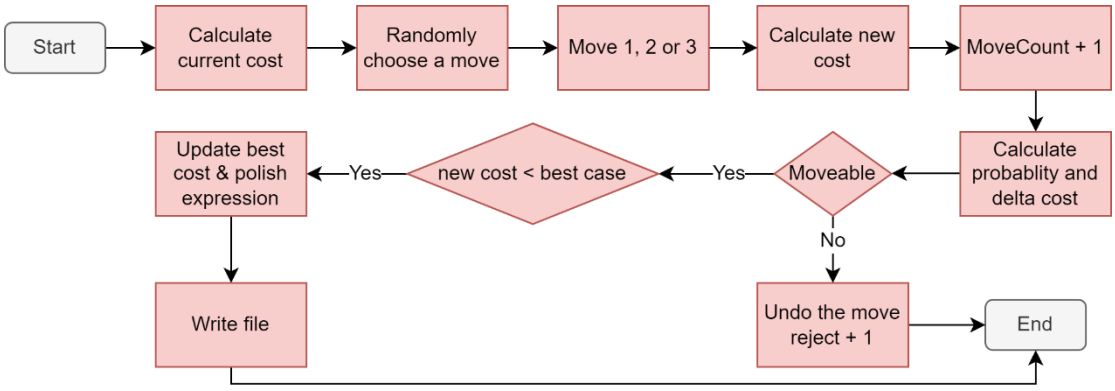

Fig. 11

Data structure in program

1. **class Size** – indicates the aspect of an area

Data type	Name	Purpose
double	width	records the width of an area
double	height	records the height of an area

2. **class Module** – records the area and the final selected aspect of an input module

Data type	Name	Purpose
double	area	the area of input module
Size	aspect	the final selected aspect of an input module

3. **class Block** – records one possibility of the aspect combining the left-child aspect and the right-child aspect, and also their index

Data type	Name	Purpose
int	Lindex	the index of the selected left-child aspect in this block
int	Rindex	the index of the selected right-child aspect in this block
Size	aspect	the aspect combining the left-child aspect and the right-child aspect

4. **class Node** – a tree node including pointers, the slicing information and all possibilities of combined aspects

Data type	Name	Purpose
class Node *	left	points to its left child
class Node *	right	points to its right child
class Node *	parent	points to its parent
string	data	the information of slicing tree, either is a character “V” and “H” or a module number
bool	update	false indicates need to update; true indicates no need to update
vector<Block>	blks	all possibilities of combined aspects

Important variables

1. In class Floorplanning

Data type	Name	Purpose
int	moduleCount	the number of modules
double	total_area	sum of the area of input modules
double	global_cost	best cost between all SA results
int	pts	the number of points picked up in shape curve
double	iter_T	the parameter k of the equation N = kn in the thesis
Node *	root	points to the root of a slicing tree
Node *	avail	points to the available node that can be reused
vector<Module>	module	the information of input module
vector<string>	SPE	the polish expression of a slicing floorplan

2. In SA function

Data type	Name	Purpose
vector<string>	best_SPE	the polish expression leading to the smaller area in a single SA
Node *	temp_root	points to a root of temporary tree especially for move 3
double	T0	the initial temperature
double	T	the current temperature
double	w	the cooling factor
double	curr_cost	the cost of the current state of a floorplan
double	new_cost	the cost of the neighborhood state of a floorplan
double	delta_cost	the difference between the current cost and the new cost
double	best_cost	the smallest cost in a single SA
int	Move_type	the integer type ranging from 1 to 3
int	MoveCount	the move count in a specific temperature
int	uphill	the number of times accepting worse solution
int	reject	the number of times rejecting worse solution

Important functions

(the list ignores the Floorplanning:: mark and the arguments of functions)

1. the functions used in SA

1.	double init_temp()
	moves 50 times to get the average cost difference of uphill, and divides this value by ln(P), where P is near 1. T0 = ave/ln(P), ave = (uphill cost-current cost)/uphill times
2.	Node * SPE_to_tree()
	turns a polish expression into slicing tree
3.	void bottom_up()
	uses postorder traversal to combine the left-child aspect and the right-child aspect, records indexes of them, eliminates unnecessary combinations, and also sets the limit of the combination number by picking up the smaller area.
4.	double update_cooling_factor()
	w is 0.85 for T above 20, while w is 0.9 for T under 20
5.	double calculate_cost()
	the cost is normalized by the total input area and multiplied by 100
6.	int select_move()
	randomly picks up a move ranging from 1 to 3
7.	bool is_movable()
	if the cost difference is negative or the cost difference is positive but its random number is lower than the possibility, it returns true, otherwise, returns false.
8.	void write_file()
	writes the output result when the best cost in a single SA is smaller than the global cost between SAs
9.	void M1()
	randomly picks up two operands and swaps them, either in the slicing tree or in the polish expression
10.	void M2()
	randomly picks up a chain of operators and inverts them, either in the slicing tree or in the polish expression

11.	void M3()
	randomly picks up an operator and an operand and swaps them, either in the slicing tree or in the polish expression
12.	void delete_tree()
	frees nodes in the slicing tree to *Node *avail* pointer when reaching the end of SA

2. the function used in write_file()

1.	Node * top_down ()
	uses preorder traversal to give the index of selected aspect of each child, and assigns aspects into variable *vector<Module> module*. Then, we can write file by that information.

Parameter adjustment and observation

1. **Observation 1:**
   The module count and # of sampling points of shape curves strongly impacts the run time. Needless to say, the more sampling point an area has, the smaller overall area we get.
- I find that I can run multiple SA on both t10.txt and t20.txt even if they have over 10000 sampling points, so I try to increase # of sampling points to find optimal solution.
- As the module count increases, I decrease # of sampling points so that the program can run SA at least one time. (My program can only handle module count under 4000 or the # of sampling lower than 2, which is meaningless)

2. **Observation 2:**
   In case t500.txt, no matter how I change the initial condition, the pancake shape is still the best solution after finishing SA once.
- I turn to increase # of sampling point of the case t500.txt at first time, then output the pancake solution. Afterward, I decrease # of sampling points, and next start to run SA.

3. **Observation 3:**
   If I use the non-normalized area as the cost function, the initial temperature of bigger case will be very high. The cost function using normalized area has a problem that opposite to the one I just mentioned.
- Therefore, I use the normalized area multiplied by 100 as the cost function.

4. **Observation 4:**

The better solution tends to emerge at lower temperature.

- I use the factor 0.85 recommended in thesis for T > 20, while it's 0.9 for T <= 20.

Makefile

Because I use single .cpp file in this project, there is only one executable file created, i.e., *108501023_PA3.o*. The following is source code of Makefile.

```
# PA3 cpp compiler

.PHONY: all run clean
all: 108501023_PA3.o
	@g++ -std=c++11 108501023_PA3.o -o exe
run:
	@./exe $(input) $(output)
clean:
	@rm *.o
108501023_PA3.o: 108501023_PA3.cpp
	@g++ -std=c++11 -c 108501023_PA3.cpp
```

Fig. 12

Hardness

The hardest part is the debugging procedure after the overall program architecture is nearly finished. The SA engine starts smoothly but the output file is not correct, so I have to trace code to find where the bug is. Independent Move1, Move2 and Move are executed correctly. However, when I mix them into SA, it turns out to be wrong. That's quite annoying.

Also, the adjustment of parameters is also a great challenge. During that time, I still can't figure out a systematic approach to find the best parameter. The only thing I can do is trial-and-error through little observation. Therefore, I learn that I should study more thesis so that I can get more knowledge and come up with more ingenious ideas.

Suggestion

I am grateful for having this project, it helps me integrating data structure background into this project.

Reference

[1] The class slide of chapter 9.
(2023Spring_EE6094_CAD_Chapter9_FloorPlanning)

[2] Wong & Liu, "A new algorithm for floorplan design," DAC-86.