Data Structures Chapter 9 Problems
Noah Weiner

Problem 9.3

Write a program to perform a topological sort on a graph.

There are a few different algorithms one could use for a topological sort, as explained by wikipedia. The algorithm the book uses appears to be a slightly more optimized version of Kahn's algorithm. For a graph of n nodes and m edges, it has a complexity of O(n + m), essentially linear time.

The book uses a queue to cycle through nodes of indegree 0, and I see no reason to modify that. I have added code for computing the indegrees of nodes.

```
/* assumes graph maintains the following private members
 * - vector<vector<int>> adjacencies: adjacency list
 *
 * Nodes are represented as indicies, so that an index
 * int the adjancency list can correspond to an index
 * in a node list.
 */
void Graph::topsort()
{
    queue<node> q;
    vector<int> sorted;
    int indegrees[adjacencies.size()];
    int counter = 0;

    for (vector<int> connected : adjacencies)
        for (int i : connected)
            ++indegress[i];

    q.makeEmpty();
    for (int i = 0; i < adjacencies.size(); ++i)
        if (indegrees[i] == 0)
            q.push_back(i);

    while(!q.empty()) {
        int i = q.front();
        sorted.push_back(i);
        ++counter;

        for (int n : adjacencies[i]) {
            --indegrees[n];
            if (indegrees[n] <= 0)
                q.push_back(n);
```

```
        }

        q.pop();
    }
    if( counter != adjacencies.size() )
        throw CycleFoundException{ };
}
```

Problem 9.9

Write a program to solve the single-source shortest-path problem.

The simplest solution I know of (and the one from the book) is Djistra's algorithm.

In order to implement it, you need to use some kind of heap to keep track of the unvisited node with the shortest distance. The best option would probably a Fibonacci heap since that gives us an O(1) decrease-key operation, which will be used whenever we update the distances of adjacent nodes.

Problem 9.15

a. Find a minimum spanning tree for the graph in Figure 9.84 using both Prim's and Kruskal's algorithms.

(see figure)

b. Is this minimum spanning tree unique? Why?

I'm not quite sure what's meant by "unique". There're certainly other minimum spanning trees like it. But there's more than one MST possible for this particular graph, it that's what you want to know.

Problem 9.16

Does either Prim's or Kruskal's algorithm work if there are negative edge weights?

Both Prim's and Kruskal's algorithms should work fine with negative weights, given that both have checks to prevent cycles.

Problem 9.32

a. Write a program to find an Euler circuit in a graph if one exists.

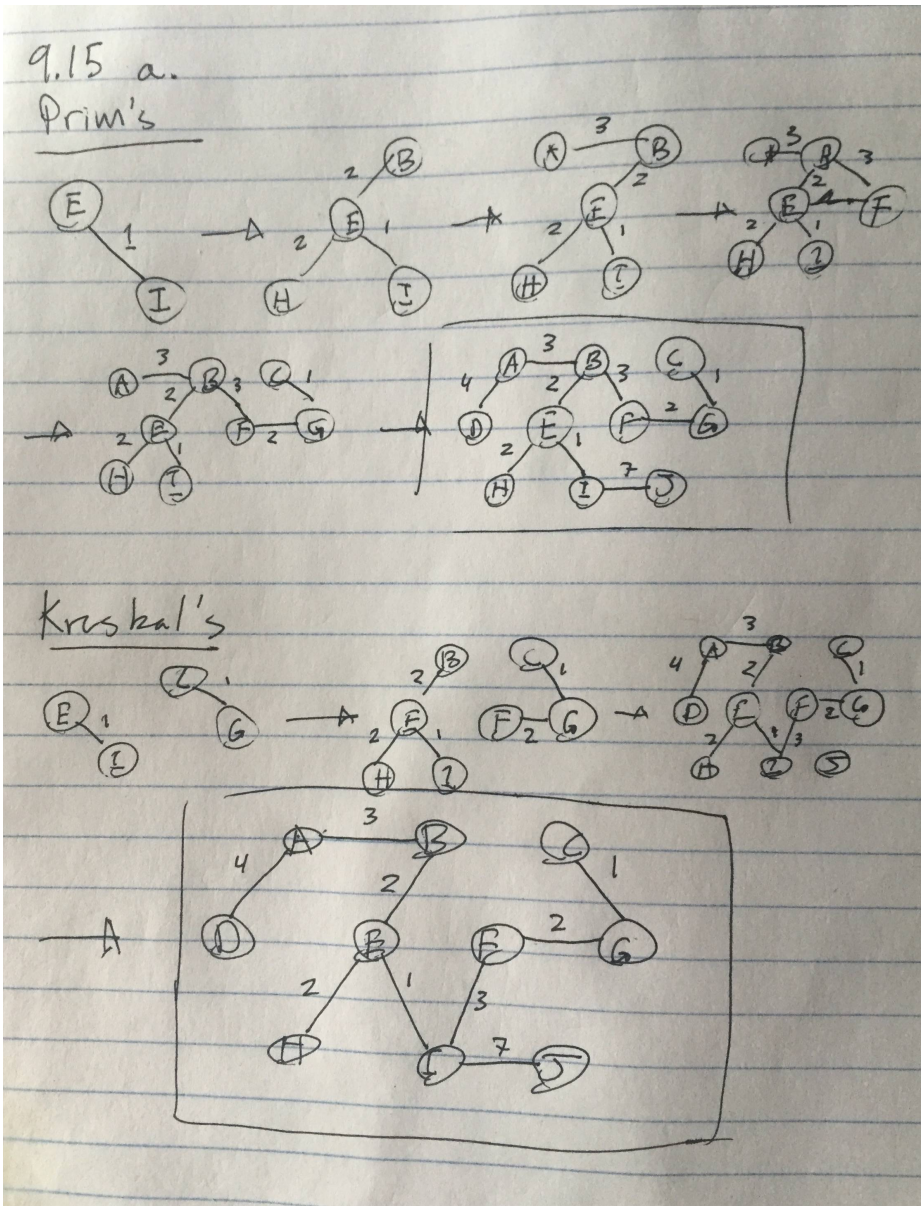

Figure 1: 9.15a

My solution is inspired by the Python implementation given here. Nodes are represented by integer indicies, and the graph is a 2D adjacency list. Assume no double entries in the adjacency list (i.e. adjencies[4] might have an entry for 3, but adjacencies[3] doesn't have an entry for 4.

```
list<int>& euler_circuit(const vector<list<int>>& adjacencies) {
  list<int> path;   /* return path */
  stack<int> stack;   /* stack to keep track of current circuit */
  vector<list<int>::const_iterator> itrs(adjacencies.size());
  int node;

  stack.push(0); /* start at node 0 */

  /* initialize iterators list */
  for (int i = 0; i < adjacencies.size(); ++i)
    itrs[i] = adjacencies[i].begin();

  while (!stack.empty()) {
    node = stack.top();

    if (itrs[node] != adjacencies[node].end()) {
      stack.push(*(itrs[node]));
      ++(itrs[node]);
    } else {
      path.push_back(node);
      stack.pop()
    }
  }
}
```

b. Write a program to find an Euler tour in a graph if one exists.

The program can be modified easily to handle tours by checking for odd edges, and starting on the first odd edge found.

Problem 9.46

> Section 8.7 described the generating of mazes. Suppose we want to output the path in the maze. Assume that the maze is represented as a matrix; each cell in the matrix stores information about what walls are present (or absent).

I added pathfinding functionality to the maze class I created for the chapter eight homework. Here's the code:

```
set<int> Maze::shortest_path(int start, int end) const {
  struct node {
    int id, prev, dist;
```

```
  bool visited;
};

class node_compare {
public:
  bool operator()(const node *a, const node *b)
    { return a->dist > b->dist; }
};

node nodes[cells.size()];
vector<node *> q;   /* let's just pretend this is a fib heap */
set<int> ret;
vector<node *> adjacent(4);
node *n;

/* initialize nodes */
for (int i = 0; i < cells.size(); ++i) {
  nodes[i].id = i;
  nodes[i].prev = -1;
  nodes[i].dist = i == start ? 0 : INT_MAX;
  nodes[i].visited = false;
  q.push_back(&nodes[i]);
}

/* really should use a fib heap though */
make_heap(q.begin(), q.end(), node_compare());

/* run Dijkstra */
while (!q.empty()) {
  n = q.front();
  pop_heap(q.begin(), q.end());
  q.pop_back();

  n->visited = true;

  /* find all adjacent cells */
  adjacent.clear();
  if ((n->id+1) % width != 0)
    adjacent.push_back(&nodes[n->id+1]);
  if (n->id % width != 0)
    adjacent.push_back(&nodes[n->id-1]);
  if (n->id + width < cells.size())
    adjacent.push_back(&nodes[n->id + width]);
  if (n->id - width > 0)
    adjacent.push_back(&nodes[n->id - width]);
```

```
    for (node *adj : adjacent) {
       if (!adj->visited && !walled(n->id, adj->id)
           && n->dist + 1 < adj->dist) {
         adj->dist = n->dist + 1;
         adj->prev = n->id;
       }
    }

    make_heap(q.begin(), q.end(), node_compare());
  }

  /* build path */
  n = &nodes[end];
  ret.insert(n->id);
  while (n->prev != -1) {
    n = &nodes[n->prev];
    ret.insert(n->id);
  }

  return ret;
}
```

Problem 9.54

> The clique problem can be stated as follows: Given an undirected graph, G = (V, E), and an integer, K, does G contain a complete subgraph of at least K vertices?
>
> The vertex cover problem can be stated as follows: Given an undirected graph, G = (V,E), and an integer, K, does G contain a subset V V such that |V | K and every edge in G has a vertex in V ? Show that the clique problem is polynomially reducible to vertex cover.

For any graph *G = (V, E)*, let *G' = (V, E')* where *E'* is all edges between nodes *u* and *v* where *(u, v)* is not in *E*. Given an integer *k*, if a clique *V'* exists for *G* and */V'/ <= k*, then there is a vertex cover *V - V'* in *G'*. That is, all nodes not part of a clique in *G* are part of a vertex cover in *G'*.

Suppose that graph *G = (V, E)* has a vertex cover *V'*, and that */V'/ <= k*. For each pair of nodes *u, v* in *G*, if both *u* and *v* are in *V'*, then *(u, v)* must be in *E*. If *(u, v)* is in *E'* (as defined above), then either *u* or *v* or both aren't in *V'* but in *V - V'*, so that *(u, v)* is then covered by *V - V'*. Because this is true for every edge in *E'*, then every edge of *E'* is covered by *V - V'*. Therefore, *V - V'* is a vertex cover of graph *G'*, with a size of */V/ - k*.

Generating the complementary graph (*G'*) consists of scanning every pair of

nodes *(u, v)* and determining if *(u, v)* is in *E*. This task can be completed in polynomial time, so therefore the clique problem is polynomially reducible to the vertex cover problem.

Source: http://www.cs.toronto.edu/~ekzhu/teaching/csc373summer2015/tut9.pdf