Homework 11: Graphs

Due: 11:59 PM, Nov 20, 2018

Contents

1 Sets 2

2 Depth-first search 2

2.1 Testing ................................................................. 4

2.2 A note on graphs ..................................................... 4

Objectives

By the end of this homework you will be able to:

1. implement a set module
2. implement depth-first search graph algorithm

How to Hand In

For this (and all) homework assignments, you should hand in answers for all the non-practice questions. For this homework specifically, this entails answering the Set and Depth-first Search questions.

In order to hand in your solutions to these problems, they must be stored in appropriately-named files. In particular, each should be named for the corresponding problem:

- README.txt
- CS17setup.ml
- sig_set.ml
- list_set.ml
- depth_first_search.ml

For this assignment, all files you turn in that contain code must be OCaml files, so they must end with extension .ml. For all coding problems in this homework, you should follow the OCaml design recipe outlined in the previous homework.

For this and every future assignment, you should include the CS17setup.ml file with your hand-in. You should also have a README.txt file whose first line contains only your Banner ID, and
optionally with a message to the person grading explaining anything peculiar about the handin. For example:

README.txt:
klein@brown.edu
There’s nothing to say except that I’m turning in these files plus this README the way the instructions say that I should.

To hand in your solutions to these problems, you must upload them to Gradescope. Do not zip or compress them. If you re-submit your homework, you must re-submit all files. If you choose to also store these files on department machines, all your solution files should reside in your `/course/cs0170/homeworks/hw11` directory.

Set-Up

We have provided you with all of the files you will need for this homework assignment. The complete CS17setup file as well as outlines for the other files you will fill out for this assignment are in the `hw11/src` directory.

Before starting this assignment you’ll want to copy the contents of the `hw11/src` directory into your personal `hw11` directory. On the department system, this would look something like this:

```bash
cp /course/cs0170/src/hw11/* ~/course/cs0170/homeworks/hw11
```

After you’ve transferred the files, begin filling in your solutions to the tasks on this homework. Make sure you do not modify any of the code we have written for you, especially not the `#use` statements. If you modify the template, there will be deductions.

Problems

1 Sets

As you did in your lab, you’ll write an implementation of a set data structure. You’ll be implementing a simplified version of this to be used in later parts of this homework. You might even be able to utilize some of your lab code for this part.

We’ve supplied a module type `SET` that has the following signature. It is provided in `set_sig.ml`.

```ml
module type SET =

sig
  type 'a set
  (* outputs an empty set *)
  val empty: 'a set
  (* outputs a set with the new element added to the old set *)
  val insert: 'a set * 'a -> 'a set
  (* outputs true if set contains given element, false otherwise *)
  val containsP: 'a set * 'a -> bool
end
```

Task: Using this signature, implement this module in `list_set.ml`. Be sure to test all procedures.
2 Depth-first search

In class, we’ve introduced the concept of graph search. Abstractly, a graph is made up of nodes (a.k.a. vertices) and edges (a.k.a. arcs). Node \( v \) is a neighbor of node \( u \) if there is an edge from \( u \) to \( v \).

A path in a graph is a sequence of nodes \( v_1v_2\cdots v_k \) such that \( v_2 \) is a neighbor of \( v_1 \), \( v_3 \) is a neighbor of \( v_2 \), and so on. It is a path from \( v_1 \) to \( v_k \). A path from \( v \) to \( v \) is a sequence consisting of just \( v \).

It is useful to determine if there is a path from a given origin node to a given destination node, and to find such a path if it exists. It would allow us to figure out which flights to take to get from one city to another, or figure out how two people know each other through mutual friends.

You will write a procedure that solves this problem. It will find a path from a given node to another given node if one exists.

In this homework, we will represent a graph as a procedure. Specifically, it is a procedure that takes in a node and produces a list consisting of all neighbors of that node.

The formal type definition is as follows:

```ocaml
type 'a graph = 'a -> 'a list;
```

The type parameter ‘a stands for the OCaml type used in representing nodes. Thus this graph representation is polymorphic in that a graph can be based on any type of node.

We’ve given an example of a graph in depth_first_search.ml. Consider the problem of trying to go from an airport in one city to an airport in another city. We have modeled this as a graph with vertices being airports, and flights being edges. We included flights from this map:

![Graph Diagram]

A sample graph search problem is finding a path from PVD to SFO

**Task:** Complete the procedure `is_valid_path` which takes in a graph, an option of a list (which is the path), as well as a source, and a destination. This procedure returns true if the provided path is valid, and false otherwise.

**Note:** You can use the the provided graph (or another if you’d like!) to test `is_valid_path`.

**Note:** If given `None` as the path, `is_valid_path` should return false.
Depth-first search is one algorithm for searching a graph and it can be used to find a path between given nodes if such a path exists. Note that it is not likely to find the shortest path.

In order to avoid visiting the same node over and over (and its neighbors, and its neighbors' neighbors, and so on), DFS keep track of which vertices have been visited, using a set. In visiting the neighbors of a node \( v \), DFS skips visiting neighbors that have previously been visited. When it visits a node, it adds that node to the set of visited nodes before continuing the search.

**Task:** In `depth_first_search.ml`, the procedure `dfs` takes in a graph, a source node, and a destination node. If the graph contains a path from the source to the destination, the procedure should find the path \([v_1; v_2; \cdots; v_k]\) where \( v_1 \) is the source and \( v_k \) is the destination. The return value in this case should be `Some [v_1; v_2; \cdots; v_k]`. If there is no such path, the procedure should return `None`.

**Hint:** As mentioned above, the depth-first search algorithm keeps track of a set `visited` consisting of all the previously visited nodes. However, the procedure `dfs` does not take in a set. It is therefore convenient to have a recursive helper procedure `dfs_helper` that does take in a `visited` set and implements depth-first search; the original procedure `dfs` merely calls `dfs_helper` with an empty set as argument. This is outlined in the code we provide in `depth_first_search.ml`.

### 2.1 Testing

You can use the provided graph (or another if you’d like!) to test your depth-first search algorithm. Your test cases should verify that the path `dfs` produces is a valid path. You should use `is_valid_path` within your test cases.

### 2.2 A note on graphs

This graph representation is not complete; a full graph representation would also specify the set of nodes, but we will omit this for this assignment.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS 17 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/csci0170/feedback](http://cs.brown.edu/courses/csci0170/feedback)