# CS421 Final Project: Implement Graph Data Structure and Depth-First Search Algorithm

Name: Xi Luo
Email: xiluo4@illinois.edu

## Overview

In this project, I implemented the Graph data structure and the Depth First Search algorithm with graphs, based on an ACM paper: Structuring Depth-First Search Algorithms in Haskell.

Graph algorithms expressed in functional languages often suffered from their inherited imperative, state-based style. This paper introduced methods to implement DFS algorithms in linear time in Haskell by making use of monads to provide updatable state.

The goals of this project from the proposal:

1. To implement graph data type that represents nodes and edges of directed and undirected graphs
2. To implement relevant methods such as
    - Map function for graph
    - Build function to build a graph
    - Reverse function to reverse all edges
    - Transpose function to transpose a graph by reversing all the edges
3. To implement DFS algorithms detailed in the paper
4. To write tests for graph and DFS implementations

I have accomplished all the goals stated in the proposal, and have also achieved a better understanding of monads and state transformation from this project.

# Implementation

## Major tasks or capabilities of the code

1. Implement the graph data structure

```haskell
--- Implement Graph
type Table a = Array Char a
type Graph = Table [Char]

vertices :: Graph -> [Char]
vertices = indices

type Edge = (Char, Char)
edges :: Graph -> [Edge]
edges g = [(v,w) | v <- vertices g, w <- g!v]
```

In my code implementation, Graph is represented by an array of adjacency lists. The array is indexed by vertices, and each adjacency list contains all other vertices that are reachable along a single edge. Each vertex is represented by a unique character. An edge is represented by a pair of vertices, with the order of the vertices indicating the direction of the edge. For example (v, w) indicates an edge directed from vertex v to vertex w.

2. Implement methods to manipulate or retrieve information from graphs

- mapT: map graph to a new graph using the method specified
- outdegree: retrieve the number of outward edges from each vertex of the graph
- buildG: build a graph from the array of adjacency lists and the 2 vertices as the outermost bounds.
- transposeG: modify the graph to its transpose, i.e. with the same set of vertices but reversed edges
- reverseE: get the list of reversed edges of the graph
- Indegree: retrieve the number of inward edges towards each vertex of the graph

```haskell
------ methods to manipulate graph / retrieve graph info
mapT :: (Char -> a -> b) -> Table a -> Table b
mapT f t = array (bounds t) [(v, f v (t!v)) | v <- indices t]

type Bounds = (Char, Char)
outdegree :: Graph -> Table Int
outdegree g = mapT numEdges g
    where numEdges v ws = length ws

buildG :: Bounds -> [Edge] -> Graph
buildG bnds es = accumArray (flip (:)) [] bnds es

transposeG :: Graph -> Graph
transposeG g = buildG (bounds g) (reverseE g)

reverseE :: Graph -> [Edge]
reverseE g = [ (w,v) | (v,w) <- edges g ]

indegree :: Graph -> Table Int
indegree g = outdegree (transposeG g)
```

3. Implement Depth First Search algorithm

In the previous section, we represented the Graph as a table (an array of adjacency lists). In this section, we represent the outcome of the DFS search as a depth-first forest of nodes from the input graph.

```haskell
--- Implement DFS
data Tree a = Node a (Forest a)
                deriving (Eq, Show)
type Forest a = [Tree a]

dfs :: Graph -> [Char] -> Forest Char
dfs g vs = prune (bounds g) (map (generate g) vs)

dff :: Graph -> Forest Char
dff g = dfs g (vertices g)
```

- dff: manipulates the input graph to construct the depth-first forest containing all the vertices from the graph. The depth-first forest constructed from the graph can be considered as a subgraph that contains the same vertices, but the subset of the graph edges. If there is a circle in the graph, visited vertices will not appear again in the resulting depth-first forest. For example, in the Fig 1 graph below, there is a circle a -> g -> b -> a, and the generate the depth-first forest starting from vertex a will only contain a -> [g -> [b -> i, f -> null]], while the edge b -> a is lost.

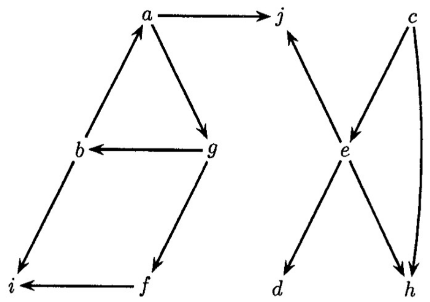


Fig 1. A graph example

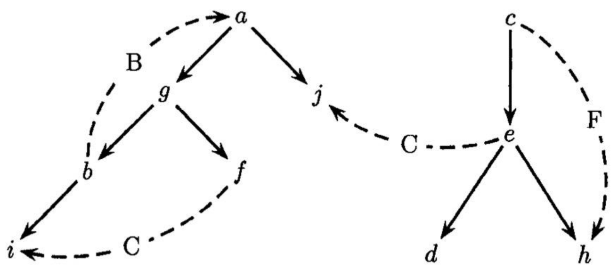


Fig 2. A depth-first forest generated from Fig 1 graph

- dfs: a more flexible function that takes in a graph and a subset of the vertices from the graph, and outputs the depth-first forest rooted from the set of vertices. The subset of the vertices provides an initial ordering of the vertices. For example, for the Fig 1 graph, dfs from [b] will result in a depth-first forest b -> [a -> [g -> f, j -> null], i -> null]. Comparing this result from the previous example, note that in this example, edge b -> a is preserved in the resulting depth-first forest, while edge g -> b is lost. In this case, edge b -> a is preserved because it is generating a depth-first forest starting from vertex b, and vertex a is not in the initial ordering vertex set. Similarly, g -> b is lost because b is the root of the resulting depth-first forest, and is being visited already.

Helper functions in implementing dfs:

- generate: generate a tree from the input graph. If the graph is cyclic, the tree will be infinite, but this will not be a problem since the tree is generated on demand
- prune: the pruning process discards the vertices that have been visited as discussed previously. This is done using state transformers and maintaining an array of booleans indexed by the set of vertices (set m in code). It begins from a fresh state and an empty set, and then calls chop function which chops the input list of trees until all the subtrees from the forest have already

been chopped, and the prune function returns the result from chop function and discards the final state.

- chop: the chop function takes in a list of trees, and
    a. If the root v of the first tree is already visited,
        i. Discard the whole tree, and move on to the next tree in the list
    b. If the root v of the first tree is not visited,
        i. Add the vertex to the set (set m)
        ii. Chop the children forest ts of root node v, adding all the nodes to set m.
        iii. Move on to chop the next tree in the list with the updated set m
        iv. Construct and return the resulting forest from the chopped subforest from node v, and the chopped remaining forest.

```haskell
------ Generating
generate :: Graph -> Char -> Tree Char
generate g v = Node v (map (generate g) (g!v))

------ Pruning
type Set s = STArray s Char Bool

mkEmpty :: Bounds -> ST s (Set s)
mkEmpty bnds = newArray bnds False

contains :: Set s -> Char -> ST s Bool
contains m v = readArray m v

include :: Set s -> Char -> ST s ()
include m v = writeArray m v True

prune :: Bounds -> Forest Char -> Forest Char
prune bnds ts
    = runST (mkEmpty bnds >>= \m ->
                chop m ts)

chop :: Set s -> Forest Char -> ST s (Forest Char)
chop m [] = return []
chop m (Node v ts : us)
    = contains m v >>= \visited ->
        if visited then
            chop m us
        else
            include m v     >>= \_     ->
            chop m ts       >>= \as    ->
            chop m us       >>= \bs    ->
            return ((Node v as) : bs)
```

4. Implementation of DFS applications

I implemented some simple applications of DFS to validate the algorithm implementation:

**Numbering**

- preOrd: making use of dff function, generates a depth-first forest from the input graph, and returns the vertices in pre-order. For example, for Fig 1 graph, the result will be "agbifjcedh"
- tabulate: returns pre-ordered positions of vertices from the preOrd result. For example, vertex b has a position 3, and vertex h has a position 10.

**Topological Sorting**

- postOrd: similar to preOrd, by making use of dff function, generates a depth-first forest from the input graph, and returns the vertices in post-order. For example, for Fig 1 graph, the result will be "ibfgjadhec"
- topSort: topological sort of the graph returns an arrangement of the vertices into a linear sequence v1, ... vn, such that there are no edges from vj to vi where i < j.

**Finding Reachable Vertex**

- reachable: making use of dfs function and preorder function to find the list of vertices reachable from a vertex in the graph (including the vertex itself). It first generates a depth-first forest starting at the given vertex from the graph, and then returns all the vertices of the generated depth-first forest. For example, finding all vertices reachable from vertex b in Fig 1 graph will give "bagfij"

## Components of the code

- app/Main.hs
  This is the haskell code that contains the main function, and is the entry point to run the whole project.

- src/Lib.hs
  This contains the main implementations, including implementation of graph data structure, DFS algorithm, and the applications of DFS

- test/
  - Spec.hs
    This is the haskell code that facilitates running of the test cases, printing the test results and scores. It enables the tests to be run in a whole or individually.

    ○ Tests.hs

      This contains all the test cases for the project, including the unit tests, feature tests for DFS, and the larger test codes of the functional tests for DFS applications.

- finalproj.cabal

  This configures how the project should be built, the internal dependencies between the haskell codes, and the external dependencies and their release versions (such as unordered-containers >= 0.2, array > 0.5)

### Status of the project

I have accomplished all the goals stated in the proposal.

## Tests

The implementation has passed all the tests. Some of the test cases have already been discussed as examples in the "Implementation" section of this report.

```
allTests = [
            --- unit tests
            (tests_build_graph, "ut - build graph")
           , (tests_get_vertices, "ut - get vertices")
           , (tests_get_edges, "ut - get edges")
           , (tests_get_outdegree, "ut - get outdegree")
           , (tests_transposeG, "ut - transpose graph")
           , (tests_get_indegree, "ut - get indegree")
           , (tests_generate_tree, "ut - generate tree at vertex")

            --- Feature Tests
           , (tests_depth_first_forest, "dfs - depth first forest from 1st vertex of graph")
           , (tests_depth_first_search, "dfs - depth first search from specified vertex of graph")

            --- DFS Applications Tests
           , (tests_preord_graph, "numbering - get pre-order of depth-first forest from graph")
           , (tests_tabulate_vertices, "numbering - get pre-ordered positions of vertices")
           , (tests_postord_graph, "top sort - get post-order of depth-first forest from graph")
           , (tests_topsort_graph, "top sort - get topological sort from graph")
           , (tests_reachable_vertices, "reachable - get rechable vertices from given vertex in graph")
           ]
```

```
Running Tests
=============
Pass: ut - build graph
Pass: ut - get vertices
Pass: ut - get edges
Pass: ut - get outdegree
Pass: ut - transpose graph
Pass: ut - get indegree
Pass: ut - generate tree at vertex
Pass: dfs - depth first forest from 1st vertex of graph
Pass: dfs - depth first search from specified vertex of graph
Pass: numbering - get pre-order of depth-first forest from graph
Pass: numbering - get pre-ordered positions of vertices
Pass: top sort - get post-order of depth-first forest from graph
Pass: top sort - get topological sort from graph
Pass: reachable - get rechable vertices from given vertex in graph

Score: 100 / 100
All tests passed.
```

### Unit Tests

The unit tests cover as many as possible of all the helper functions in implementing graph and DFS algorithms.

### Feature Tests

The feature tests mainly test on the features of DFS algorithm, which is to generate depth-first forest from the input graph as expected.

### Larger Test Codes

The rest of the tests cover the applications of DFS algorithms, testing the numbering, topological search, and finding reachable vertices which make use of the DFS implementation: dff and dfs functions.

## Listing

src/Lib.hs

```
--- Getting Started

--- ================



module Lib where

import Data.Array

import Control.Monad.ST
```

```haskell
import Data.Array.ST


--- Implement Graph

type Table a = Array Char a

type Graph = Table [Char]


vertices :: Graph -> [Char]

vertices = indices


type Edge = (Char, Char)

edges :: Graph -> [Edge]

edges g = [(v,w) | v <- vertices g, w <- g!v]


------ methods to manipulate graph / retrieve graph info

mapT :: (Char -> a -> b) -> Table a -> Table b

mapT f t = array (bounds t) [(v, f v (t!v)) | v <- indices t]


type Bounds = (Char, Char)

outdegree :: Graph -> Table Int

outdegree g = mapT numEdges g

  where numEdges v ws = length ws


buildG :: Bounds -> [Edge] -> Graph

buildG bnds es = accumArray (flip (:)) [] bnds es
```

```haskell
transposeG :: Graph -> Graph

transposeG g = buildG (bounds g) (reverseE g)



reverseE :: Graph -> [Edge]

reverseE g = [ (w,v) | (v,w) <- edges g ]



indegree :: Graph -> Table Int

indegree g = outdegree (transposeG g)



--- Implement DFS

data Tree a = Node a (Forest a)

            deriving (Eq, Show)

type Forest a = [Tree a]



dfs :: Graph -> [Char] -> Forest Char

dfs g vs = prune (bounds g) (map (generate g) vs)



dff :: Graph -> Forest Char

dff g = dfs g (vertices g)



------ Generating

generate :: Graph -> Char -> Tree Char

generate g v = Node v (map (generate g) (g!v))



------ Pruning
```

```haskell
type Set s = STArray s Char Bool

mkEmpty :: Bounds -> ST s (Set s)
mkEmpty bnds = newArray bnds False

contains :: Set s -> Char -> ST s Bool
contains m v = readArray m v

include :: Set s -> Char -> ST s ()
include m v = writeArray m v True

prune :: Bounds -> Forest Char -> Forest Char
prune bnds ts
  = runST (mkEmpty bnds >>= \m ->
          chop m ts)

chop :: Set s -> Forest Char -> ST s (Forest Char)
chop m [] = return []
chop m (Node v ts : us)
  = contains m v >>= \visited ->
     if visited then
        chop m us
     else
        include m v    >>= \_    ->
        chop m ts      >>= \as   ->
```

```
        chop m us       >>= \bs    ->

        return ((Node v as) : bs)


--- DFS Applications

------ Numbering

preorder :: Tree a -> [a]

preorder (Node a ts) = [a] ++ preorderF ts



preorderF :: Forest a -> [a]

preorderF ts = concat (map preorder ts)



preOrd :: Graph -> [Char]

preOrd g = preorderF (dff g)



tabulate :: Bounds -> [Char] -> Table Int

tabulate bnds vs = array bnds (zip vs [1..])



preArr :: Bounds -> Forest Char -> Table Int

preArr bnds ts = tabulate bnds (preorderF ts)



------ Topological sorting

postorder :: Tree a -> [a]

postorder (Node a ts) = postorderF ts ++ [a]



postorderF :: Forest a -> [a]
```

```haskell
postorderF ts = concat (map postorder ts)

postOrd g = postorderF (dff g)


topSort :: Graph -> [Char]

topSort g = reverse (postOrd g)


------ Connected components

components :: Graph -> Forest Char

components g = dff (undirected g)


undirected :: Graph -> Graph

undirected g = buildG (bounds g) (edges g ++ reverseE g)


------ Strong connected components

scc :: Graph -> Forest Char

scc g = dfs (transposeG g) (reverse (postOrd g))


scc' :: Graph -> Forest Char

scc' g = dfs g (reverse (postOrd (transposeG g)))


------ Finding reachable vertices

reachable :: Graph -> Char -> [Char]

reachable g v = preorderF (dfs g [v])


path :: Graph -> Char -> Char -> Bool
```

```
path g v w = w `elem` (reachable g v)
```

test/Tests.hs

```
--- Getting Started

--- ================



--- Testing Your Code

--- ----------------



module Tests where



import Data.List ((\\))

import Data.Array

import Lib



allTests :: [([Bool], String)]

allTests = [

         --- unit tests

         (tests_build_graph, "ut - build graph")

        , (tests_get_vertices, "ut - get vertices")

        , (tests_get_edges, "ut - get edges")

        , (tests_get_outdegree, "ut - get outdegree")

        , (tests_transposeG, "ut - transpose graph")

        , (tests_get_indegree, "ut - get indegree")

        , (tests_generate_tree, "ut - generate tree at vertex")
```

```haskell
    --- Feature Tests

    , (tests_depth_first_forest, "dfs - depth first forest from 1st vertex of graph")

    , (tests_depth_first_search, "dfs - depth first search from specified vertex of graph")


    --- DFS Applications Tests

    , (tests_preord_graph, "numbering - get pre-order of depth-first forest from graph")

    , (tests_tabulate_vertices, "numbering - get pre-ordered positions of vertices")

    , (tests_postord_graph, "top sort - get post-order of depth-first forest from graph")

    , (tests_topsort_graph, "top sort - get topological sort from graph")

    , (tests_reachable_vertices, "reachable - get rechable vertices from given vertex in graph")

    ]




--- unit tests

graph = buildG ('a','j')

        [('a','j'),('a','g'),('b','i'),('b','a'),('c','h'),('c','e'),('e','j'),('e','h'),('e','d'),('f','i'),('g','f'),('g','b')]


tests_build_graph :: [Bool]

tests_build_graph = [ graph ! 'e' == ['d', 'h', 'j'] ]


tests_get_vertices :: [Bool]

tests_get_vertices = [ vertices graph == "abcdefghij" ]


tests_get_edges :: [Bool]
```

```
tests_get_edges = [ edges graph == [('a','g'),('a','j'),('b','a'),('b','i'),('c','e'),('c','h'),('e','d'),('e','h'),('e','j'),('f','i'),('g','b'),('g','f')]]



tests_get_outdegree :: [Bool]

tests_get_outdegree = [ show (outdegree graph)

            == "array ('a','j') [('a',2),('b',2),('c',2),('d',0),('e',3),('f',1),('g',2),('h',0),('i',0),('j',0)]"

         ]



tests_transposeG :: [Bool]

tests_transposeG = [ graph ! 'c' == ['e', 'h'], (transposeG graph) ! 'e' == ['c'] ]



tests_get_indegree :: [Bool]

tests_get_indegree = [ show (indegree graph)

            == "array ('a','j') [('a',1),('b',1),('c',0),('d',1),('e',1),('f',1),('g',1),('h',2),('i',2),('j',2)]"

         ]



tests_generate_tree :: [Bool]

tests_generate_tree = [ generate graph 'e' == Node 'e' [Node 'd' [],Node 'h' [],Node 'j' []] ]



--- Feature Tests

tests_depth_first_forest :: [Bool]

tests_depth_first_forest = [ dff graph == [Node 'a' [Node 'g' [Node 'b' [Node 'i' []],Node 'f' []],Node 'j' []],Node 'c' [Node 'e' [Node 'd' [],Node 'h' []]]] ]



tests_depth_first_search :: [Bool]

tests_depth_first_search = [ dfs graph ['b'] == [Node 'b' [Node 'a' [Node 'g' [Node 'f' [Node 'i' []]],Node 'j' []]]] ]
```

```
--- DFS Applications Tests

------ Numbering

tests_preord_graph :: [Bool]

tests_preord_graph = [ preOrd graph == "agbifjcedh" ]



tests_tabulate_vertices :: [Bool]

tests_tabulate_vertices = [ show (tabulate ('a','j') (preOrd graph)) == "array ('a','j') [('a',1),('b',3),('c',7),('d',9),('e',8),('f',5),('g',2),('h',10),('i',4),('j',6)]" ]



------ Topological Sort

tests_postord_graph :: [Bool]

tests_postord_graph = [ postOrd graph == "ibfgjadhec" ]



tests_topsort_graph :: [Bool]

tests_topsort_graph = [ topSort graph == "cehdajgfbi" ]



------ Finding reachable vertices

tests_reachable_vertices :: [Bool]

tests_reachable_vertices = [ reachable graph 'b' == "bagfij" ]
```