Lab 13 - Graphs

CS 251, Fall 2024

Copyright Notice

Exam 4

This exam is on **paper**. Please follow your TAs' instructions. Don't open the exam until instructed to do so.

Learning Goals

By the end of this lab, you'll:

- Understand how worklist algorithms work in helping us visit all nodes of a graph to solve problems related to graphs.
- Have practiced running the worklist algorithms to execute breadth-first search (BFS), depth-first search (DFS), and Dijkstra's algorithm on graphs.

Starter Code

N/A, today's lab is fully on paper.

Tasks

For this lab worksheet, work on the tasks in the **given order**. We strongly encourage collaboration in labs. Work with your peers (once they're done with the exam), and ask your TAs questions!

Throughout this lab, we will work on tracing the execution of worklist algorithms that we introduced in lecture. Below you can see the implementation of the worklist algorithm that we used for searching on a graph and which we will use for the first two tasks:

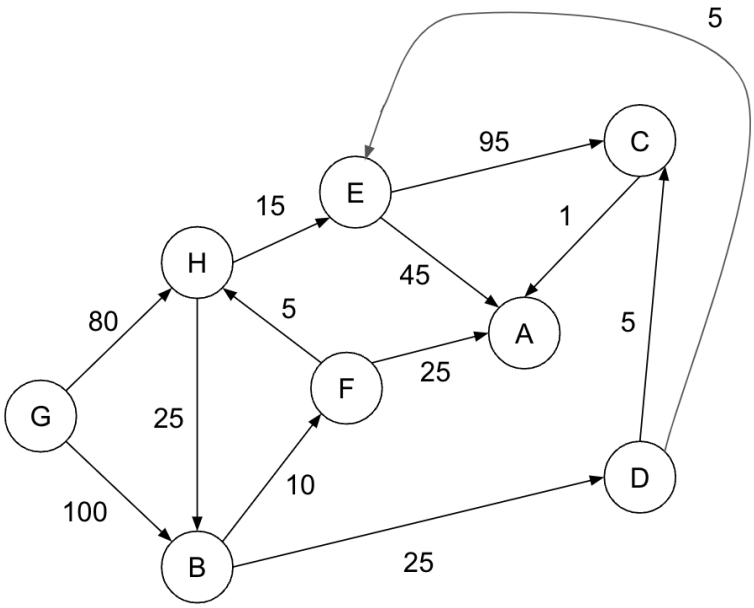
```
void search (char v) {
  Worklist worklist;
  worklist.add(v);
  unordered_set<char> seen;
  seen.insert(v);
  while (worklist.hasWork()) {
    char curr = worklist.next();
    doWork(curr);
    for (char w : neighbors(curr)) {
      if (seen.count(w) == 0) {
        worklist.add(w);
        seen.insert(w);
      }
  }
}
```

Worklists have three main functions:

- `add(v)` - adds the vertex `v` to the worklist
- `next()` - returns and removes the next vertex from the worklist
- `hasWork()` - returns false if the worklist is empty and true otherwise

How the `add` and `next` functions will work, will depend on the type of the worklist. For example, if the worklist is a queue, then `add` is the same as `enqueue` and `next` will be a `dequeue`. If the worklist is a stack, then `add` is a push on the stack and `next` is a `pop` from the stack.

In this lab activity we will trace the execution of BFS (which uses a queue as a worklist), DFS (stack), and Dijkstra's algorithm (priority queue) on a graph. For all of the problems in this lab activity we will trace the execution of a worklist algorithm on the following graph:



Task 1: BFS

In class we mentioned that BFS searches the graph by using the above worklist algorithm, with the worklist being a queue. Fill out the following table that executes the execution of the above `search` function. For each iteration of the `while` loop, keep track of the values for `curr`, `worklist`, `seen`, and the order in which the nodes were visited. For this task, you can disregard the weights of the edges.

We will start the BFS from node G. For iteration 0, we have filled out the values of `worklist` and `seen` before we enter the `while` loop. For the following iterations, you should put the contents of each variable, once that iteration is done.

Here we will assume that the `neighbors(curr)` function call returns a list of the neighbors `curr` in alphabetical order. We only say this so that we can all get the same answers. This isn't really a requirement of BFS.

We suggest that you copy paste the table from this doc, and paste it in another doc so that you can simply fill out the table using your computer keyboard.

Fill the table below with the tracing of the execution of the BFS algorithm on the above graph:

Iteration #	`curr`	`worklist`	`seen`	Visit order
0	N/A	G	{G}	N/A
1				
2				
3				
4				
5				
6				
7				
8				

Task 2: DFS

Now let's do a similar exercise but we will do DFS. In class we mentioned that DFS searches the graph by using the above worklist algorithm, with the worklist being a stack. Fill out the following table that executes the execution of the above `search` function. For each iteration of the `while` loop, keep track of the values for `curr`, `worklist`, `seen`, and the order in which the nodes were visited using DFS. For this task, you can disregard the weights of the edges.

We will start the DFS from node G. For iteration 0, we have filled out the values of `worklist` and `seen` before we enter the `while` loop. For the following iterations, you should put the contents of each variable, once that iteration is done. You can write the stack contents from left to right, where the leftmost value is the bottom of the stack and the rightmost value is the top of the stack, which means that when we pop from the stack we are removing the rightmost value on the list.

Here we will assume that the `neighbors(curr)` function call returns a list of the neighbors `curr` in alphabetical order. We only say this so that we can all get the same answers. This isn't really a requirement of BFS.

Fill the table below with the tracing of the execution of the BFS algorithm on the above graph:

Iteration #	`curr`	`worklist`	`seen`	Visit order
0	N/A	G	{G}	N/A
1				
2				
3				
4				
5				
6				
7				
8				

Task 3: Dijkstra's Algorithm

To perform the tracing of Dijkstra's algorithm, we will use the worklist algorithm that we presented in class, in which the worklist is a priority queue. As discussed in lecture, the priority value is taken to be our current upper bound for the length of the shortest path. If we use the analogy from lecture of pouring water into the nodes and water flowing through the edges according to the weights, this priority is the time in which water will arrive to that node by using the shortest path we have seen so far that gets to that node.

Remember also, that when checking the neighbors of `curr`, in that iteration we need to add them to `seen`, and add them to the `worklist` as appropriate. But also, if the neighbor is already in the `worklist` then we may need to update the current distance that is saved for that node (and in that case the predecessor as well).

The table you will fill out for the tracing is a bit different than before because now we are not necessarily interested in outputting the visit order. The output we are interested in is the confirmed lengths of the shortest path from node G to each other node. At each iteration, we will be able to output that length for whichever node is `curr` in that iteration. We also will be able to output the predecessor that led to that node and gave us that shortest path. This information you can enter in the last column of the table.

In the worklist, make sure to include the values listed in the priority queue as a triple that contains the <node, distance, predecessor>. So for example, if so far, for node B we have that the shortest path is of length 100 and the predecessor is G, then the worklist will have (B, 100, G).

We have filled out the first two rows for you as illustration.

Iteration #	`curr`	`worklist`	`seen`	<node, distance, predecessor>
0	N/A	G	{G}	N/A
1	G	(H, 80, G) (B, 100, G)	{G, H, B}	G, 0, N/A
2				
3				
4				
5				
6				
7				
8				

What is the shortest path from G to A? Trace back the predecessor to find out!

Deliverables

Filled out tables of the tracing of each of the algorithms with the correct orders for BFS and DFS and the correct shortest paths for Djikstra's algorithm.