Kurt Medley
Computing Practice and Theory
May 23, 2013

Cross-Paradigm Modeling Analysis

As a preface to this report, I will briefly describe the semantics of the title, as it may seem ambiguous. The point of this research project was to investigate the stylistic differences between modeling problems with a functional paradigmatic design and an imperative, object oriented design. I chose Haskell as my core (functional) language in which to write all of my source code. I converted my last implemented program, breadth-first search, into Java (imperative, object oriented) for analysis of language features and syntax. I will discuss my experience through a series of detailed comparisons of language particulars and present a case for addressing distinct problems with specific languages. I will include snippets of the actual source code interspersed with my descriptions as well as the functioning programs in the addendum. The analysis of this report will come in sections 2 and 3.

Section 1: A list of the programs completed

1.1 - Haskell - Deterministic Finite-state Automata Minimizer (DFA minimizer)

1.2 - Haskell - Construction of a Regular Expression from a Finite State Machine (Alg622)

1.3 - Haskell - Breadth-First Search (BFS)

1.4 - Java - Breadth-First Search (H - translation)

To give some context to each of these programs, I'll sectionally divide the report into segments, explaining each. I chose to work, primarily, with Thomas Sudkamp's *Language and Machines*, for which I extracted pseudo-code from the automata-theoretic algorithms presented in the text and translated them into functioning programs. Toward the end of the project's elapsed time, I worked with the graph-theoretic algorithms in John Dossey's (et al.) *Discrete Mathematics* and produced a functioning Breadth-First Search program in Haskell which I then translated into a functioning Java implementation.

Section 1.1 : The DFA minimizer is a program that takes a representation of a DFA (usually a quintuple → { Q, E, D, q0, F } where Q = { States }, E = { Alphabet }, D = { Delta Transitions }, q0 = Start State, F = Final) and collapses it to a minimal 'state'; the result being observably simpler. This can naturally be represented by Haskell's list notation. My grammar/type looks like this:

```
type DFA = [Transition]
type Transition = (String, Int, Int, String, String)
```

A DFA is a list of transitions, namely [Transition] = (String → current state, Int → start state?, Int → is final?, String → input symbol, String → end state) The algorithm works by collapsing "undistinguishable" states. In this implementation, pairs of states are analyzed and decidedly "distinguishable" if one state is ∈ F and one is not. The minimizer produces a final, reduced machine that is equivalent to the initial, more convoluted machine without altering the regular language accepted by the machine. A visual representation of the algorithm at work:

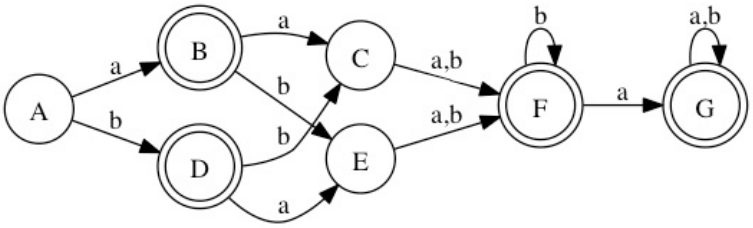


Figure 1 : A DFA that accepts the regular language a∪b(a∪b(a∪b)+)?

Figure 1 shows a machine that stands to be minimized. It is unclear by first glance what regular language this machine accepts. The next visuals show each iteration through the algorithm. The machine is being "pruned", or collapsing based on how the algorithm determines which pairs of states are indistinguishable.

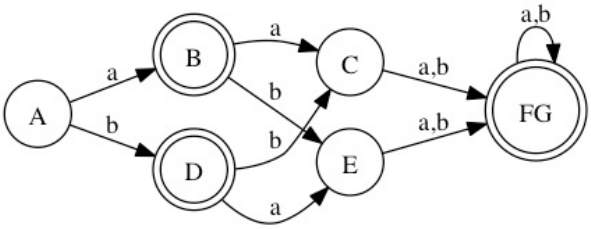


Figure 2 : States F and G have collapsed/merged into one state. Notice the language is not altered.

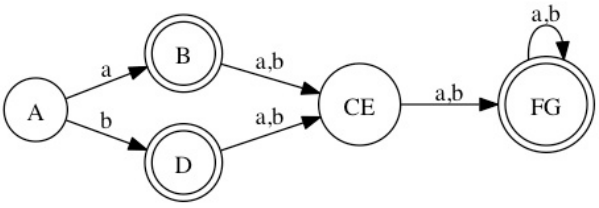


Figure 3 : States C and E have collapsed/merged into one state. Notice the language is not altered.

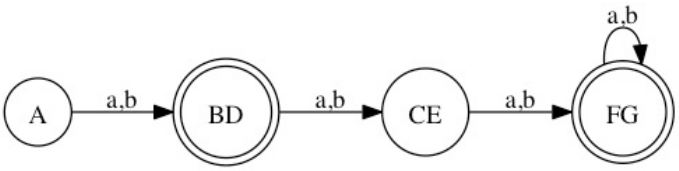


Figure 4 : States B and D have collapsed/merged into one state. Notice the language is not altered.

Figure 4 represents our final iteration through the algorithm as there are no more indistinguishable states. The DFA of Figure 4 is an equivalent to the DFA of Figure 1. Here is the Haskell representation of Figure 1:

```
dfa2 = [    ("A",0,1,"a","B"),("A",0,1,"b","D"),("B",1,0,"a","C"),("B",1,0,"b","E"),
            ("C",1,1,"a","F"),("C",1,1,"b","F"),("D",1,0,"a","E"),("D",1,0,"b","C"),
            ("E",1,1,"a","F"),("E",1,1,"b","F"),("F",1,0,"a","G"),("F",1,0,"b","F"),
            ("G",1,0,"a","G"),("G",1,0,"b","G")          ]
```

The details of this implementation can be observed in the addendum. Some important functions over this implementation include:

```
-- Create a list of states as tuples
pairs [] = []
pairs dfa = [ (x,y) | [x,y] <- subsequences $ states dfa ]


-- Distinguishable given pair of states, 1 is final, 1 is not
isDist (x,y) dfa
            | (isFinal x dfa) && (not $ isFinal y dfa) ||
              (isFinal y dfa) && (not $ isFinal x dfa)          = True
            | otherwise                                         = False


-- Propagate a list of distinguishable pairs
distinguishList dfa = [ (x,y) | (x,y) <- pairs dfa, isDist (x,y) dfa ]
```

This program produces the representation of figure 4. In ghci, running "main" will give the DFA of Figure 1 and Figure 4.

```
Undistinguishable State(s): [("E","C"),("F","G"),("B","D")]
Distinguishable State(s): ["A"]
Alphabet: ["a","b"]
Final States: [("F","G"),("B","D")]
```

Section 1.2 : Algorithm 6.2.2 (or Construction of a Regular Expression from a Finite State Machine) is a slightly different minimizing algorithm. Instead of merging states, it eliminates all intermediary nodes of an FSM (of n nodes) until only the start and final nodes exist. This minimized "expression graph", a particular FSM, can then be evaluated and used to construct a regular expression. Essentially the idea is the same as the DFA minimizer; take a convoluted FSM and prune it until some base case is reached, and then interpret the more readable FSM. I worked with example 6.2.1 from Sudkamp's book (p. 195-196), compiling a Haskell representa-

tion of this algorithm at work. My algebraic data type (ADT) was a list of states in the machine [a], a list of moves [Move a], and the designated start and final states (a -- q0, a -- qF).

```
data Fsm a = FSM [a] [Move a] a a deriving (Show, Eq, Ord)
data Move a = Move a String a deriving (Show, Eq, Ord)
```

Here is example 6.2.1 using my representation:

```
fsm2 = FSM [0 .. 3]
[ Move 0 “a” 2, Move 0 “a” 3, Move 1 “b” 0, Move 2 “b” 2, Move 2 “a” 1, Move 3 “b” 1, Move 3 “b” 0] 0 3
```

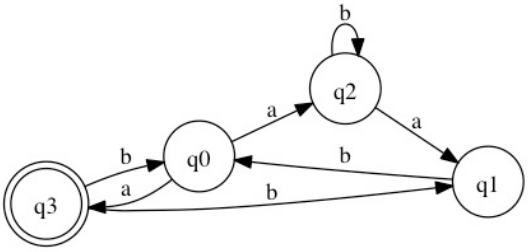


Figure 5 : Example 6.2.1 - An expression graph with an unrealizable regular expression.

As Figure 5 suggests, the presence of the intermediary nodes, q1 and q2 make interpreting this expression graph a hassle. Alg622 works to eliminate them. After the first iteration through the algorithm and the deletion of q1, the reduced expression graph looks like:

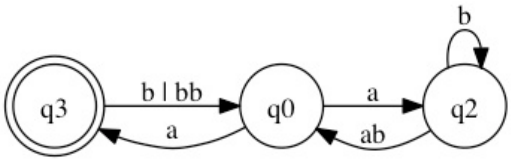


Figure 6 : Alg622 removes intermediary node q1.

The pruning of the expression graph does not alter the regular expression we'll ultimately end with, as we'll see with the output of our second and final iteration through the algorithm.

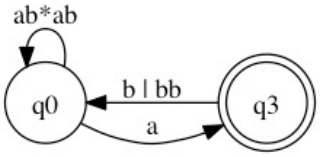


Figure 7 : Alg622 removes intermediary node q2 for a final representation of the expression graph.

Now that the expression graph is reduced, we may infer what regular expression this graph produces. In this case, the machine accepts the regular expression: (ab*ab)*(b∪bb)(a(ab*ab)*(b∪bb))*

I use a naming convention in my Haskell representation that correlates with the pseudo-algorithmic procedure from the text. The procedure is present in the addendum. The following step in the pseudo-code is replicated in Haskell.

iii) if nodes Qj and Qk have arcs labeled W1,W2,..,Ws connecting them, then replace the arcs by a single arc labeled W1 ∪ W2 ∪ .. ∪ Ws.

```
partIII i fsm@(FSM sts moves q0 qt) =
        replace $ (nub $ groupN newMoves newMoves)
                where
                newMoves = [ Move j s k | Move j s k <- partIandII i fsm, j /= i, k /= i ]
                replace [] = []
                replace (xs:xss) = merge xs : replace xss
                merge strings@(x:xs) = Move (getJ x) (getOr strings) (getK x)
```

The function "compile" recursively iterates through the intermediary nodes, Qi's, and works to accomplish the effect of pruning the tree. It is a common practice to pipeline the output of functions that pass entire data structures into new functions for further extrapolation. Notice the local definition of **newMoves** under the where clause uses a list comprehension, a structure that makes manipulating components of a list convenient, to propagate and filter "new moves". partIandII i fsm correlates, unsurprisingly, to parts i) and ii) of the pseudo-code in the text. Each instance of **Move j s k ← partIandII i fsm** is assembled into a new list and passed to a function called "replace", which takes care of the string manipulation from each arc. The output of running runStateT generator fsm2 (generator is a monadic transformer) yields:

```
Inputed Finite State Machine:
FSM [0,1,2,3]
[Move 0 "a" 2,Move 0 "a" 3,Move 1 "b" 0,Move 2 "b" 2,Move 2 "a" 1,Move 3 "b" 1,Move 3 "b" 0] 0 3

Resulting Finite State Machine:
FSM [0,3] [Move 0 "aabla(b)*ab" 0,Move 3 "bblb" 0,Move 0 "a" 3] 0 3

Regular Expression Generated:
(aabla(b)*ab)*a(bblb(aabla(b)*ab)*a)*
```

The program's output shows a slightly modified but correct representation of Figure 7's regular expression. Notice there's a legal string, aab, Or'd together with the string ab*ab that isn't in Figure 7's representation.

Section 1.3 : Breadth-First Search (BFS) is a fundamental graph traversal algorithm that most computer science students learn in discrete mathematics. I had an insatiable urge to program this algorithm using Haskell. BFS works by seeding a graph with an initial "root", or "start" node with a label 0 with no predecessor. From the root, label the nodes adjacent to it with k+1 and set their predecessors as the root. Iterate through the nodes in this fashion until there are no more nodes without a label connected to the graph whose predecessors connect them to the root. After the graph is labeled appropriately, various special functions can be run to determine details of its structure. The Dossey pseudo-code for this algorithm has "step 3" as constructing a short-

est path to a particular node. A shortest path is assembled by taking the input node and tracing its successive predecessors to the root node. The following figures were derived from Dossey's example 4.17 (pg 181):

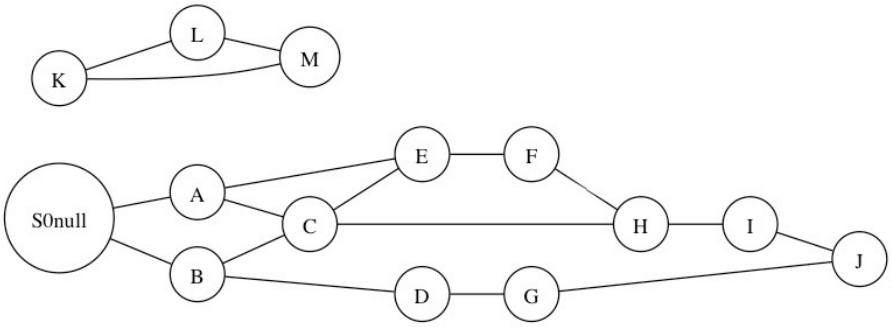


Figure 8 : An undirected graph with an initial start node S with label 0 and predecessor null.

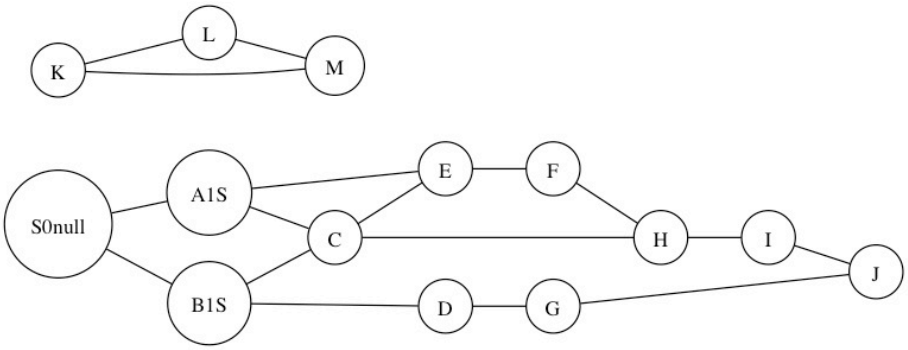


Figure 9 : Nodes A and B are labeled k+1 and their predecessors are set to S.

Because A and B are adjacent to S, their label is S's label + 1 (k+1) and their predecessor is set to S. The last figure gives the graph labeled in its entirety.

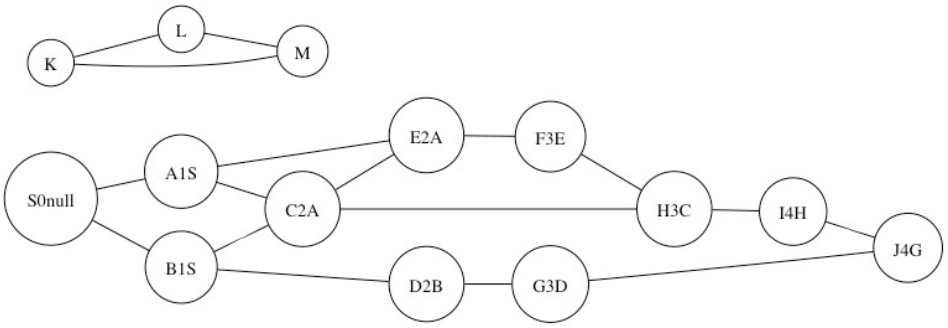


Figure 10 : Final graph after the termination of BFS

Nodes K, L, and M, although part of the graph, remain unlabeled as none of them connect to any of the nodes adjacent to S, A, B, C, D, E, F, G, H, I, or J. And so there is no shortest path to K, L, or M. It is worth mentioning that this algorithm does not give every shortest path possible. Although we may calculate a shortest path to C as {S, A, C}, there exists another path, {S, B,

C}, which is just as accurate. Because this algorithm represents a singular labeling system, the path {S,A,C} is the only shortest path given the predecessor of C, namely A.

The Haskell implementation of this program uses a single function called "propagate" that enlarges the graph's list of labels. Since step 1 involves only seeding the data structure with Label "S" (0,"-"), it is woven into the entire enlarging process of labelVs.

```
labelVs g@(G edges labels) = propagate lbs g
        where
        labelS = assign (Label "S" (0,"-")) g
        lbs = getLabels (labelS)
```

Propogate enlarges the graph's list of labels after the initial seed is set. Passing **lbs** to propagate assigns S (0,"-") to **g** and allows the processing to continue.

```
propagate [] graph = graph
propagate (Label v (k,pred):lbs) graph@(G edges labels) = do
    let gLabels = [ Label a (k+1,v) | a <- adjacencyList v graph, not $ hasLabel a graph, a /= "S" ]
    propagate (lbs ++ gLabels) (assignLabels gLabels graph)
```

One pass through propagate will demonstrate how this labeling system works. Initially our label list only contains [Label "S" (0,"-")]. Using the constructs of Haskell's list notation, we can view this list as Label "S" (0,"-"):[] - or the label S appended (:) on to the rest of the list (which happens to be empty for the moment ([])). **gLabels** generates a list of labels given the adjacency list of the vertex named in the label list (S). The recursive call to propagate passes the new labels appended to the empty list **(lbs ++ gLabels)** -- where lbs is empty -- and uses **assignLabels** to give the graph its new label list.

```
*Main> labelVs g1
    [Label "B" (1,"S"),Label "A" (1,"S"),Label "E" (2,"A"),Label "C" (2,"A"),
     Label "D" (2,"B"),Label "H" (3,"C"),Label "F" (3,"E"),Label "G" (3,"D"),
     Label "I" (4,"H"),Label "J" (4,"G")]
```

The final pass through propagate gives our final list of labels. It can inferred that S is already assigned which is why its presence is absent from the label list (for the moment). It is manually appended in the construction of the shortest path function. See addendum.

Section 1.4 : Translation of BFS from Haskell into Java. To mimic the algebraic data type I used in the Haskell implementation

```
data Graph a = G [(a,a)] [Label a] deriving (Eq, Ord, Show, Read)
data Label a = Label a (Int,a) deriving (Eq, Ord, Show, Read)
```

I used three separate classes. A Label class that constructed a label object with a name, label, and predecessor; an Edge class that constructed an edge object with a start and finish node and a Graph class that initialized an ArrayList of Label and an ArrayList of Edge. The BFS class rep-

resented the main algorithmic logic component to the Java implementation. Here is a snipped of BFS.java. I initialize a graph object with the appropriate edges (from example 4.17) and an empty label ArrayList, seed the graph's label with Label("S",0,"-") and the label ArrayList with "S".

```
// Step 1 (a) : Assign S the label 0, and let S have no predecessor.
    graph.assign(new Label("S",0,"-"));
// Step 1 (b) : Set L = {S}
    vertLabels.add("S");
 // Step 2 : enlarge labeling
    graph.enlarge(vertLabels, graph);
```

The method enlarge is equivalent to the propagate function in Haskell. The methods to work with graph are, unsurprisingly, located in Graph.java.

```
public Graph enlarge(ArrayList<String> vls, Graph g) {
    ArrayList<String> adj;
    ArrayList<String> newAdj = new ArrayList<>();
    String pred;
      for(String v : vls) {
        newAdj.addAll(g.adjacencyList(v));
      }
      if (g.allLabeled(newAdj))
        return g;
      else
        for(String v : vls) {
          adj = g.adjacencyList(v);
          pred = v;
          for(String s : adj)
            if (!g.hasLabel(s) && !s.equals("S"))
            g.assign(new Label (s,(g.getLabel(pred) + 1),v));
    }
    return enlarge(newAdj, g);
```

I initialize two ArrayLists and a predecessor variable, **pred**, by which I add all of the adjacent nodes using the for-each control structure to the **newAdj** ArrayList. I use a boolean checker, **allLabeled** as my base case for returning the graph. The method then assigns new labels for each string adjacent to the vertex its evaluating (starting with S). Here is an example of an output:

```
Graph Labels: [Label S (0,-), Label A (1,S), Label B (1,S), Label E (2,A), Label C (2,A),
Label D (2,B), Label F (3,E), Label H (3,C), Label G (3,D), Label I (4,H), Label J (4,G)]
```

Section 2 and 3 : Analysis and Reflection -- Admittedly, I have a stronger background programming in the functional paradigmatic style and so consequently my skills using the imperative, object oriented style are reflected in my Java implementation of BFS. Because I'm used to programming with functions that do not alter some "global" state, I tend to pass functions that iteratively alter, for example, an incremental or decremental value in a recursive call. This is the reason that the labeling routine in **enlarge** assigns a new label as its predecessor + 1 and is altered

(incremented) every time the function is recursively called instead of instantiating a variable k in a global state and then incrementing it every time a method is called.

```
Haskell:    gLabels = [ Label a (k+1,v) | a <- adjacencyList v graph, not $ hasLabel a graph, a /= "S" ]
            propagate (lbs ++ gLabels) (assignLabels gLabels graph)

Java:       g.assign(new Label (s,(g.getLabel(pred) + 1),v));
            return enlarge(newAdj, g);
```

Comparing the recursive calls for both implementations reveals how similar in style I modeled the Java version. I pass a new adjacency list of vertices to be examined along with a new graph with labels assigned to it. The Haskell version passes new labels appended to be evaluated **(lbs ++ gLabels)** plus a new graph **(assignLabels gLabels graph)**.

A major difference I observed programming the Java representation was the need to define many accessor methods. Because each classes' instance variable is sheathed in its allocation, we must define a series of methods for accessing them. The patterns in Haskell eliminate this need by allowing the user to define mathematical, piecewise style functions. This dichotomy between paradigms can be observed in the following example

```
Haskell:    hasLabel x (G edges labels) = any (==x) [ a | Label a (lb,pred) <- labels ]

Java:       public Boolean hasLabel(String x) {
                    Boolean value = false;
                    for(Label l : labels){
                            if (x.equals(l.getName()))
                            value = true;
                     }
                    return value;
```

Both of these functions represent testing a string x against a list of labels. Because we pass the ADT of the graph representation in the Haskell version, we're able to pattern match the list of labels within the list comprehension structure on the right side of the equation. And because Haskell supports so many native operations over the list structure, we can invoke list operations that are already predefined in the prelude. **any (==x)** does, essentially, what the Java description of **hasLabel** does, except more generally for all things list related. In Java, this method must be specifically catered for its implementation. **hasLabel** assumes you're dealing with a Graph object, as it takes a for-each structure meant for an ArrayList of labels and tests whether or not x equals a label's name (which I must invoke the **getName()** method to access). Conversely, I can define a function called **any** to test a predicate on a list of any-thing (pardon the pun). The following example will work on a list of Strings, Ints, a graph algebraic data type, or anything else that can be tested for equality! The type variable system allows for easy coercion.

```
any :: (a -> bool) -> [a] -> Bool
any p [] = False
any p (x:xs)
        | p x           = True
        | otherwise     = any p xs
```

*** A recursive function for testing a predicate against the elements in a list**

Here the predicate, p, represents a function tested against a piece of the list, namely x from **(x:xs)**. In **any (==x) [a | Label a (lb,pred) <- labels]** pass the partially applied function **(==x)** to each element in a list of names, **a**, generated by the labels in the ADT. If any of the label's names match the string being tested (x), the resulting value is True, otherwise False. Throwing around pieces of a data structure allows for an intuitive experience in defining functions. I find that the procedural, object oriented style clouds the overall intent of a program by forcing the programmer to define specific functionality for every part of a data structure.

This being said, I find that it is a much more intuitive experience using a language with mutable state when things like random number generation are involved, or the encoding of a GUI. These are things the functional paradigm have a more obscure and less readable solution to. Often functional languages have no "state" and therefore no side effects. I have yet to determine if being able to deterministically alter the state of something and risking inadvertently altering a state is worth the risk. I can see how its useful but I can also see how it may be detrimental to the development of software. There are many times where I see examples of object oriented code that may be developed using more general, reusable, and less verbose code. After a cursory search, the only implementations of this algorithm I found using Java involved using the Queue and LinkedList interfaces:

```
public void bfs()
{
        //BFS uses Queue data structure
        Queue q=new LinkedList();
        q.add(this.rootNode);
        printNode(this.rootNode);
        rootNode.visited=true;
        while(!q.isEmpty())
        {
                Node n=(Node)q.remove();
                Node child=null;
                while((child=getUnvisitedChildNode(n))!=null)
                {
                        child.visited=true;
                        printNode(child);
                        q.add(child);
                }
        }
//Clear visited property of nodes
```

```
        clearNodes(); }
```

*** Alternate BFS implementation. Source:**
http://www.codeproject.com/Articles/32212/Introduction-to-Graph-with-Breadth-First-Search-BF

This involves using the queue structure to systematically add a node's children (adjacent nodes) to a linked list. This small piece of code represents the algorithm correctly but has a less than desirable labeling system. This implementation will not produce a coherent representation of what's been labeled through-out its structure. It will also give no hint as to what sort of priority a node's label may have been assigned to it and why.

The goal of this project was to keep an objective mind whilst coding alternate versions of specific algorithms defined in computer science. It was enormously helpful exercise to write the same implementations of a graph traversal algorithm in Haskell and Java. In conclusion, I found that no specific paradigm is better at addressing all problems, but rather, each paradigm has specific language features useful in addressing particular problems but may lack the intuitive strength in others. I found the class organization aspect of Java useful but verbose, and Haskell list functionality efficient but less readable. The readability can become gray on both ends of the stick, though, as using the more abstract functionality of a language's features with a compact naming scheme can obfuscate meaning whilst long, drawn out procedural instruction divided across multiple class files can do the same.

I had an ambitious goal of thoroughly developing 5-8 programs drafted from selected algorithms in Sudkamp/Dossey's text that I was interested in but soon realized the scope of the project's duration was limiting. I only had time to translate one program from Haskell into Java. It would be interesting to develop a series of alternate graph traversal algorithms first from Java and then Haskell and observe if their style had any influence on their developed outcome in Haskell. This project has helped reignite my interest in graph theory and implementing search algorithms over data structures. The following pages give my works cited and the lines of code I wrote over this last quarter. Thanks for reading!

Works Cited

Dossey, John A. *Discrete Mathematics*. Glenview, IL: Scott, Foresman, 1987. Print.

"Introduction to Graph with Breadth First Search(BFS) and Depth First Search(DFS) Traversal Implemented in JAVA." - *CodeProject*. N.p., n.d. Web. 28 May 2013.

Lipovača, Miran. *Learn You a Haskell for Great Good!: A Beginner's Guide*. San Francisco, CA: No Starch, 2012. Print.

Sudkamp, Thomas A. *Languages and Machines: An Introduction to the Theory of Computer Science*. Reading, Mass [u.a.: Addison-Wesley, 1997. Print.

Thompson, Simon. *Haskell: The Craft of Functional Programming*. Harlow, Eng.: Addison Wesley, 1999. Print.

Thompson, Simon. "Regular Expressions and Automata Using Haskell." *University of Kent at Canterbury*. N.p., n.d. Web. <http://kar.kent.ac.uk/22057/2/Regular_Expressions_and_Automata_using_Haskell.pdf>.

Addendum

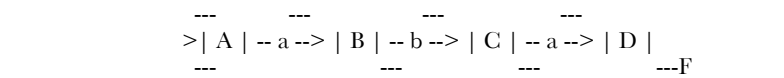
```
-- DFA Minimization
-- Kurt Medley
-- April 2013
--------------------------------
A deterministic finite automaton (DFA) —also known as deterministic finite state machine— is a finite state machine that
accepts/rejects finite strings of symbols and only produces a unique computation (or run) of the automaton for each input string.
--Hopcroft, John E.; Motwani, Rajeev; Ullman, Jeffrey D. (2001). Introduction to Automata Theory, Languages, and Computation (2 ed.). Addison Wesley. ISBN 0-201-44124-1.
--------------------------------
This program presents a schema for the state diagram generated by the provided DFA-M.
Let DFA M = (Q,E,D,q0,F) where
        Q  = {States},
        E  = {Alphabet},
        D  = {Delta Transitions}
        q0 = Start State
        F  = Final State
--------------------------------
A DFA-M can be inferred by a list of transitions given by the quintuple ((x1,x2,x3,x4,x5)).
 x1 - Current state <Q> 0 yes, 1 no
 x2 - Start state? <q0>
 x3 - Is this a final state? <F> 0 yes, 1 no
 x4 - Input symbol <E>
 x5 - End in state <Q>
--------------------------------

> import Data.List (nub, subsequences)
> type Transition = (String, Int, Int, String, String)

> example :: Transition
> example = ("A",0,1,"a","B")
> -- Start state; Beginning state A, processing input symbol a, end in state B; Not a final state.
```

```
> type DFA = [Transition]
```

Example state diagram processing the string aba.




```
> dfa1 :: DFA
> dfa1 = [ ("A",0,1,"a","B"),
>          ("B",1,1,"b","C"),
>          ("C",1,1,"a","D"),
>          ("D",1,0,"lam","D") ]

> dfa2 :: DFA
> dfa2 = [ ("A",0,1,"a","B"),("A",0,1,"b","D"),
>          ("B",1,0,"a","C"),("B",1,0,"b","E"),
>          ("C",1,1,"a","F"),("C",1,1,"b","F"),
>          ("D",1,0,"a","E"),("D",1,0,"b","C"),
>          ("E",1,1,"a","F"),("E",1,1,"b","F"),
>          ("F",1,0,"a","G"),("F",1,0,"b","F"),
>          ("G",1,0,"a","G"),("G",1,0,"b","G") ]
```

* lam = lambda, or the empty string. These will only be present in final states.

Here we can infer the specifics of the given DFA-M by the transition table..
Qi: { A,B,C,D }
E: { a,b }
q0:{ A }
F: { D }

```
> main = do
>          putStrLn $ "\n" ++ show (dfa2) ++ "\n"
>          putStrLn $ "\n" ++ "Original DFA: " ++ "\n"
>          showDFA dfa2
>          putStrLn $ "\n" ++ "New DFA: " ++ "\n"
>          newDFA dfa2
```

The first step in DFA minimization is generating a distinguishable list. A 'distinguish list' compares every pair of states. A pair is distinguishable iff one state is final, and the other is not. Example (A,D),(B,D),(C,D) are distinguishable states whereas (A,B),(B,C),(D,D) are not disguishable.. yet. So, the first step is finding the states of the DFA-M. Constructing a series of 'lookup' functions that will associate states with their desired traits..

```
> -- DFA boolean tests
> isStart :: String -> DFA -> Bool
> isStart state ((a,b,c,d,e):xs)
>          | (state == a) && (b == 0)     = True
>          | otherwise                                = isStart state xs
> isStart state []                        = False

> isFinal :: String -> DFA -> Bool
> isFinal state ((a,b,c,d,e):xs)
>          | (state == a) && (c == 0)     = True
>          | otherwise                                = isFinal state xs
> isFinal state []                        = False

> -- DFA characteristical lookup functions
> states :: DFA -> [String]
> states [] = []
> states ((a,b,c,d,e):xs) = nub $ a:e:states xs
```

```
> startState :: DFA -> [String]
> startState [] = []
> startState ((a,b,c,d,e):xs)
>             | b == 0  = nub $ a : startState xs
>             | otherwise             = startState xs

> finalStates :: DFA -> [String]
> finalStates [] = []
> finalStates ((a,b,c,d,e):xs)
>             | c == 0  = nub $ a : finalStates xs
>             | otherwise             = finalStates xs

> alphabet :: DFA -> [String]
> alphabet [] = []
> alphabet ((a,b,c,d,e):xs) = nub $ d : alphabet xs

> -- Transition to: Gives the ending state of a state and its input
> transTo :: String -> String -> DFA -> String
> transTo start sym ((a,b,c,d,e):xs)
>             | start == a && sym == d     = e
>             | otherwise                                 = transTo start sym xs
> transTo start sym [] = error "Input symbol does not exist."

> showDFA dfa = do
>             putStrLn $              "States: " ++ show (states dfa) ++ "\n" ++
>                                     "Start State: " ++ show (startState dfa) ++ "\n" ++
>                                     "Final States: " ++ show (finalStates dfa) ++ "\n" ++
>                                     "Alphabet: " ++ show (alphabet dfa)
> --------------------------------
```

And now, creating the subsequences of states to create a "distinguish list".

```
> -- Create a list of states as tuples
> pairs [] = []
> pairs dfa = [ (x,y) | [x,y] <- subsequences $ states dfa ]

> -- Distinguishable given pair of states, 1 is final, 1 is not
> isDist (x,y) dfa
>             | (isFinal x dfa) && (not $ isFinal y dfa) | |
>              (isFinal y dfa) && (not $ isFinal x dfa)  = True
>             | otherwise                                                 = False

> -- Propegate a list of distinguishable pairs
> distinguishList dfa = [ (x,y) | (x,y) <- pairs dfa, isDist (x,y) dfa ]
```

Create a dependancy list from pairs that aren't distinguishable

```
> nonDistinguished dfa = [ (x,y) | (x,y) <- pairs dfa, not $ isDist (x,y) dfa ]

> -- Input symbol list, nondistinguished list, dfa
> dependancyList1 dfa = [ ((transTo x z dfa),(transTo y z dfa)) |
>                                     (x,y) <- nonDistinguished dfa, z <- alphabet dfa ]

> dependancyList2 dfa = [ ((transTo y z dfa), (transTo x z dfa)) |
>                                     (x,y) <- nonDistinguished dfa, z <- alphabet dfa ]

> -- finalDep combines all possible pairs of states that represent the dependancy list.
> finalDep dfa = (dependancyList1 dfa) ++ (dependancyList2 dfa)
```

We need a function that will take each element from the non-distinguished list and examine if its an element of the dependancy list. If its true, add it to the distinguished list.

```
> -- if a pair's dependent pair is itself or a pair of matching states, the resulting pair is NOT distinguishable.
```

mark elements from the nondistinguished list whose dependency pairs are members of the distinguished list.

```
> mark [] dfa = []
> mark (depx:depxs) dfa
>         | depx `elem` distinguishList dfa         = depx : mark depxs dfa
>         | otherwise                                = mark depxs dfa

> itself (x,y) (a:as) dfa
>         |         ((transTo x a dfa) == x && (transTo y a dfa) == y)
>                   || (transTo x a dfa) == (transTo y a dfa)         = True
>         | otherwise = itself (x,y) as dfa
> itself (x,y) [] dfa = False
```

mark elements from the nondistinguished list whose dependency pairs are members of the distinguished list

```
> congeal [] dependL dfa      = []
> congeal (x:xs) dependL dfa
>         | x `elem` dependL && not (itself (fst x, snd x) (alphabet dfa) dfa)
>                                   = x:(congeal xs (x:dependL) dfa)
>         | otherwise               = congeal xs dependL dfa

> test dfa = mark (distinguishList dfa) dfa ++ congeal (nonDistinguished dfa) (finalDep dfa) dfa

> comb dfa = congeal (test dfa) (finalDep dfa) dfa

> dependsOn x dfa = [ ((transTo (fst x) i dfa), (transTo (snd x) i dfa)) | i <- alphabet dfa ]

> collapse (x:xs) ys
>         | x `elem` ys        = collapse xs ys
>         | otherwise          = x : collapse xs ys
> collapse [] ys = []

> collapse1 dfa = [ (x,y) | i <- nonDistinguished dfa, (x,y) <- dependsOn i dfa, not (elem (x,y) (comb dfa)) && not (elem (x,y) (distinguishList dfa)) ]

> remD (x:xs)
>         | (fst x) == (snd x)   = remD xs
>         | otherwise                 = x : remD xs
> remD [] = []

> remPairs [] = []
> remPairs (y:[]) = [y]
> remPairs (x:y:xs)
>         | (fst x == fst y) || (fst x == snd y) || (snd x == fst y) || (snd x == snd y)
>                              = remPairs (y:xs)
>         | otherwise          = x : remPairs (y:xs)

> collapse2 dfa =
>         [ i | i <- nonDistinguished dfa, (x,y) <- dependsOn i dfa, elem (x,y) (remD $ collapse1 dfa) ]

> finalCollapse dfa = remPairs $ nub $ remD $ (collapse1 dfa) ++ (collapse2 dfa)
```

----
finalCollapse produces the list of collapsable, undistinguished states. What's left is all for aesthetic purposes. Everything else about the dfa can be inferred deductively. States (E,C), (F,G), and (B,D) are generated; Each state within the pair has its deterministic transition to a different state. For example, E is neither a start or final state, on an input, "a" or "b", the machine transitions to "F"; However, we observe that F is undistinguishable from G; Now examining C, on inputs "a" and "b", the machine transitions to F. So we say within states E or C, on inputs "a" and "b", the machine transitions to the state (F,G) -- (Remembering that F is undistinguishable from G). Where is A? A is a state by itself that transitions to the undistinguishable states B and D; this is another pairwise state (B,D).

```
                                                                   ------
 ---              -------             -------        ------- | a,b
| A | - a,b -> | (B,D) | - a,b -> | (E,C) | - a,b -> | (F,G) | <-
 ---              -------F      -------       -------F
```

Thus we observe that dfa2 is a FSM that accepts the regular language:
a U b (a U b (a U b)+)?
ex strings accepted: a, b, aaa, aba, babababababa, aaababababa
---

```
> newStates :: Eq a => [a] -> [(a,a)] -> [a]
> newStates (x:xs) tups
>         | elem x (flattenTup tups)      = newStates xs tups
>         | otherwise                             = x : newStates xs tups
> newStates [] tups = []

> flattenTup :: [(a,a)] -> [a]
> flattenTup [] = []
> flattenTup (x:xs) = fst x : snd x : flattenTup xs

> newFinalStates (x:xs) dfa
>         | isFinal (fst x) dfa   = x : newFinalStates xs dfa
>         | otherwise                     = newFinalStates xs dfa
> newFinalStates [] dfa = []

>-- newDFA :: DFA -> IO ()
> newDFA dfa = do
>         putStrLn $ "Undistinguishable State(s): " ++ show (finalCollapse dfa) ++ "\n" ++
>                    "Distinguishable State(s): " ++ show (newStates (states dfa) (finalCollapse dfa)) ++ "\n" ++ "Alphabet: " ++
show (alphabet dfa) ++ "\n" ++ "Final States: " ++ show (newFinalStates (finalCollapse dfa) dfa)
```

-- Construction of a Regular Expression from a Finite Automaton
-- (c) Kurt Medley
-- April 2013

From Thomas Sudkamp's Languages and Machines (pgs 193-194):
An expression graph is a labeled directed graph in which the arcs are labeled by regular expressions. An expression graph, like a state diagram, contains a distinguished start node and a set of accepting nodes. A procedure is developed to reduce an arbitrary expression graph to an expression graph containing at most two nodes. The reduction is accomplished by repeatedly removing nodes from the graph in a manner that preserves the language of the graph.

Algorithm 6.2.2

input: state diagram G of a finite automaton with one accepting state

Let Q0 be the start state and Qt the accepting state of G.
1. repeat
    1.1 choose a node Qi that is neither Q0 nor Qt
    1.2 delete the node Qi from G according to the following procedure
        1.2.1 for every j,k not equal to i (this includes j=k) do
            i) if Wj,i /= 0, Wi,k /= 0 and Wi,i = 0, then add an
                arc from node j to node k labeled Wj,iWi,k
            ii) if Wj,i /= 0, Wi,k /= 0 and Wi,i /= 0 then add
                an arc from node Qj to node Qk labeled
                Wj,i(Wi,i)*Wi,k
            iii) if nodes Qj and Qk have arcs labeled W1,W2,..,Ws
                connecting them, then replace the arcs by
                a single arc labeled W1 U W2 U .. U Ws
        1.2.2 remove the node Qi and all arcs incident to it in G
            until the only nodes in G are Q0 and Qt
2. determine the expression accepted by G

```
> import Control.Monad.State
> import Data.List (nub, nubBy, permutations, union, groupBy, partition, intersperse)

> data Fsm a =      FSM [a]             -- States
>                   [Move a]                    -- Transitions
```

```
>                       a                       -- Q0
>                       a                       -- Qt
>                   deriving (Show, Eq, Ord)

> data Move a = Move a String a deriving (Ord, Show, Eq)

> type FsmT = StateT (Fsm Int) IO

-----------------------------------------
-- A monadic transformer to minimize the FSM and generate
-- the regular expression. ex. run: runStateT <func> fsm
-----------------------------------------

> generator :: FsmT ()
> generator = do
>       fsm <- get
>       liftIO $ putStrLn $ "\n" ++ "Inputed Finite State Machine: " ++ "\n" ++ (show fsm) ++ "\n"
>       put $ compile (qi fsm) fsm
>       newfsm@(FSM sts' moves q0 qt) <- get
>       liftIO $ putStrLn $ "Resulting Finite State Machine: " ++ "\n" ++ show (FSM (newStates newfsm) moves q0 qt) ++
"\n"
>       liftIO $ putStrLn $ "Regular Expression Generated: " ++ "\n" ++ (regexp newfsm) ++ "\n"

-----------------------------------------
-- Final Or'ing together of the transitions and generation
-- of the regular expression
-----------------------------------------

> regexp (FSM sts moves q0 qt) = (process startloops) ++
>                                (connect startfinish) ++
>                                (process finishloops) ++
>                                (doesExist finishstart)
>       where
>       startloops  =       [ Move x y z | Move x y z <- moves, Move x y z == Move q0 y q0 ]
>       finishloops =       [ Move x y z | Move x y z <- moves, Move x y z == Move qt y qt ]
>       startfinish =       [ Move x y z | Move x y z <- moves, Move x y z == Move q0 y qt ]
>       finishstart =       [ Move x y z | Move x y z <- moves, Move x y z == Move qt y q0 ]
>       process loops
>               | length loops == 1 = "(" ++ head [ y | Move x y z <- loops ] ++ ")" ++ "*"
>               | length loops > 1  = "(" ++ (init $ foldr (++) [] [ s ++ "|" | Move x s y <- loops ]) ++ ")" ++ "*"
>               | otherwise                 = []
>       -- base case for connect does not prepend and append ( ) * because this arc doesn't connect its current and end nodes.

>       connect loops
>               | length loops == 1         = head [ y | Move x y z <- loops ]
>               | length loops > 1  = "(" ++ (init $ foldr (++) [] [ s ++ "|" | Move x s y <- loops  ] ) ++ ")"
>               | otherwise                 = []
>
>       doesExist loops
>               | length loops >= 1
>               = "(" ++ connect finishstart ++ process startloops ++ connect startfinish ++ process finishloops ++ ")" ++
"*"
>               | otherwise                 = []

-----------------------------------------
-- Compile: where the magic happens.
-- Recursively call partIII (which works on deleting
-- intermediate transitions) until only q0 and qt remain.
-- ex usage: compile (qi fsm1) fsm1
-----------------------------------------

> compile [] fsm = fsm
> compile (i:is) fsm@(FSM sts moves q0 qt) = compile is (FSM sts newMoves q0 qt)
>               where
```

```
>                       newMoves            = partIII i fsm

-----------------------------------------
 -- partIandII and partIII correlate to the algorithm above;
 -- These functions are piped together in 'compile' and reduce
 -- a FSM G to a minimal state, namely Q = [ q0, qt ]
-----------------------------------------

> partIandII i fsm@(FSM sts moves q0 qt)
>           = addarcs' (nub $ funci newArcs fsm) moves
>                       where
>                       newArcs =
>                                   [ [Move j set1 i,Move i set2 i,Move i set3 k] |
>                                               j <- sts, k <- sts, j /= i, k /= i,
>                                               Move a set1 c <- moves, -- Get Strings from trans
>                                               Move q set2 w <- moves,
>                                               Move t set3 u <- moves,
>                                               Move j set1 i `elem` moves
>                                               && Move i set3 k `elem` moves ]
>                                               -- Filter all triples [x,y,z]'s whose
>                                               -- j,i i,k exist within moves otherwise no arc exists


> funci ([x,y,z] : xyzs) fsm@(FSM sts moves start finish)
>           -- i)
>           | x `elem` moves && not (y `elem` moves) && z `elem` moves
>                       = (Move (getJ x) (getStr x ++ getStr z) (getK z))
>                                   : funci xyzs fsm
>           -- ii)
>           | x `elem` moves && y `elem` moves && z `elem` moves
>                       = (Move (getJ x) ( getStr x ++ "(" ++ getStr y ++ ")" ++ "*"
>                                   ++ getStr z) (getK z)) : funci xyzs fsm
>           | otherwise =           funci xyzs fsm
> funci [] fsm = []
>
> partIII i fsm@(FSM sts moves q0 qt) =
>                       -- iii)
>                       replace $ (nub $ groupN newMoves newMoves)
>                                   where
>                                   newMoves = [ Move j s k | Move j s k <- partIandII i fsm,
>                                                                                   j /= i, k /= i ]
>                                   replace [] = []
>                                   replace (xs:xss) = merge xs : replace xss
>                                   merge strings@(x:xs) = Move (getJ x) (getOr strings) (getK x)

-----------------------------------------
-- Auxillary Functions
-----------------------------------------

> getOr [] = []
> getOr ms = (init $ foldr (++) [] [ y ++ "|" | Move x y z <- ms ] )

> -- group takes a list of moves and sorts them by eqArc; ex. [Move 1 "a" 2, Move 1 "bb" 2, Move 0 "a" 1, Move 0 "ab" 1] = [
[Move 1 "a" 2, Move 1 "bb" 2],[Move 0 "a" 1, Move 0 "ab" 1] ] There is a bug here that may need to be addressed.  fst partition
produces the correct groupings with already used arcs.  May add unncessary but correct arcs..
> group [] = []
> group (x:xs) = aux x (x:xs) : group xs
>           where
>           aux x = (\xs -> fst $ partition (eqArc x) xs)

> groupN [] _ = []
> groupN ((Move x y z):xs) mvs =
>           (fst $ partition (\m -> eqArc m (Move x y z)) mvs)
>                       : groupN xs mvs
```

```
> min' mvs = minimum $ concat [ [a] ++ [c] | Move a b c <- mvs ]
> max' mvs = maximum $ concat [ [a] ++ [c] | Move a b c <- mvs ]

> eqArc (Move a b c) (Move x y z) = a == x && c == z
> getStr (Move x y z) = y
> getJ (Move x y z) = x
> getK (Move x y z) = z
> getAlpha (FSM sts [] q0 qt)          = []
> getAlpha (FSM sts (x:xs) q0 qt)      = getStr x : getAlpha (FSM sts xs q0 qt)

> -- Generate the nodes to be deleted
> qi (FSM states moves q0 qt) = [ sts | sts <- states, sts /= q0, sts /= qt ]

> -- Add an arc to a provided FSM, produce a new FSM with added arc
> addarc x (FSM sts moves q0 qt) = FSM sts (x:moves) q0 qt

> addarcs [] fsm@(FSM sts moves q0 qt) = fsm
> addarcs (x:xs) (FSM sts moves q0 qt) = addarcs xs (FSM sts (x:moves) q0 qt)

> addarcs' [] ys = ys
> addarcs' (x:xs) ys = x : addarcs' xs ys

> -- Delete the states of the machine; only the start and finish states remain
> newStates (FSM sts moves q0 qt) = [ states | states <- [q0,qt] ]

-----------------------------------------
-- Test FSMs
-----------------------------------------

> fsm1 :: Fsm Int
> fsm1 = FSM [0..3] [ Move 0 "a" 1, Move 1 "b" 2, Move 2 "a" 3 ] 0 3

> fsm2 :: Fsm Int
> fsm2 = FSM [0..3] [ Move 0 "a" 2, Move 0 "a" 3, Move 1 "b" 0, Move 2 "b" 2, Move 2 "a" 1, Move 3 "b" 1, Move 3 "b" 0] 0 3

> fsm3 :: Fsm Int -- From Sudkamp's example 6.3.4 DFA M
> fsm3 = FSM [0..3] [ Move 0 "a" 1, Move 0 "b" 3, Move 1 "b" 2, Move 1 "b" 0, Move 2 "b" 1, Move 2 "a" 3, Move 3 "a" 2,
Move 3 "b" 0] 0 3

-- Breadth-First Search
-- (c) Kurt Medley
-- May 2013
-------------------------------------
Discrete Mathematics 5th edition by Dossey et al. (pg 183)
"This algorithm determines the distance and a shortest path in a graph from vertex S to every other vertex for which there is a
path from S.  In the algorithm, L denotes the set of labeled vertices, and the predecessor of vertex A is a vertex in L that is used in
labeling A.

Step 1     (label S)
                              (a) Assign S the label 0, and let S have no predecessor.
                              (b) Set L = {S} and k = 0.
Step 2     (label vertices)
                    repeat
                              Step 2.1 (increase the label)
                                        Replace k with k+1
                              Step 2.2 (enlarge labeling)
                                        while L contains a vertex V with label k-1 that is adjacent to a vertex W not in L
                                                  (a) Assign the label k to W.
                                                  (b) Assign V to be the predecessor of W.
                                                  (c) Include W in L
                                        endwhile
                    until no vertex in L is adjacent to a vertex not in L
Step 3     (construct a shortest path to a vertex)
```

```
                        if a vertex T is in L
                                The label on T is its distance from S. A shortest path from S to T is formed by taking
                                in reverse order T, the predecessor of T, the predecessor of the predecessor of T,
                                and so forth, until S is reached.
                        otherwise
                                There is no path from S to T.
                        endif
---------------------------------------

> import Control.Monad.State
> import Data.List

> data Graph a = G [(a,a)]    -- Edges
>                  [Label a]                    -- Vertex label and predecessor [Label "S" (0,"-")]
>                  deriving (Eq, Ord, Show, Read)

> data Label a = Label a (Int,a) deriving (Eq, Ord, Show, Read)

> type GraphT = StateT (Graph [Char]) IO

---------------------------------------
-- Monadic BFS Runner
---------------------------------------

> bfsRunner = runStateT bfsRunT g1

> bfsRunT :: GraphT ()
> bfsRunT = do
>           graph <- get
>           put $ labelVs graph
>           liftIO $ putStrLn $ "Shortest path to which vertex?" ++ "\n" ++ show (vertices graph)
>           vertex <- liftIO $ getLine
>           graph <- get
>           if vertex `notElem` (vertLabels graph)
>                           then do error $ "Vertex is not labeled and is therefore not connected " ++
>                                                                   "to root S or Vertex does not exist. Case Sensistive." ++
"\n"
>                                                                   ++ "\n" ++ "Labeled vertices: " ++ show (vertLabels
graph)
>                           else liftIO $ putStrLn $
>                                   "\n" ++ "Shortest path to " ++ vertex ++ " is: " ++
>                                   show (construct vertex graph) ++ "\n"

---------------------------------------
-- Steps 1 and 2: Label S and Vertices
---------------------------------------

> labelVs g@(G edges labels) = propagate lbs g
>           where
>           labelS = assign (Label "S" (0,"-")) g
>           lbs = getLabels (labelS)

---------------------------------------
-- Step 3: Construct a Shortest Path
---------------------------------------

> construct v g@(G edges labels) = reverse $ (v : shortestPath v g)

> shortestPath v g@(G edges labels)
>           | v `elem` vertLabels g && v /= "S"      = predecessor v g : shortestPath (predecessor v g) g
>           | otherwise                                         = []

---------------------------------------
-- Auxillary Functions
```

```
---------------------------------------

> -- propogate generates the list of labels associated with step 2
> propagate [] graph = graph
> propagate (Label v (k,pred):lbs) graph@(G edges labels) = do
>         let gLabels = [ Label a (k+1,v) | a <- adjacencyList v graph, not $ hasLabel a graph, a /= "S" ]
>         propagate (lbs ++ gLabels) (assignLabels gLabels graph)

> -- return the list of labels in a graph; [Label x (y,z)]
> getLabels (G edges labels) = labels

> -- List of vertices with labels; Label x (y,z) -> [x]
> vertLabels (G edges []) = []
> vertLabels (G edges (Label x (y,z):labels)) = x : vertLabels (G edges labels)

> -- assign a label to a vertex: Label v (<int>, <pred>)
> assign l (G edges labels) = G edges (labels ++ [l])

> -- assign a list of labels to a graph
> assignLabels [] graph = graph
> assignLabels (x:xs) graph = assign x (assignLabels xs graph)

> -- give the list of vertices in a graph
> vertices :: Eq a => Graph a -> [a]
> vertices (G edges labels) = nub [ e1 | (e1,e2) <- edges ]

> -- give the list of edges in a graph
> edges :: Eq a => Graph a -> [(a,a)]
> edges (G edges labels) = edges

> -- tests if two vertices are adjacent
> isAdj x y edges
>         | (x,y) `elem` edges || (y,x) `elem` edges          = True
>         | otherwise                                                                 = False

> -- adjacencyList gives the list of vertices adjacent to x
> adjacencyList x graph@(G edges labels) = [ y | y <- vertices graph, isAdj x y edges ]

> -- predecessor returns the predecessor of a given vertex
> predecessor x (G edges []) = []
> predecessor x (G edges (Label a (k,pred):labels))
>         | x == a  = pred
>         | otherwise          = predecessor x (G edges labels)

> -- hasLabel tests if a vertex "x" already has a label assigned to it
> hasLabel x (G edges labels) = any (==x) [ a | Label a (lb,pred) <- labels ]

> -- k returns the label (integer) of a given vertex
> k x (G edges []) = error "No label associated with vertex"
> k x (G edges (Label a (y,pred):labels))
>         | x == a  = y
>         | otherwise          = k x (G edges labels)

---------------------------------------
-- Test Graphs
---------------------------------------

> g1 :: Graph [Char]
> g1 = G [        ("S","A"),("S","B"),("A","E"),("A","C"),
>                 ("A","S"),("B","S"),("B","C"),("B","D"),
>                 ("C","E"),("C","H"),("D","G"),("E","C"),
>                 ("E","F"),("F","E"),("F","H"),("G","J"),
>                 ("H","C"),("H","F"),("H","I"),("I","H"),
>                 ("I","J"),("J","I"),("J","G"),("K","L"),
```

```
>                   ("K","M"),("L","K"),("L","M"),("M","K"),
>                   ("M","L") ] []
```

JAVA IMPLEMENTATION OF BFS

```
public class Label {
   String name;
   int label;
   String predecessor;

   // Constructs a label of form (name,label,pred)
   public Label(String n, int l, String pred) {
      name = n; label = l; predecessor = pred;
   }
   public String getName() {
      return this.name;
   }
   public int getLabel() {
      return this.label;
   }
   public String getPred() {
      return this.predecessor;
   }

   @Override // Give a representation of "Labels" Object: Label x (lab,pred)
   public String toString(){
      return "Label " + name + " " + "(" + label + "," + predecessor + ")";
   }

}

public class Edge {
   String node1;
   String node2;

   public Edge(String nd1, String nd2){
      node1 = nd1;
      node2 = nd2;
   }
   public String getNode1(){
      return this.node1;
   }
   public String getNode2(){
      return this.node2;
   }
   @Override
   public String toString(){
      return "(" + node1 + "," + node2 + ")";
   }
}

public class Graph {
   ArrayList<Edge> edges = new ArrayList<>();
   ArrayList<Label> labels = new ArrayList<>();

   // A graph contains a a list of edges and a list of labels
   public Graph(ArrayList<Edge> es, ArrayList<Label> lbs) {
      edges = es; labels = lbs;
   }
   // Return the edges array list of a graph.
   public ArrayList<Edge> getEdges() {
      return this.edges;
   }
   // Return the labels array list of a graph.
```

```
public ArrayList<Label> getLabels() {
   return this.labels;
}
// Give a representation of a "Graph"
@Override
public String toString() {
   String graph = "Graph Edges: " + this.edges + "\n" +
                  "Graph Labels: " + this.labels;
   return graph;
}
// Test if two vertices are adjacent in a graph
public Boolean isAdj(String x, String y) {
   Boolean value = false;
   for(Edge e : edges) {
      if (x.equals(e.getNode1()) && y.equals(e.getNode2()) ||
         x.equals(e.getNode2()) && y.equals(e.getNode1()))
         value = true; }
   return value;
}
// Return the vertices of a graph
public ArrayList<String> vertices() {
   Set<String> vertices = new HashSet<>();
   for(Edge e : edges) {
      vertices.add(e.getNode1());
      vertices.add(e.getNode2());
   }
   return new ArrayList<>(vertices);
}
// Return an adjacency list of a vertex x
public ArrayList<String> adjacencyList(String x) {
   ArrayList<String> adjList = new ArrayList<>();
   for(Edge e : edges)
      if (x.equals(e.getNode1()))
         adjList.add(e.getNode2());
   return adjList;

}
// Test if a vertex x has a label assigned to it
public Boolean hasLabel(String x) {
   Boolean value = false;
   for(Label l : labels){
      if (x.equals(l.getName()))
         value = true;
   }
   return value;
}
// Assign a label l to a graph
public void assign(Label l) {
   labels.add(l);
}
// Assign multiple labels, ls, to a graph
public void assignLabels(ArrayList<Label> ls) {
   for(Label l : ls)
      this.assign(l);
}
// Return a label of the designated vertex from labels
public int getLabel(String x) {
   int label = 0;
   for(Label l : labels)
      if (x.equals(l.getName()))
         label = l.getLabel();
   return label;
}
// Return a pred of the designated vertex from labels
```

```
    public String getPred(String x) {
        String pred = "";
        for(Label l : labels)
            if (x.equals(l.getName()))
                pred = l.getPred();
        return pred;
    }
    // Test if all the vertices are labeled.
    public Boolean allLabeled(ArrayList<String> vls) {
        Boolean value = true;
        for(String v : vls)
            if (!this.hasLabel(v))
                value = false;
        return value;
    }
    /* enlarge : enlarge the labeling of a graph with
     * an initial root; L = {S} aka L contains the label
     * "Label S (0,-)"
     */
    public Graph enlarge(ArrayList<String> vls, Graph g) {
        ArrayList<String> adj;
        ArrayList<String> newAdj = new ArrayList<>();
        String pred;
            for(String v : vls) {
                newAdj.addAll(g.adjacencyList(v));
            }
            if (g.allLabeled(newAdj))
                return g;
            else
                for(String v : vls) {
                    adj = g.adjacencyList(v);
                    pred = v;
                    for(String s : adj)
                        if (!g.hasLabel(s) && !s.equals("S"))
                        g.assign(new Label (s,(g.getLabel(pred) + 1),v));
        }
        return enlarge(newAdj, g);

    }
    // Construct a shortest path to a vertex
    public ArrayList<String> shortestPath(String x, ArrayList<String> list) {
        ArrayList<String> temp = new ArrayList<>();
        temp.add(x);
        temp.addAll(this.predList(x,list));
        Collections.reverse(temp);
        return temp;
    }
        private ArrayList<String> predList(String x, ArrayList<String> list) {
            if (x.equals("S")) {
                return list;
            }
            else
                list.add(this.getPred(x));
            return predList(this.getPred(x), list);
        }
}

public class BFS {

    public static void main(String[] args) {
        ArrayList<String> vertLabels = new ArrayList<>();

        // initialize a graph with edges and no labels
        ArrayList<Label> ls = new ArrayList<>();
```

```
        ArrayList<Edge> es = new ArrayList<>(Arrays.asList
        (new Edge("S","A"),new Edge("S","B"),new Edge("A","E"),new Edge("A","C"),
         new Edge("A","S"),new Edge("B","S"),new Edge("B","C"),new Edge("B","D"),
         new Edge("C","E"),new Edge("C","H"),new Edge("D","G"),new Edge("E","C"),
         new Edge("E","F"),new Edge("F","E"),new Edge("F","H"),new Edge("G","J"),
         new Edge("H","C"),new Edge("H","F"),new Edge("H","I"),new Edge("I","H"),
         new Edge("I","J"),new Edge("J","I"),new Edge("J","G"),new Edge("K","L"),
         new Edge("K","M"),new Edge("L","K"),new Edge("L","M"),new Edge("M","K"),
         new Edge("M","L")));

        Graph graph = new Graph(es, ls);

        // Step 1 (a) : Assign S the label 0, and let S have no predecessor.
           graph.assign(new Label("S",0,"-"));
        // Step 1 (b) : Set L = {S}
           vertLabels.add("S");
        // Step 2 : enlarge labeling
           graph.enlarge(vertLabels, graph);
        // Step 3 : Construct a shortest path to a vertex.
        try {
           ArrayList<String> path = new ArrayList<>();
           System.out.println("Shortest Path to J: " + graph.shortestPath("e", path));

        } catch (StackOverflowError e) {
           System.out.println
           ("Vertex does not exist or there is no path. Case sensitive");
        }

        System.out.println(graph.toString());

    }
}
```