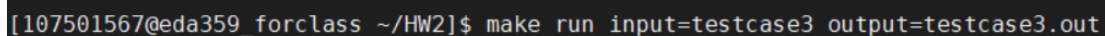


- **Makefile & Readme**

My makefile contains 7 commands, which is (1) *make all*, (2) *make run*, (3) *make run_all*, (4) *make execute*, (5) *make checker*, (6) *make check_all*, (7) *make clean*. The descriptions of each command are shown below.

- (1) *make all*: This command will automatically compile my source code and generate the corresponding objects and executable file.
- (2) *make run*: This command will execute my executable file and read input file to generate corresponding output file.
- (3) *make run_all*: This command will automatically execute my executable file and read all of the input testcase files then generate corresponding output files in orderly.
- (4) *make execute*: This command is an extension of *make run* command. In addition to the original functionality of make run, it also runs checker to check whether my output file is correct or not.
- (5) *make checker*: This command will run checker to check the correctness of my output file.
- (6) *make check_all*: This command will automatically run checker to check all of my output files (testcase1.out ~ testcase3.out) in orderly.
- (7) *make clean*: This command will automatically remove all the output files generated by my executable file (.out) and the objects and executable file generated by *make all*.

To compile and execute my program, first we type command *make all* to compile my source file and generate an executable file, then we can either use command *make run_all* to schedule and generate output files of 3 test cases at the same time, or we can use *make run* to generate output files separately. For example shown in figure 1, we can type *make run* command to read and generate output files of testcase3.



```
[107501567@eda359_forclass ~/HW2]$ make run input=testcase3 output=testcase3.out
```

Figure 1 make run

To verify the correctness of the output files, we can either type *make check_all* to check all the output files or *make checker* to check separately. Figure 2 is the example of *make check_all*, and figure 3 is the example of command *make checker*, if the output files are correct, then we will see a large “YA” on the screen, which will be shown at **Completion** part.

```
[107501567@eda359_forclass ~/HW2]$ make check_all
```

Figure 2 make check_all

```
[107501567@eda359_forclass ~/HW2]$ make checker input=testcase3
```

Figure 3 make checker

- **Completion: All**

I check my output files by command *make check_all*, as figure 4.

```
[107501567@eda359_forclass ~/HW2]$ make check_all  
-----CHECKER-----  
testcase1 is correct! Resource: 4  
*  
 * *  
  * *  
   * *  
    * *  
     *****  
      * *  
       * *  
        *  
-----CHECKER-----  
testcase2 is correct! Resource: 11  
*  
 * *  
  * *  
   * *  
    * *  
     *****  
      * *  
       * *  
        *  
-----CHECKER-----  
testcase3 is correct! Resource: 28  
*  
 * *  
  * *  
   * *  
    * *  
     *****  
      * *  
       * *  
        *
```

Figure 4 Verification

● Program Structure

➤ Data Structure

Since we can get the number of the operations in the circuit from input file, I decided to use struct array instead of dynamic data structure. The data structure I created to store the information of circuit operations is shown in Figure 5:

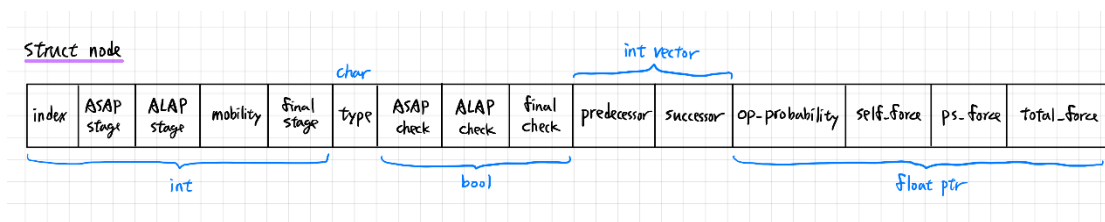


Figure 5 Data Structure

1. index: integer variable which stores the index of the operation
2. ASAP_stage & ALAP_stage: 2 integer variables which store the scheduling result of ASAP and ALAP algorithm
3. mobility: integer variable which stores the mobility of the operation
4. final_stage: integer variable which stores the final scheduling result
5. type: char variable which stores the operation type of the operation
6. ASAP Check & ALAP Check: 2 bool variables which represent whether

the operation has been scheduled by ASAP/ALAP or not

7. final_Check: bool variable showing that whether the operation has been scheduled by the entire algorithm or not
8. predecessors & successor: 2 integer vector variable, storing the indexes of the operation's predecessors and successors.

(Since we don't know the number of predecessors or successors of the operation, so I choose to use vector instead of array)

9. op_probability: float pointer for creating dynamic array, storing the probability of the operation at each stage
10. self_force: float pointer for creating dynamic array, storing self force of the operation when schedule at each stage
11. ps_force: float pointer for creating dynamic array, storing PS force of the operation when schedule at each stage
12. total_force: float pointer for creating dynamic array, storing total force of the operation when schedule at each stage

➤ Algorithm

Figure 6 is the overall structure of my program, and the algorithm I decided to implement is Force-Directed Scheduling, which is marked in the red block with label “FDS”. The detail description of each function in scheduling process will be explain below.

1. ASAP: Figure 7 is the flow chart of my

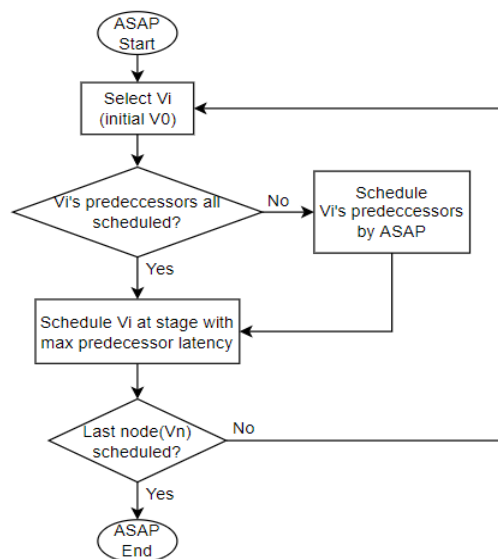


Figure 7 ASAP Flow Chart

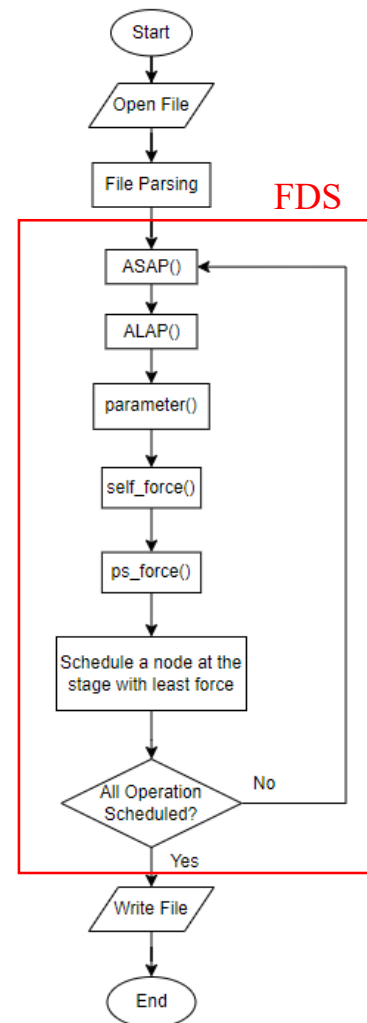


Figure 6 Overall Structure

ASAP algorithm. This function will schedule all the operations at the earliest stage that won't cause dependency error iteratively, but when a operation's predecessors haven't been scheduled, the function will recursively call ASAP again to schedule predecessors first.

2. ALAP: Figure 8 is the flow chart of my ALAP algorithm. Same as ASAP, this function will iteratively schedule the operation at the stage according to the latency constrain, but when an operation's successors haven't been scheduled, this function will recursively call ALAP again to schedule the successors first.

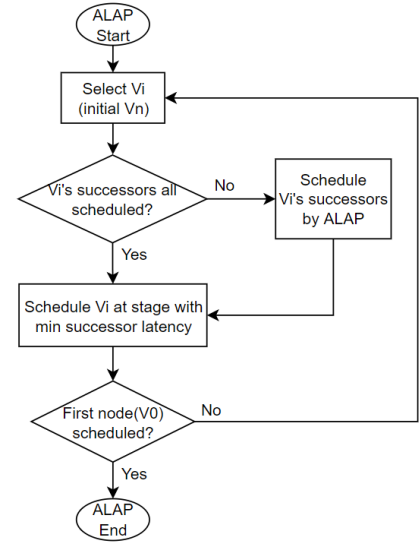


Figure 8 ALAP Flow Chart

4. self_force: This function is to calculate the force for operation v_i in when scheduling at stage l . Figure 9 is the flow chart of this function, which contains 2 loop for traversaling all the possible stage to schedule for every operations.
5. ps_force: This function is to calculate the force from operation v_i 's predecessors and successors when scheduling at stage l . Figure 11 is the flow chart of this function, which is shown in next page due to the size of the chart. This function also calculates total force of each operation when scheduled at every possible stages and return the operation has the least force with the stage it will be scheduled.
6. After function "ps_force", we can get the operation that has the least force. The final step of the FDS algorithm is to schedule the

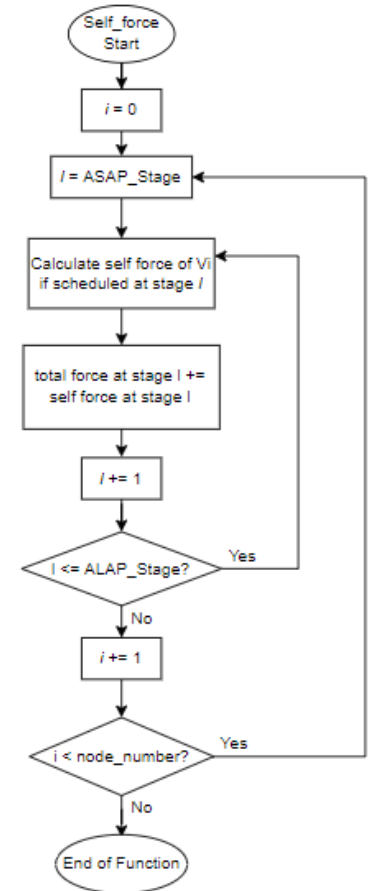


Figure 10 self_force Flow Chart

operation at the stage returned from ps_force function, and this is the end of the entire algorithm, the rest is to repeat the steps above until all the operations have been scheduled.

● Hardness

Compare to PA1, I think the difficulty of PA2 is much harder, although the difficulty of parsing input files is much easier, it's a hard time to implement the whole algorithm. It only took me about 2 hours to implement ASAP and ALAP, but I spent a whole Qingming Festival trying to understand and implement the structure and the process of Force-Directed Scheduling.

The largest difficulty I met was about optimization. I spent a lot of time trying to reduce the run time of scheduling, but the result didn't get any better though. Apart from the time complexity of FDS algorithm structure, I guess another reason is because the space complexity of my program is too high, which will affect the execution speed of my program. Maybe I will continue trying to optimize my program after deadline.

● Review

Although this programming assignment is quite difficult, I believe I've learned a lot during the implementation of FDS algorithm, including my programming skill and the knowledge about scheduling.

P.S. 助教辛苦了

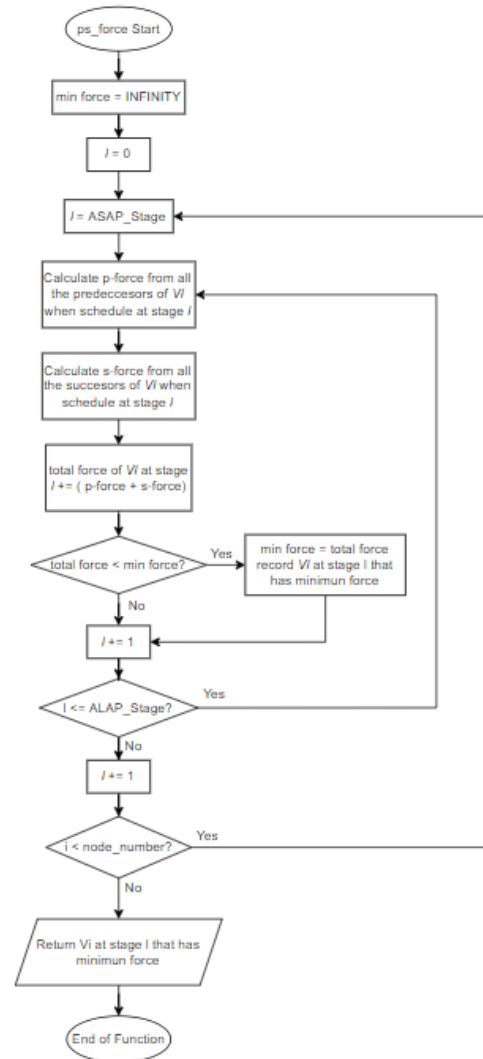


Figure 11 ps_force Flow Chart