Walker Mitchell
CS361
6/1/2019

Lab 4: DFA and Bellman-Ford

Task 1: DFA

This task was to implement a specific DFA given the 5 components of the tuple that defines it. I primarily used my notes from class, though I did check the textbook a couple times.

```
# Task 1: DFA
# For this task my sources are the textbook and my class notes.

def DFA(checkString):

    states = ("S", "Q1", "Q2", "R1", "R2")  # This is Q

    currentState = states[0]                # This is q1

    for i in range(0, len(checkString)):    # Process the string according to given table
        if checkString[i] not in ['a','b']: # If string has a character not in DFA's alphabet, reject it.
            return False

        if checkString[i] == 'a' and currentState in states[:3]:
            currentState = states[1]

        if checkString[i] == 'a' and currentState in states[3:]:
            currentState = states[4]

        if checkString[i] == 'b' and (currentState == states[0] or currentState in states[3:]):
            currentState = states[3]

        if checkString[i] == 'b' and currentState in states[1:3]:
            currentState = states[2]

    if currentState == states[1] or currentState == states[3]:  # Checks if DFA is in an accept state after processing string
        return True

    else:
        return False
```

Task 2: DFA Output

This task was to run Task 1's DFA code on 5 specific test strings: ababa, baba, aababaab, babaabaaabb and the empty string.

```
print("Task 2: Testing DFA code")

testString = "ababa"
print("Testing the string " + testString + ":")
print(DFA(testString))

testString = "baba"
print("Testing the string " + testString + ":")
print(DFA(testString))

testString = "aababaab"
print("Testing the string " + testString + ":")
print(DFA(testString))

testString = "babaaabaaabb"
print("Testing the string " + testString + ":")
print(DFA(testString))

testString = ""
print("Testing the empty string:")
print(DFA(testString))
```

```
Task 2: Testing DFA code
Testing the string ababa:
True
Testing the string baba:
False
Testing the string aababaab:
False
Testing the string babaaabaaabb:
True
Testing the empty string:
False
```

Task 3: Bellman-Ford

This task was to implement the Bellman-Ford algorithm for determining if a graph contains a negative weight cycle. For this task I referenced my class notes, but mostly used the textbook's pseudocode for INIT-SINGLE-SOURCE, RELAX, and the Bellman-Ford algorithm itself. Since INIT-SINGLE-SOURCE is only used as set up for the main algorithm, I just implemented it inside the primary function instead of writing it as a separate function. However, I did write RELAX as a separate function that the primary function calls. I chose to implement my graph using an adjacency list, so my Bellman-Ford and Relax functions are set up to use an adjacency list as the representation of the graph.

```
def relax(adjList, dist, pred, startVert, edgeIndex):
    endVert = adjList[startVert][edgeIndex][0]          # Pull edge's destination vertex out of adjacency list
    edgeWeight = adjList[startVert][edgeIndex][1]       # Pull edge's weight out of adjacency list
    if dist[endVert] > dist[startVert] + edgeWeight:
        dist[endVert] = dist[startVert] + edgeWeight
        pred[endVert] = startVert
```

```
def bellmanFord(adjList, sourceVert):

    # Initialize Single-Source
    predecessors = [None for i in range(len(adjList))]
    distances = [sys.maxsize for i in range(len(adjList))]
    distances[sourceVert] = 0

    # Actual Bellman-Ford logic
    for i in range(len(adjList)-1):
        for j in range(len(adjList)):
            for k in range(len(adjList[j])):
                relax(adjList, distances, predecessors, j, k)

    # Final check for negative weight cycle
    negCycle = False
    for j in range(len(adjList)):
            for k in range(len(adjList[j])):
                if distances[adjList[j][k][0]] > distances[j] + adjList[j][k][1]:
                    negCycle = True

    # Print distances, predecessors, and whether or not a negative weight cycle was found.
    print("Using vertex " + chr(65 + sourceVert) + " as the source vertex:")

    for i in range(len(adjList)):
        if(predecessors[i] == None):
            print("Vertex " + chr(65 + i) + " has distance " + str(distances[i]) + " and it has no parent.")
        else:
            print("Vertex " + chr(65 + i) + " has distance " + str(distances[i]) +
                  " and its parent is vertex " + chr(65 + predecessors[i]))

    if(negCycle == False):
        print("This graph has no negative weight cycle.")
    else:
        print("A negative weight cycle was found.")
```

Task 4: Bellman-Ford Output

This task was to run Task 3's algorithm on a given weighted digraph, output all vertices' distances and predecessors, and determine whether the graph contains a negative weight cycle. I chose to represent this graph as an adjacency list. Each index of this list contains a list of 2-tuples, where each tuple represents an outgoing edge starting at the vertex corresponding to the outer list's index. The first element of each tuple is the destination vertex, and the second is the edge's weight.

```
print("\nTask 4: Testing Bellman-Ford code")

# Each tuple represents an outgoing edge, with the first element being the
# destination vertex and the second element being the weight of the edge.
testGraphAdjList = [[(3,3)],                                  # A
                    [(0,-2)],                                 # B
                    [(1,1), (12,3)],                          # C
                    [(4,2), (5,6), (6,-1), (13,-1)],          # D
                    [(5,3)],                                  # E
                    [(7,-2)],                                 # F
                    [(7,1), (9,3)],                           # G
                    [(10,-1)],                                # H
                    [(7,-4)],                                 # I
                    [(8,2), (10,3)],                          # J
                    [],                                       # K
                    [(10,2)],                                 # L
                    [(11,-4)],                                # M
                    [(2,-3), (9,5), (12,8)]]                  # N

bellmanFord(testGraphAdjList, 0)
```

Since the lab prompt didn't specify which vertex to use as the source, I defaulted to using vertex A. As it turned out, a negative weight cycle does exist in this graph, namely {A, D, N, C, B, A}.

```
Task 4: Testing Bellman-Ford code
Using vertex A as the source vertex:
Vertex A has distance -8 and its parent is vertex B
Vertex B has distance -6 and its parent is vertex C
Vertex C has distance -9 and its parent is vertex N
Vertex D has distance -5 and its parent is vertex A
Vertex E has distance -3 and its parent is vertex D
Vertex F has distance 0 and its parent is vertex E
Vertex G has distance -6 and its parent is vertex D
Vertex H has distance -5 and its parent is vertex G
Vertex I has distance -1 and its parent is vertex J
Vertex J has distance -3 and its parent is vertex G
Vertex K has distance -6 and its parent is vertex L
Vertex L has distance -8 and its parent is vertex M
Vertex M has distance -4 and its parent is vertex C
Vertex N has distance -6 and its parent is vertex D
A negative weight cycle was found.
```