Archives Package Management

Hans C. Suganda

21^{st} June 2022

Contents

0.1 Rationale

As the Archives project gets larger and larger, it became necessary to organize the content. A good document must contain a high density of useful information, which is defined as the amount of useful information divided by page number. If the information density is too low, the reader will get disinterested.

Since the Archives project is meant to cater to a large variety of audience whose technical backgrounds vary from experts to beginners, the Archives project needs capabilities to display the "depth" of a certain topic. For example, a beginner might need an explanation to the most basic pre-requisite concepts to an advanced topic, but an expert might not need this since they already know from heart the basics of the topic.

The reader can be thought of something similar to a computational engine in logical capability, and the Archives can be thought of something akin to source code. What is needed now is a package manager that compiles the content/source code into useful readable format with all the necessary pre-requisites and is conflict-free.

0.2 Learning Model

How do humans really learn? From experience, it seems that academic knowledge should be treated like some kind of game. This seems odd. Really? A game? Let me explain.

A game is typically iterable over long periods of time. A game ideally should be non-deterministic because if it was deterministic then everyone would know the final outcome of the game based on the initial setup. If everyone could predict the final outcome of the game, there is no point in playing it and hence, the game is not iterable over time. This is an important quality that makes games "fun" because it is really difficult to tell who is going to "win" or "what" is going to happen. Academic knowledge is somewhat like this. Because of the complexities and difficulty within it, it is essentially impossible to tell what would happen in the future, which means that academic knowledge first the iterable criteria.

A game has to have rules. These rules are important because it constrains the players in a way such that their interactions become interesting. If there were no rules, then the game quickly devolves to a more predictable and less interesting state. Academic knowledge is constrained to be grounded in objective truth. The rules to academic knowledge is basically the rules of the universe itself, and the universe is unpredictable. This means investigators to these types of games have a high chance of discovering unexpected behaviours and patterns which makes academic knowledge very interesting. This means that academic knowledge satisfies the rules constrains.

A game typically involves multiple parties. A game that is played alone can be fun, but it would be more fun to play with friends. Academic knowledge is highly collaborative because of the sheer vastness of it. It is impossible for someone to know all the sum of human knowledge all at the same time, so having collaboration is the only viable way to proceed more effectively. This nature certainly encourages collaboration and hence ties the community aspect of games into academic knowledge.

0.2.1 Human Motivation

Humans are not machines. We have an intrinsic will and frankly learning a new subject is very difficult. Before we even begin to ask that a learner make the appropriate sacrifices in time and effort, we must show "why" they should. A vision that is powerful enough and inspiring enough is necessary to push through the difficulties of learning. It is not sufficient to just "command" that we all should learn a topic and sink so much time and rescources if there is not much to be ultimately gained.

0.2.2 Understanding New Concepts

From experience, it seems that there are 2 primary ways someone could learn a particular topic. The first one is to learn the "game" by being told explicitly what the rules are and then playing the "game". The second one is to learn the "game" by first playing it and based on what works and fails, infer the rules that dictate the "game". There are advantages and disadvantages to both methods.

Learning a concept by looking at the derivation is a very direct and general way to learn a particular topic. If this process is completed succesfully, the learner would have a complete picture and is able to understand all the possible cases. However, this method requires extreme attention to detail and requires alot of mental effort. Moreover, not everyone has the necessary pre-requisite to attempt such a direct assault on the topic.

Learning a concept by looking at the examples and then trying to "infer" the theory is a widely used method. This method typically requires less mental effort, but would fail in letting the learner know why and how the theory came to be. Moreover, if the example does not cover all possible cases, then it is possible that the learner later be exposed to a case not covered in the examples part and they would be left confused and unable to do anything.

It is also very common to learn the rules first by looking at the examples, and then reading the theory behind it. This is a good method provided examples do exist because it combines the initial speed and ease of learning by example, with the generality and completeness of the theory/derivation. A purely leraning by theory approach and purely learning by example approach seems inadequate.

0.2.3 Reinforcement & Adjustments

Human beings are creatures of habits and repetition. To really decrease the overhead of thinking about something and increase speed, humans need repetition at a particular task. This is very applicable to learning. Without applying what we learn, we would simply forget what was learnt and would have to rebuild from scratch again. This is why it is incredibly important to reinforce the concept using repetition. Understanding the concept is halfway, the other half is repetition to truly remember everything. Eventually with sufficient reptition, even the hardest concepts are as easy as flipping out like $2+3=5$. Remember that once upon a time a very long time ago, we all had difficulty with additions. There is no difference back then and now, only the names of the topics have changed.

No matter how perfectly we try to understand new novel information, it is very likely we would not understand it completely correctly at first. This is limited by alot of factors but primarily by language and context. A particular word like say "value" would mean many different things to many different people from a variety of life contexts. Therefore for a learner to truly grasp a particular concept as perfectly as possible, the last stage would be to correct all of the misconceptions about a particular topic.

The most effective way to figure out all the misconceptions is to try out practise problems and checking with the solution. Practising the problems force the learner to apply what they know so that any misconceptions can be revealed. Checking against the solution verifies any major discrepancies and mistakes. These mistakes can be used by the learner as feedback to correct their own understanding. By the end of the process, the learner should have a complete understanding of the concept with as few misconceptions as possible.

0.3 Structure of Archives Project

The Archives Project is essentially made of Nodes. Conceptually, a Node is the most indivisible unit of content. A Node should contain all of the following files, in the same directory. This makes sense

because why would you want to spread a single Node's content over differing directories? Splitting the content this way allows us to focus on one aspect at a time. Apart from the metadata, all the files listed below should contain valid LaTeXcontent but without the pre-amble. This means the pure content is not compilable on its own but with the correct pre-amble, is compilable.

1. Metadata: This contains information on how this particular depends on other Nodes, and which other Nodes depend on this particular Node.

2. Rationale: Why should you learn this topic? Why is this topic important? What are the implications to this topic?

3. Theory file: The derivation, and the most abstract generalized theory the entire concept is based on. This is for learners who learn the game by being told what the rules are.

4. Examples: This is how the theory is applied. This is a realization of the Theory to a specific set of cases. This is for learners who learn the game by playing it first and then inferring the rules.

5. Problems: These are problem sets which are similar to the examples which helps learners train for a particular concept. These in conjunction to the solution can be used for extra examples.

6. Solutions: These are the exact solutions to the problems posed. This is important to give feedback on particular mistakes.

The file structure is translated directly from the human learning model. The rationale covers the human motivational aspect. Without the rationale, humans do not have anything they can hold on to in withstanding the difficulty of learning a new subject.

The Theory & Example are files which corresponds to the Understanding New Concepts part. Both files support learning the game by playing it or learning the game by reading its explicit rules. This makes the Archives a very versatile environment.

The Problems & Solutions correspond to the Reinforcement-Adjustment aspect. The Problems & Solutions ensure that learners can test their undertanding and be corrected in a constructive and helpful environment.

0.4 Pre-Processor

We take inspiration of the pre-processor from the pre-processor of the `C` language whose primary function is copy-pasting source code. The pre-processor for `Beringin` is written in `Rust` for a high performance and memory safety in handling nodes and so on. Currently, the pre-processor contains 4 separate modules:

- Tree: this is where a single node structure is defined. Helper functions pertaining to the initialization, and printing of a node structure are implemented in this module.

- Loading: this handles going through all the directories in Beringin and loading the node representations into main working memory.

- Assembly: given the intended node target, this handles the printing queue order of all the relevant nodes.

- Output: given the printing queue order, this handles copy pasting the node contents in order into a designated output file.

0.4.1 Trees.rs

The Node data is obtained from the Metadata file. Programmatically, the Node is an object which contains the information listed below:

- Topic Name: A unique name to identify this Node or content. This is the only way to identify a Node. No two Node can have the same name.
- Description: This is a short description of the topic contained in this Node. This
- Pre-Requisites: What are the other Nodes does this particular Node depend on? The other Nodes will be identified by their unique topic name.
- Builds-Into: What are other Nodes uses this particular Node as a pre-requisite? The other Nodes will be identified by their unique topic name.
- File List: What are the names of the files containing the Rationale, Theory, Examples, Problems, and Solutions? If this is left blank, then a default name is assumed.
- Organizational Hierarchy Level: Relative level of the Node. This can only hold the values between $1 - 7$ which corresponds to the default maximum organizational hierarchy LaTeXhas.
- Author: Who wrote this particular piece. This is not really necessary, it is just for fun. This can be left blank.

The source code implementation of a node from our previous discussion is shown below,

```
use std::fmt;

/*7 levels of LaTeX organizational hierarchy*/
pub enum OrgHier {
    //A is largest, like part and G is the lowest, like sub-paragraph
    A, B, C, D, E, F, G,
}

/*This contains just the metadata of a single knowledge archive Node*/
pub struct Node {//Core Metadata of a single Node
    pub Name: String, //Name of the Node
    pub Desc: String, //Short Description of what the Node contains
    pub Pre_Req: Vec<usize>, //Topic Names of this Node's Pre-Requisites
    pub Builds_To: Vec<usize>, //Topic Names of what this Node builds into
    pub Org_Hier: OrgHier, //Where in the Latex organizational level does this reside in
    pub Author: Vec<String>, //Who are the contributing authors to this node
    //Optional Specificity on the Node's file Names
    pub FRationale: Option<String>, //Rationale File-Name (optional)
    pub FTheory: Option<String>, //Theory File-Name (optional)
    pub FExamples: Option<String>, //Examples File-Name (optional)
    pub FProblems: Option<String>, //Problems File-Name (optional)
    pub FSolutions: Option<String>, //Solutions File-Name (optional)
}

/*Given an OrgHier object, output the corresponding Name in String*/
fn Org2Str(OrgObj: &OrgHier) -> String {
    match OrgObj { //Uses the match control flow to convert representations
        OrgHier::A => String::from("A"),
        OrgHier::B => String::from("B"),
        OrgHier::C => String::from("C"),
        OrgHier::D => String::from("D"),
        OrgHier::E => String::from("E"),
        OrgHier::F => String::from("F"),
        OrgHier::G => String::from("G"), }}

//Implement std::fmt::Display on Node so we can see what is inside Node
```

```
impl fmt::Display for Node {
    fn fmt(&self, f:&mut fmt::Formatter) -> fmt::Result {
        let mut Out: String = String::new();
        //Just Appending the Strings of the Name and Description to the output String
        Out = Out + "Node␣Name:␣" + &self.Name.to_string() + "\n";
        Out = Out + "Desc:␣" + &self.Desc.to_string() + "\n";
        //Using the DEBUG formatter to print the vectors of usizes for both pre-req and build to
        Out = Out + "Pre_Req:␣" + &format!("{:?}", &self.Pre_Req).to_string()  + "\n";
        Out = Out + "Builds_To:␣" + &format!("{:?}", &self.Builds_To).to_string()  + "\n";
        //Uses some function Org2Str to convert the enum to its corresponding string representation
        Out = Out + "Org_Hier:␣" + &Org2Str(&self.Org_Hier) + "\n";
        //Using the DEBUG formatter to print the list of authors
        Out = Out + "Author:␣" + &format!("{:?}", &self.Author).to_string()  + "\n";
        //Return the result of a display using write!
        write!(f, "{}", Out)}}
```

The source code above also implements an `enum` named `OrgHier` that is used to represent the organizational hierarchy level of a certain node as discussed. The source code above also implements debugging functions to display the Node data structure from the command line so that debugging can be more convenient.

0.4.2 Loading.rs

All LaTeXnodes within Beringin are placed in a single main directory. Each node resides in its own node-specific directory whose name is going to represesnt the Node name in main memory. All of the metadata files and the LaTeXfile specifically for a single node is going to be placed inside each individual node-specific directory. Nodes are NEVER to be hidden inside the directory of another node. This means that all of the nodes in the archive is explicit and the directory structure is very shallow. No recursion is needed to traverse ALL of the nodes in the project.

The source code that handles the node loading is shown below,

```
use std::io::Read; //Allow to read file contents
use walkdir; //Allow to go through contents of current directory while skipping hidden files
use std::collections::HashMap; //Allow the creation and usage of hash maps
use crate::Trees; //Read in the structure definitions of single Nodes and all asociated functions

/*Uses the given path to a file to extract file's contents and put it in a string*/
pub fn ExtractFile (RelPath: &std::path::Path, Holder: &mut String){
    //Error message that shows if the file cannot be opened
    let ErrMsg = "Cannot␣Open␣File...";
    //Open the file using the given path and set it to the file object
    let mut FileObj = std::fs::File::open(RelPath).expect(ErrMsg);
    //Use the file object to then put all of the file's contents into the given string
    FileObj.read_to_string(Holder);}

/*Function to determine if a file/directory is a dotfile or not*/
fn is_hidden(entry: &walkdir::DirEntry) -> bool {
    entry.file_name().to_str().map(|s| s.starts_with(".")).unwrap_or(false)}

/*Gets toml::value and attempts to parse out the string. returns Option::None if fail*/
fn Toml2String(Parser: &toml::Value, index: &str) -> Option<String> {
    let extracted = Parser.get(index);
    let mut ans = Option::None;
    if extracted != Option::None {
        ans = Some(String::from(extracted.unwrap().as_str().unwrap()));}
    ans}

/*Gets toml::value and Parses out a string vector*/
fn Toml2StringVec(Parser: &toml::Value, index: &str) -> Vec<String> {
    let extracted = &Parser[index];
    let mut ans: Vec<String> = Vec::new();
```

```
    let TomlVect = extracted.as_array().unwrap().to_vec();
    for i in TomlVect{
        ans.push(String::from(i.as_str().unwrap()));}
    ans}

/*Gets toml:: value and Parses out OrgHier for a Node*/
fn Toml2OrgHier(Parser: &toml::Value, index: &str) -> Trees::OrgHier {
    let extracted = &Parser[index];
    let Char = extracted.as_str().unwrap();
    let mut ans = Trees::OrgHier::G;
    if Char == "A" {ans = Trees::OrgHier::A;}
    else if Char == "B" {ans = Trees::OrgHier::B;}
    else if Char == "C" {ans = Trees::OrgHier::C;}
    else if Char == "D" {ans = Trees::OrgHier::D;}
    else if Char == "E" {ans = Trees::OrgHier::E;}
    else if Char == "F" {ans = Trees::OrgHier::F;}
    ans}

/*Initialize a new Node of a tree based on data from a string*/
fn ParseNode(DirName: &String, Parser: &toml::Value) -> Trees::Node {
    //Declaring a Node and returning the Node
    Trees::Node {
        Name: DirName.clone(),
        Desc: String::from(Parser["Description"].as_str().unwrap()),
        Pre_Req: Vec::new(),
        Builds_To: Vec::new(),
        Org_Hier: Toml2OrgHier(&Parser, "Organization"),
        Author: Toml2StringVec(&Parser, "Authors"),
        FRationale: Toml2String(&Parser, "Rationale_Name") ,
        FTheory: Toml2String(&Parser, "Theory_Name") ,
        FExamples: Toml2String(&Parser, "Example_Name") ,
        FProblems: Toml2String(&Parser, "Problem_Name") ,
        FSolutions: Toml2String(&Parser, "Solution_Name") ,
    }}

/*Initialize string vector containing dependency of a node based on data from a string*/
fn ParseDep(Parser: &toml::Value) -> Vec<String> {
    let mut out: Vec<String> = Vec::new();
    let extracted = &Parser["Pre_Requisites"];
    let TomlVect = extracted.as_array().unwrap().to_vec();
    //VERY IMPORTANT! WE JUST FOUND A NEW METHOD CALLED TO STRING
    for i in TomlVect{out.push(String::from(i.as_str().unwrap()));}
    //return the vector of strings
    out}

/*Binds Nodes in a pre-requisite built to link*/
fn BindSingle(data: &mut Vec<Trees::Node>, BuiltLoc:usize, PreReqLoc:usize) {
    data[BuiltLoc].Pre_Req.push(PreReqLoc); data[PreReqLoc].Builds_To.push(BuiltLoc);}

/*Iterates through all directories and loads all the metadata Nodes*/
pub fn LoadAllNodes() -> Vec<Trees::Node> {
    //Declaration of vector that holds the entire project's metadata Nodes
    let mut data: Vec<Trees::Node> = Vec::new();
    //Declaration of a vector of vector of strings holding the pre requisite chain
    let mut PreReqRaw: Vec<Vec<String>> = Vec::new();
    //Declaration of a hash map
    let mut Name2Index: HashMap<String, usize> = HashMap::new();
    //Declaration and Initialization of indices
    let mut index: usize = 0;
    //Creation of Directory Iterator. Min Depth and Max Depth set to see just 1 level
    let CurrDir = walkdir::WalkDir::new(".").min_depth(1).max_depth(1).into_iter();
    //Uses the iterator in for loop, but ignores all the hidden files using the is_hidden function
    for i in CurrDir.filter_entry(|e| !is_hidden(e)){
        //Refresh String each loop iteration to contain content of a file
```

```
        let mut FContent = String::new();
        //Refresh String each loop iteration to contain name of Node
        let mut NodeName = String::from(i.as_ref()
            .unwrap().path().file_name().unwrap().to_str().unwrap());
        //Append the path with a default metadata file assumed name and read that file
        ExtractFile(&i.unwrap().path().join("Daun.toml"), &mut FContent);
        //Parse the contents of the file using toml library
        let Parser: toml::Value = toml::from_str(&FContent).unwrap();
        //Create a Node based on the directory name and metadata file and append to vector of Nodes
        data.push(ParseNode(&NodeName, &Parser));
        //Append the vector of pre-requisites
        PreReqRaw.push(ParseDep(&Parser));
        //Append to the Hash map and consume the Node Name
        Name2Index.insert(NodeName, index); index += 1;}
    //Check that the length of pre-requisite vector and Node Vector is the same
    assert!(data.len() == PreReqRaw.len());
    //Bind the various topics together using the hash map and the pre-requisite string vector
    for i in 0..data.len(){//Iterate through all Loaded Nodes
        for j in &PreReqRaw[i]{//Within a single Node, iterate through its pre-req list
            if j != &String::from("Base") {//Don't do anything if a Node's pre-req is "Base"
                //If a Node is not a "Base Node" then perform Binding on its pre-requisites
                BindSingle(&mut data, i, Name2Index[j].clone());}}}
    //Return the vector of the entire project's metadata Nodes
    data}

/*Function that goes through the vector of Nodes and prints it out*/
pub fn ShowInputData(data: &Vec<Trees::Node>) {
    for i in 0..data.len(){println!("Index:␣{}\n{}", i, data[i]);}}
```

Each LaTeXnode has a metadata file named `Daun.toml` . When the pre-processor is run, the pre-processor only loads the information presented in the metadata file into main memory. The pre-processor will go through every node-specific directory within the singe main directory and then initialize a node object and append it to a rust vector. Each node that has been read will reside within this rust vector. This mmeans that all loaded nodes can be identified by their position in the rust vector. This is the reason why all the pre-requiste and build-to vectors are of type `usize`: because they reference other nodes based on position in the node vector.

Loading the nodes only require a single pass through all of the directories. Nodes are initialized with their pre-requiste and build-to lists as empty vectors. The pre-requiste list and the build-to list of the metadata file are loaded into string vectors. We then used a hash map to match the string vectors to the actual names of the nodes and "bind" the pre-requiste node and the build node. Binding is a process where we append the pre-requiste list of the build-to node and we also append the build-to list of the pre-requistie node.

0.4.3 Assembly.rs

Complicated topics have a "deep" dependency, requiring alot of other nodes meanwhile some simple topics might have a "shallow" dependency, only requiring just a few other nodes. The "depth" that the pre-processor has to travel to is changing based on which node the pre-processor is asked to run. This is the perfect problem to apply recursion to. The source code below handles building the nodes in the correct order by using recursion,

```
use std::collections::HashMap; //Allow the creation and usage of hash maps
use crate::Trees; //Read in the structure definitions of single Nodes and all asociated functions

/*Use recursion to assemble the build order of the nodes based on target*/
fn PathRecurse(TreeRepr: &Vec<Trees::Node>, BuildOrder: &mut Vec<usize>,
EntryPoint: usize, Map: &mut HashMap<usize, bool>){
    //True only if we have not reached the base of the knowledge tree
```

```
    let Enter: bool = TreeRepr[EntryPoint].Pre_Req.len() != 0;
    if Enter {//Enter Recursion only if the conditions above are satisfied
        //Iterate over the pre-requisite of a given node first before doing contents this node
        for i in &TreeRepr[EntryPoint].Pre_Req{
        //Only Enter Recursion if a given pre-requisite has not been added yet
        if Map.get(&i) == Option::None {//Enter Recursion
            PathRecurse(TreeRepr, BuildOrder, i.clone(), Map);}}}
    //Append the Current Node to the hash map
    Map.insert(EntryPoint, true);
    //Append the Current Node to the build Order
    BuildOrder.push(EntryPoint);
}

/*This function is just a setup for the entry point and hash map for the recursion path function*/
pub fn FormPath(FullMetadata: &Vec<Trees::Node>, BuildOrder: &mut Vec<usize>, NodeSel: & Vec<usize>)
    {
    //Declare a Hash Map for a simple search algorithm
    let mut IndexSearch: HashMap<usize, bool> = HashMap::new();
    //Use the location of the node as entry point and iterate over the selected Nodes
    for i in NodeSel{
        if IndexSearch.get(i) == Option::None {
        PathRecurse(FullMetadata, BuildOrder, i.clone(), &mut IndexSearch);}
    }}
```

0.4.4 Output.rs

Producing an output is relatively easy. Once we have the build-order which is basically a vector of `usize` indicating which nodes to print and in which order, we can begin copy pasting all the required nodes to a specified output file. The copy pasting function works by loading the contents of the original LaTeXfile onto main memory and then appending the contents to the target file. After appending the contents of one file, the content of the already copied file is freed from main memory and the process begins again. This makes the pre-processor very memory efficient because it only loads the content one node at a time instead of loading the contents of ALL NODES.

```
use std::path::Path; //Give Ability to create a path address
use std::io::Write; //Give Ability to write string to files
use std::fs::OpenOptions; //Give Ability to Open File with Certain Options
use crate::Trees;
use crate::Loading;

/*Copy Paste input file to output file*/
fn AppendCopyPaste(InputAddr: &Path, OutputAddr: &Path) {
    //Declare a string that contains the content of the input file
    let mut content: String = String::new();
    //Read in the file to be copied based on given address
    Loading::ExtractFile(InputAddr, &mut content);
    //Open the File we want to Write To
    let mut OutFile = OpenOptions::new().append(true).create(true)
        .open(OutputAddr).expect("Opening␣File␣Failed");
    //Write out the file that we want to Copy
    OutFile.write(content.as_bytes()).expect("Write␣Failed");}

/*Generate Latex Output by Loading each Node one at a time and copy pasting*/
pub fn BuildLaTeX(data: &Vec<Trees::Node>, Order: &Vec<usize>) {
    //Location of where the intended output file would be
    let OutAddr = Path::new("/home/hyahoos/Utility/Free/Testing.tex");
    //Location of the beginning document  File
    let BegAddr = Path::new("../Environment/Doc_Beginning.tex");
    //Location of the Ending document File
    let EndAddr = Path::new("../Environment/Doc_Ending.tex");
    //Generate the Pre-Amble Part First
    AppendCopyPaste(&BegAddr, &OutAddr);
```

```
    //Iterate through the build order which contains the ordered Nodes ready to be copy pasted
    for i in Order{//Perform the Copy Pasting for each element in the build order vector
        //Use the Name of the Node to figure out its relative path and the default theory file name
        AppendCopyPaste(&Path::new(&(data[*i].Name.clone()+&"/Theory.tex")), &OutAddr);}
    //Generate the end Part of the Document
    AppendCopyPaste(&EndAddr, &OutAddr);}
```

0.4.5 Further Features

There are a few important realizations. Each run of the algorithm will choose Nodes such that their organizational Hierarchy Level is going to be consecutive. Not every single Node will have a complete File-List. This is because there will be some Nodes which are just sections, which will contain the rationale, but then not contain practise problems. There will be the subsubsections which are specific and may not contain rationale but then contains problems and solution so not every node will have one.

Why do we do it this way? because there are certain classes of nodes that rely on the same rationale or that its not possible for you to have questions from such a broad topic. For example you have differential equations, but which type do you want? Certainly the study of differential equations has a good rationale, but do you want practise problems literally placed at the section before the different types of odes are even explained?

Suppose we have a run that produces Nodes with organizational hierarchy lists: 7, 6, 5, 4, the user should also have some way of reducing all of them by a set amount so we dont have parts and chapters for a small document. So the organizational hierarchy list after modification becomes 5, 4, 3, 2 or something like that.

We may also need a package checker, to make sure that the dependencies are done correctly.