50.005 CSE

INFORMATION SYSTEMS TECHNOLOGY AND DESIGN

Programming Assignment 1

Natalie Agus

1 Purpose: Process Tree Management

1. To develop a program that models control and data dependencies, and uses UNIX system calls (fork, exec, wait etc.) in C, to traverse a directed acyclic graph (DAG) of user programs in parallel and execute them.

2. *A data dependency* is one where the outcome of program i becomes the input of program j.

3. *A control dependency* is one where the outcome of program i determines whether program j should be executed or not.

4. *Real-life example:* the UNIX `make` program creates a dependence graph for all the files involved in making an executable binary file. It uses the `makefile`, for which you can write commands in it to organise code compilation. Independent files are compiled individually, whereas dependent files have to wait for all their control dependencies to be cleared before they can be compiled or linked.

5. In this assignment, we are going to build a dependency graph (that is DAG), and execute the codes in each node accordingly. When each node is turned into a process, then we need to manage the process tree to ensure that data and control dependency is not violated. Now instead of using arbitrary codes for each node and whip up some dependency between them, we are going to build a dependency graph that makes sense. One scenario of dependency that you have encountered before is dynamic programming (DP). In the next section, we are going to recap a little bit about DP.

2 Problem Statement: Composing Dynamic Programming as DAG

Remember from a subject that you loved last term, that DP is a method to solve complex problems by breaking it down into a pool of simpler subproblems. We solve each subproblem exactly once, and reuse the solutions from these subproblems each time they were called. If you recall, this technique of storing the results of each subproblem for later lookup is called *memoization*. In other words, dynamic programming traverse the execution of subproblems in topological order, and there exist data dependency between these subproblems, i.e: where subproblem i requires output from subproblem j. There also exist control dependency, where subproblem i should not be recomputed if it is already memoized.

Still don't remember anything? Don't worry, lets dive into the task and figure things out along the way.

2.1 The Coin Change Problem

The particular DP problem that you will encounter in this assignment is specifically the *coin change problem*. It is particularly useful in the process of designing new currencies, that is, to figure out what denomination should be chosen for the coins in order to optimise the average cost of making change. There are many variations to this problem, but the one that is used for this assignment is as simple as follows,

Given a set of m different and unlimited coins: $S = \{S1, S2, S3, ..., Sm\}$, find out how many ways we can make up the change amount V, without considering the order of the coins

Also, **assume that both V and S[i] are always in terms of integer.** So for example, if we have $S = \{1, 2, 3\}$, and $V = 4$, we have **4** different ways to make up this value. They are:

1. Using all 1's : 1 + 1 + 1 + 1 = 4
2. Using all 2's : 2 + 2 = 4
3. Using a combination of 1's and 2: 1 + 1 + 2 = 4
4. Using a combination of 3 and 1: 1 + 3 = 4

The naive way to solve this problem is by recursion. Let `count(S, m , V)` be the function to solve the amount of coin combination that makes up a value of V, and the choices of coins with denomination of $S[1], S[2], ...$ to $S[m]$ are all given to us:

```
int count(int* S, int m, int V){
       if (V == 0) return 1;
       else if (V < 0) return 0;
```

```
else if (V > 0 && m <= 0) return 0;
else return count(S, m, V-S[m]) + count(S, m-1, V);
}
```

While the code above can give us the right answer, it is not an optimum solution as it computes repeated subproblems multiple times. Lets plot the function calls of the recursive function `count` above using a specific example.

2.2 Example

Given:

$$S = \{1, 2, 3\} \text{ and } V = 4,$$

Figure 1 denotes the naive recursive function calls. Repeated function calls are done, denoted with the same color, e.g: `count(S, 1, 2)` is called twice, `count(S, 1, 1)` is called thrice, and `count(S,1,0)` is called four times. These repeated executions are unnecessary and is a waste of resources such as stack frame and CPU time. We can reduce the execution of `count(S, m, V)` by storing its output value the first time it is called. Subsequent calls to `count(S, m V)` are redirected to the stored value instead. Figure 2 illustrates a smaller function calls with dynamic programming to solve the same problem.

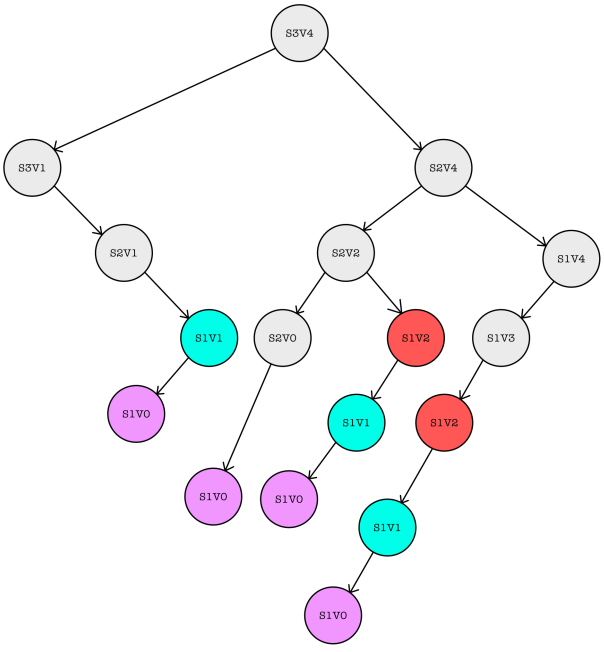

Figure 1: The function call of `count(S, m, V)` using recursive method. Each node $SxVy$ represents each function call, where x is the argument m and y is the argument V.

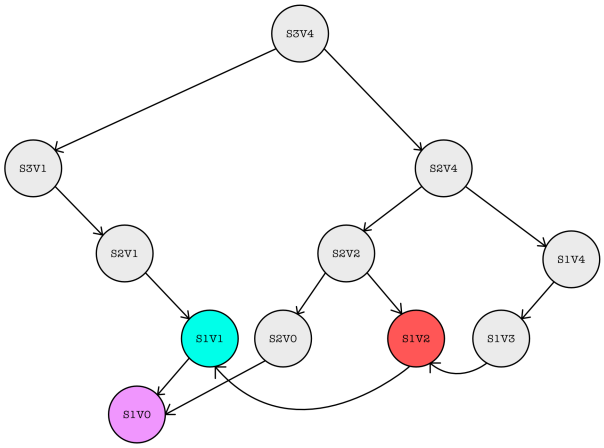

Figure 2: The function call `count(S, m, V)` using dynamic programming. The results of each `count(S, m, V)` are memoized so that there is no double execution of the same function.

2.3 Composing The Dependency Graph of Coin Change DP Problem as a DAG

Notice that the graph in Figure 2 is a DAG. Each `count` subproblem is computed exactly once, and they are computed in a particular order. **The arrow represents *dependency*, that is if $A \rightarrow B$, it indicates that the result of A depends on the result of B.** In other words, the output of B becomes the input of A. So for example, the output from `count(S, 1, 0)` is used as an input to `count(S, 1, 1)`. An alternative way to illustrate a DP problem besides using the function call tree is through a table shown in Figure 3.

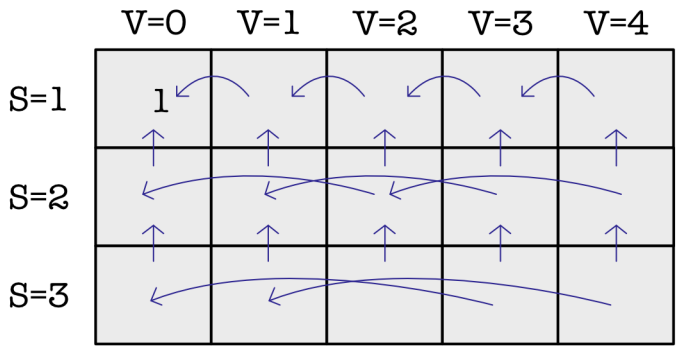

Figure 3: Example function call `count(S, 3, 4)` using dynamic programming, where $S = \{1, 2, 3\}$.

We can notice two types of dependency here:

1. *Data dependency*: e.g, output from `count(S, 2, 4)` and `count(S, 3, 1)` are used as input to `count(S, 3, 4)`.

2. *Control dependency*: e.g, `count(S,1,1)` called from `count(S,1,2)` is no longer executed if `count(S,1,1)` has already been computed before by `count(S,2,1)`.

3 Your Task

3.1 Part 1: Getting Started [5 marks]

The goal of this part is to get started and familiarize yourself with generic counting of coin change problem, where $S = \{S1, S2, ..., Sm\}$ are given to make up a total value of V in terms of DAG shown in Figure 2. Remember, **assume that both V and S[i] is always in terms of integer.** Answer the following questions for this part,

1. **[1m]** Let `count(S, m, V)` be the function to compute the number of possible coin combinations that make up a value V regardless of the order of the coins. Write down the recursive DP formula for `count`.

2. **[1m]** How many nodes **at maximum** should exist at the function call dependency graph when DP is used? Write down your answer in terms of m and V.

3. **[1m]** How many parents can each node have **at maximum** when DP is used?

4. **[1m]** How many children can each node have **at maximum** when DP is used?

5. **[1m]** Topologically fill each cell of the DP table in Figure 3. What is the answer for `count(S, 3, 4)`, where $S = \{1, 2, 3\}$?

3.2 Part 2: Creating the Dependency Graph [5 marks]

The starter code helps you initialize a number of nodes, depending on your answer in Part 1.2. The goal of this part is to link the relationship between nodes to create the function call / dependency tree. In the starter code, we gave a suggested node data structure. Figure 4 shows the screenshot of the data structure.

[5m] Write the code for function :
`void linkRelationshipProcessTreeNode(ProcessTreeNode* nodesDP, int* S, int V, int m);`

When this method finished running, each struct in `nodesDP` should points to its respective parents/children, as well as containing the information of how many parents/children this node has. The following instructions may help you get started,

```
/*
 parents            : points to parent nodes
 children           : points to children nodes
 numberOfParents    : specifies number of parents processes
 numberOfChildren   : specifies number of children processes
 name               : name of the DP cell computed by this
     process. Format is defined as SxVz where x indicates the row
     index and z indicates the column index. Length of z and x can
     vary depends on how big is V and size of S
 cellValue          : the value of the particular DP cell thats
     computed by this process
 pid                : id of this process
 processStatus      : enum type (read above)
 processMethod      : method used to compute the value of the
     particular DP cell tasked for this process
 */
typedef struct ProcTreeNode{
    struct ProcTreeNode* parents[MAX_PARENTS];
    struct ProcTreeNode* children[MAX_CHILDREN];
    int numberOfParents;
    int numberOfChildren;
    int totalNumberOfNodes;
    int nodeNumber;
    char name[BUFFERSIZE];
    int cellValue;
    pid_t pid;
    cellInformation cellInfo;
    status processStatus;
    methodType processMethod;
}ProcessTreeNode;
```

Figure 4: Suggested process tree node data structure

1. Determine the format on how you want to name each node. A suggested way is $S_x V_y$, where x indicates the index of the coin denomination, i.e: if $S = \{1, 2, 3\}$, then $S[1]$ is 1, $S[2]$ is 2, and $S[3]$ is 3, and y indicates the value.

2. Find the name of the root node (there is only 1 root node)

3. Find the name of the leaf node (there is only 1 leaf node)

4. Determine the parents (if any) and the children (if any) of each node. Given any node $SxVy$, you can actually programmatically find **the name** of all of its parents (or children) nodes (if any). *Hint: look at the recursive DP formula you wrote in Part 1. What do* `count(S, m, V)` *depend on?*

5. **It is up to you to define "parents" or "children" as long as you are consistent in your data structure**. The dependency graph does not specify who are the "parents" or the "children". Intuitively, the node at the end of the arrow is the child, and the node at the beginning of the arrow is the parent. In this case, child has to wait for parent(s)' output because the input to child depends on the output of the parent(s). However the opposite will work as well, where the node at the end of the arrow is the parent, and the node at

the beginning of the arrow is the child. In this case, the parent(s) have to wait for the children's output because the input to parents depend on the output of the children. Note that in both cases, if you use $fork()$ system call, **the parent process has to wait for the children process to exit before exiting itself**, otherwise you will end up with plenty of zombie processes.

6. The amount of `MAX_PARENTS` and `MAX_CHILDREN` depend on your answer from Part 1.

7. `ProcessTreeNode* nodesDP` contains an array of initialised struct `ProcessTreeNode`. The size of this array is the maximum number of nodes that should exist in the tree when DP is used. It is determined by your answer in Part 1, question 2.

8. The purpose of function `linkRelationshipProcessTreeNode`, is to link up each `ProcessTreeNode` with its parent(s) or children, if any.

9. To help you debug, you can pass the array `nodesDP` and the total number of nodes in the tree to function `printTreeForDebugging`.

3.3 Part 3: Computing `cellValue` field of each node sequentially [10 marks]

The goal of this section is to figure out how to compute the `cellValue` of each node sequentially, i.e: without multithreading or multiple processes.

[10m] Write the code for function:
`void runFunctionSequentialMethod_Create(ProcessTreeNode* Root)`

When this method finished running, all the `cellValue` field of each node has to be correctly filled. You can use `printTreeForDebugging` function to check the `cellValue` of all nodes. Before setting the `cellValue` of each node, you need to make sure that all the nodes that its dependent on are finished. You can make use of `typedef enum` given in the header file to indicate whether the `cellValue` of the node has been properly computed or not.

3.4 Part 4: Computing `cellValue` field of each node using multi threads [10 marks]

The goal of this section is to figure out how to compute the `cellValue` of each node using threads: `pthread_create`.

[10m] Write the code for function:
`void runFunctionThreadMethod_Create(ProcessTreeNode* Root)`

When this method finished running, all the `cellValue` field of each node has to be correctly filled. You can use `printTreeForDebugging` function to check the `cellValue`

of all nodes. *Hint: One way to do complete this task is to initialise one thread for each node, and compute the* `cellValue` *only if all the other threads taking care of the nodes it is dependent on have finished running.* Threads share the same address space, so you can make use of this feature to pass information between threads through variables in the code itself.

3.5 Part 5: Computing `cellValue` field of each node using multi processes [10 marks]

The goal of this section is to figure out how to compute the `cellValue` of each node using processes: `fork()` system call.

[10m] Write the code for function:
`void runFunctionProcessMethod_Create(ProcessTreeNode* root)`

When this method finished running, all the `cellValue` field of each node has to be correctly filled. You can use `printTreeForDebugging` function to check the `cellValue` of all nodes. *Hint: One way to do complete this task is to initialise one process for each node, and compute the* `cellValue` *only if all the other processes taking care of the nodes it is dependent on have finished running.* Since processes do not share the same address space, you need to use **input and output redirection** or **shared memory**: `shmget` in c to pass information between processes.

3.6 Part 6: Concluding everything [10 marks]

You have basically solved the same DP problem using three methods: sequentially, multi-threaded, and multi-processed. All three methods should give the same answer given S and V. The only difference is the time taken to solve the problem. In class, we learned that process creation provides isolation but suffers overhead, while thread creation does not offer the same level of isolation but it still offers concurrency or parallelism with less overhead. Sequential execution on the other hand, is just the plain simple execution of your codes with a single thread. Let's find out the tradeoff between each method as the size of S and V varies:

1. **[3m]** Time the execution of `runFunctionSequentialMethod_Create`, `runFunctionThreadMethod_Create`, and `runFunctionProcessMethod_Create` methods separately with $S = \{1, 2, 3\}$ and V varied from 4 to 20. You can use standard C library `clock()` to aid you in this, or any other way to measure / approximate execution time.

2. **[3m]** Time the execution of `runFunctionSequentialMethod_Create`, `runFunctionThreadMethod_Create`, and `runFunctionProcessMethod_Create` methods separately with $S = \{1, 2, 3, 5, 8, 9, 10\}$, where m is varied from 3 to 7, and V fixed to 10. Example, if $m = 4$, that means $S = \{1, 2, 3, 5\}$ (only include the first four coins). You can use standard C library `clock()` to aid you in this or any other way to measure / approximate execution time.

3. **[3m]** Plot your answers from the two parts above. x-axis to indicate V or m, and y-axis to indicate time. You should have 6 different plots, 3 from each part.

4. **[1m]** Argue which method is the most efficient in solving this particular DP problem. Your answer may depend on the value of m and V.

4 Bonus (5 marks)

Another variation to the DP coin change problem is finding out the **minimum number of coin change to make up a value** V. This is applied in real-life, i.e: how the vending machines manage to give you change with the least number of coins possible. The naive recursive algorithm for minimum coin change problem is:

```
#define MAXVALUE INT_MAX

int countMin(int* S, int m, int V){
       if (V == 0) return 0;
       else if (V < 0) return MAXVALUE;
       else if (V > 0 && m <= 0) return MAXVALUE;
       else return getMin(1 + countMin(S, m, V-S[m]), count(S, m-1, V));
       }

static int getMin(int a, int b){
    return a>b ? b : a;
}
```

You can complete this section by adding small modifications to your answers from Part 2 to Part 5 (**total of [5m]**) to solve this minimum coin change problem instead. Most of the solution remains the same, i.e: the dependency between each node is the same, there is still 1 root node and 1 leaf node, V and S are all in terms if integers. The apparent differences between the two are:

1. The `cellValue` of each node should be initialized to `MAXVALUE` instead of 0

2. The setting of each `cellValue` utilizes `getMin` instead of just adding the `cellValues` from both parents together

5 Submission

The total maximum marks given for this assignment is 50. If you scored full marks on Part 1-6, score from the bonus section will not be taken into account.

Type out your answers for Part 1 and Part 6 and submit in a .pdf file format not exceeding 3 A4 pages in length. Submit it together with your code to e-dimension. You can write your answer for Part 2 to Part 5 in the skeleton code provided. **For the bonus part, make a copy of the .c file and put your answer for the bonus part in this new copy.** Although the code can be more elegant by integrating both counting and minimum coin problems together, separating them will allow us to grade you more easily.