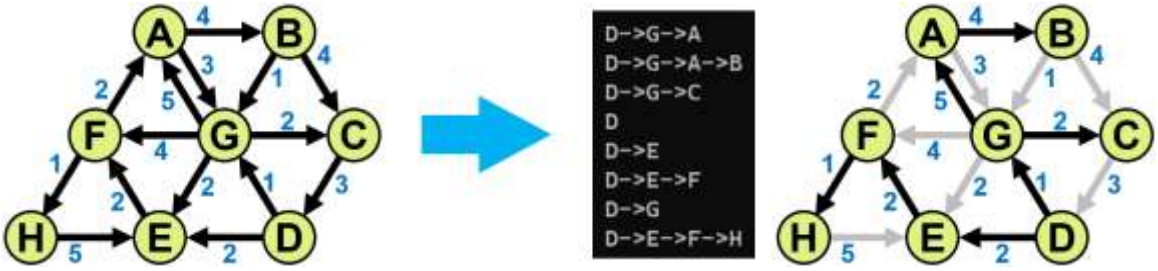
ECE220: Computer Systems & Programming **Spring 2024 ZJUI**
Machine Problem 3 **due: Saturday 9 March, 11:59:59 p.m.**

Dijkstra's Single-Source Shortest-Path Algorithm

This week, you will leverage your work creating routines for a heap to implement Dijkstra's single-source shortest-path algorithm as a subroutine. You will also write a program that calls the Dijkstra subroutine, then makes use of your MP1 subroutines to print the shortest paths found for a given graph and source node. Together, the subroutine and the program together require roughly 150 lines of LC-3 assembly code, including comments. Be aware, however, that although the code is shorter, the algorithm is more complex than the previous weeks' work.



An illustration of an input and the printed result of your program starting from node D appears above. On the right are those arcs in the graph used in any of the shortest paths. Note also that shortest paths are not unique. For example, the path D→E→F→A has the same length as the chosen D→G→A. If you follow this specification exactly, however, your code will produce the same paths as our code.

The objective for this week is to give you some experience with an important algorithm and with integrating subroutines into a complete program.

The Task

Both your program and Dijkstra's algorithm must operate on a graph stored as a matrix of arc distances at memory address x4000. The number of nodes ***V*** in the graph is stored at x3FFF, and the matrix has size ***V*** × ***V***. Node numbers are in the range [0, ***V***), which includes 0 but does not include ***V***. Given two nodes in the graph, ***M*** and ***N***, the distance for the arc (***M***, ***N***) is stored at address x4000 + ***MV*** + ***N***. Arcs that are not in the graph are given the distance "infinity," represented by the value x7FFF. Such a matrix representation of a graph is called an *adjacency matrix*.

Data structures from MP1 and 2 are also needed, but must be initialized and managed by your code:

- x5000 – graph node predecessor array (indexed by node number),
- x6000 – distance from source node array (indexed by node number),
- x7000 – heap index (indexed by node number),
- x8000 – heap (indexed by heap element index), with heap length at x7FFF, and
- xC000 – a good place to put the base of the stack needed for your MP1 subroutines.

The program that you must write is fairly simple. You may decide whether to write it before, while, or after you complete your implementation of Dijkstra's algorithm. In addition to the graph (starting at x4000, with the number of nodes at x3FFF), your program must make use of the starting node number provided to you at memory address x3FFE. This value must be passed into Dijkstra's algorithm in R0. Once Dijkstra's algorithm executes, initialize R6 as a stack—simply setting it to xC000 should be sufficient—so that your MP1 subroutines have space to operate, then use the predecessor array produced by the algorithm along with your **`FILL_STACK`** and **`PRINT_PATH`** subroutines to print the path from the source to every node in the graph. Note that all data structures

are in the same place in memory as in MP1 and MP2, so the amount of code required is quite small—fewer than 20 lines.

Dijkstra's algorithm is longer—a little over 120 lines including comments—and more complex than the program. The algorithm works by establishing a frontier consisting of shortest paths to all nodes in a neighborhood around the source node. Initially, the frontier separates the source from all other nodes, but other nodes are pulled across the frontier one by one in order of distance from the source node. For this purpose, we use a heap to keep track of the current "best" distance (and path, through the predecessor array) to each of the remaining nodes—those that have yet to be pulled across the frontier. One can prove—as you will see in a later class, perhaps ECE374—that any node is brought across the frontier only once we know a shortest path to that node. At least, assuming that all distances are non-negative.

Here's an overview of the algorithm to help you implement your subroutine, which you must call **`DIJKSTRA`**. The **`DIJKSTRA`** subroutine takes the source node number in R0 as an input, and all registers are caller-saved. In other words, you do not have to worry about preserving register values, but be sure that you keep that fact in mind when writing your program.

Step 1: Initialization – for each node in the graph, set its predecessor to -1, its heap index to -1, and its distance to "infinity" (x7FFF).

Step 2: Insert Starting Node into Heap – next, set the starting node's distance to 0 and place it into the heap at index 0. Be sure to update the size of the heap (to 1), the node number stored in heap index 0, and the heap index of the starting node. Note that since the heap contains only one element at this point, calling **`PERCOLATE_UP`** is not necessary.

Step 3: Process Nodes until the Heap is Empty – remove a node ***N*** from the heap by calling **`GET_CLOSEST`**; if the heap is empty, the algorithm has finished. Otherwise, do the following for each arc starting from node ***N***. Iterate over the arcs starting from the largest node number down to 0 so as to match our gold version exactly (doing so is also easier than the other direction).

- Call the other node ***M***.
- Does the arc (***N***, ***M***) exist? Remember that non-existent arcs have distance "infinity." If not, proceed to the next possible arc.
- Compute the cost to reach node ***M*** through node ***N***, which is the sum of the distance to node ***N*** and the distance of the arc (***N***, ***M***).
- If the distance computed is not less than the current distance to node ***M***, proceed to the next possible arc. Be sure to check correctly for "infinity" as the current distance to node ***M***.
- Set the distance to node ***M*** to the smaller value found through node ***N*** (called "relaxing" the distance), and set the predecessor for node ***M*** to node ***N***.
- If node ***M*** is not in the heap, add it to the end of the heap. Be sure to update both the new heap element and the heap index for node ***M***.
- Regardless of whether node ***M*** was in the heap, call PERCOLATE_UP on node ***M***'s heap element (using the heap index for node ***M***) to restore the heap property.
- That's all! Continue with the next arc.

How to Proceed

As always, read this document completely first.

Next, copy your working MP2 code to a new **`mp3.asm`** file. We usually allow you to make use of a peer's MP2 if you were not able to get yours working in time. However, **to avoid academic integrity violations, you must give explicit credit** to that person in your code and make clear exactly what code was taken from them, **and** you **must follow the rules about when** you are allowed to receive the code—obviously not before MP2 is due. Check Piazza for details.

You may also wish to read more online about the algorithm so as to gain a better understanding of what it does and how it works. If you do so, however, keep our earlier advice in mind so as to avoid confusion over terms. Also, be aware that Dijkstra was a famous computer scientist with many contributions, so searching simply for "Dijkstra's algorithm" may lead you to a completely different one.

Test Code Elements

In this MP, we have provided you with no code—you must extend your MP2 work and implement a program that uses your subroutines.

However, we have again given you a small variety of test cases in the **`tests`** subdirectory. These correspond to three graphs (the first two are quite similar). Start by assembling all three graph files as well as a copy of your **`mp3.asm`**. Then you can run any of the scripts, each of which is named **`s<graph#>_<source node #>`**. The corresponding output replaces "s" with "out." Read the instructions in MP2 again if you don't remember how to use scripts and **`diff`**.

Note that your final register values may be completely different, but the output of the program should match exactly. We may test more carefully when grading, and are probably going to use additional graphs. Feel free to make your own as well.

Specifics

- Your code must be written in LC-3 assembly language and must be contained in a single file called **`mp3.asm`**. We **will not grade** files with any other name.
- Do not make any assumptions unless they are stated explicitly to you in this document.
- Be sure to follow the rules and to credit your peer properly if you choose to use someone else's MP2 code (even if you only used the MP1 part!).
- Your main program must load the source node number from x3FFE into R0, call your **`DIJKSTRA`** subroutine, set up a stack in R6 (base at xC000), and print the paths to all nodes (node 0 first, then node 1, and so forth) using one call to **`FILL_STACK`** and one call to **`PRINT_PATH`** for each node.
- Your subroutine for implementing Dijkstra's algorithm must be called DIJKSTRA.
    - You may assume that a valid graph with no more than 100 nodes.
    - You may assume that all arc distances are non-negative.
    - You may assume that the longest valid path that your algorithm should explore will be have a total length (sum of arc distances) less than x4000. Note that you must check for "infinity" (x7FFF), which indicates a missing arc, and that overflow may occur if your algorithm is buggy.
    - You must implement Dijkstra's single-source shortest path algorithm as outlined in this document. If you use an outside source and your output fails to match, you may lose all functionality points.
- Your code must be well-commented, must include a header explaining the subroutine and another explaining the program, and must include a table describing how registers are used within the subroutine. Follow the style of examples provided to you in class and in the textbook.
- Neither your subroutine nor your program may access memory (neither to read nor to write) outside of your code and the regions provided for your use in this document, nor write to memory outside of your code unless specifically allowed in this document.
- Your **`DIJKSTRA`** subroutine may not produce any output to the display (neither directly nor indirectly). Neither may it read input from the keyboard.
- Your subroutine must operate correctly even when called many times without reloading code.

Testing

You are responsible for testing your code to make sure that it executes correctly and follows the specification exactly. See "Test Code Elements" for some help getting started, but note that the given files are meant as examples. You should create more tests to ensure that your code works correctly. You will also find it useful to step through your code in the simulator and inspect the state after each step to make sure that your code does exactly what you intended.

Grading Rubric

Functionality (70%)

- DIJKSTRA
  - 5% - initializes tables correctly
  - 10% - inserts source node correctly into heap
  - 5% - uses GET_CLOSEST to obtain next closest node and terminates correctly when heap is empty
  - 5% - skips missing (distance "infinity") arcs
  - 5% - computes distance through node being explored correctly
  - 5% - compares distance with current distance (including "infinity") correctly
  - 5% - relaxes distance correctly
  - 5% - inserts node at end of heap correctly when necessary
  - 5% - calls PERCOLATE_UP correctly
- program
  - 5% - calls DIJKSTRA correctly
  - 10% - uses FILL_STACK and PRINT_PATH correctly to print all shortest paths
  - 5% - terminates correctly

Comments, Clarity, and Write-up (25%)

- 10% - subroutine and program each have a paragraph explaining the purpose and interface (these are given to you; you just need to document your work)
- 10% - subroutine includes a table explaining how each register is used
- 10% - code is clear and well-commented

Note that some categories in the rubric may depend on other categories and/or criteria. For example, if your code does not assemble, you will receive no functionality points. If you name a subroutine incorrectly, you will receive no points for it, whether it works or not.

Finally, we will deduct **10 points** if the LC-3 VSCode extension developed by former ZJUI student Qi Li (look in the marketplace—you'll see his name) gives **even a single warning** on your assembly code. Fix them. Only the exceptions mentioned in MP2 are allowed.