Programming Assignment 4 (Algorithms on a DAG, Bellman-Ford, Dijkstra's Algorithm, Johnson's Algorithm, and DJP)

Department of Computer Science, University of Wisconsin – Whitewater
Theory of Algorithms (CS 433)

Instructions For Submissions

- **This assignment is to be completed individually. If you are stuck with something, consider asking the instructor for help.**
- Submission is via Canvas as a single zip file.
- **Any function with a compilation error will receive a zero, regardless of how much it has been completed.**

---

1 Overview

Your task is to implement the following methods:

- `longestPaths` and `countOddEvenHops` in `DAG`
- `execute` in `BellmanFord`
- `execute` in `Johnson`

The project also contains additional files which you do not need to modify (but need to use).

1.1 Testing Correctness

Use `TestCorrectness` file to test your code. For each part, you will get an output that you can match with the output I have given to verify whether or not your code is correct. Output is provided in the `ExpectedOutput` file. You can use `www.diffchecker.com` to tally the output.

To test the correctness of the DAG algorithms, I have included 2 sample files: `dag1.txt`, and `dag2.txt`. To test the correctness of Bellman-Ford, I have included 3 sample files: `bellmanford1.txt`, `bellmanford2.txt`, and `bellmanford3.txt`. To test the correctness of Dijkstra, I have included 2 sample files: `dijkstra1.txt`, and `dijkstra2.txt`. To test the correctness of Johnson, I have included 3 sample files: `apsp1.txt`, `apsp2.txt`, and `apsp3.txt`. The corresponding graphs are shown in the next page.

Each .txt file has the following format. First line contains the number of vertices and edges respectively. Second line onwards are the edges in the graph; in particular, each line contains three entries: the source vertex, the destination vertex, and the length of the edge.
To test the correctness of DJP, I have included: `mst_graph.txt`; the corresponding graphs and MST are shown next.

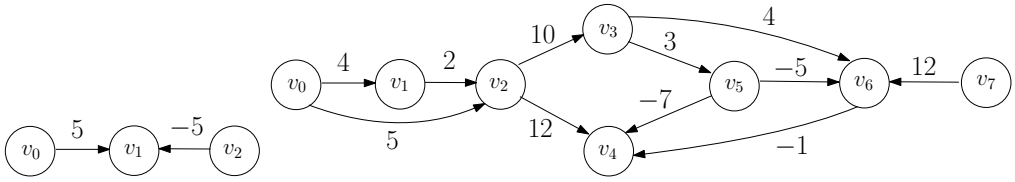

Figure 1: Graphs used for Testing DAG Algorithms

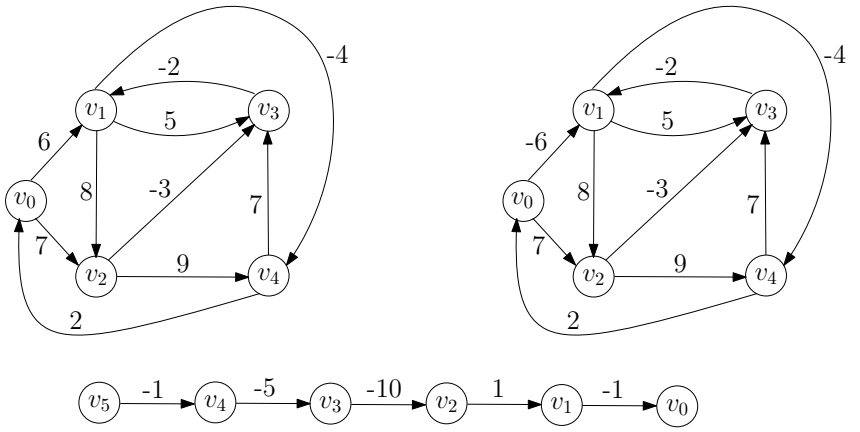

Figure 2: Graphs used for Testing Bellman-Ford Algorithm

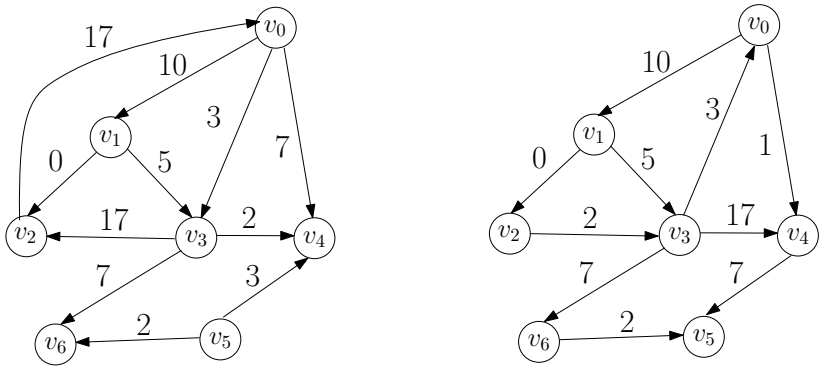

Figure 3: Graphs used for Testing Dijkstra's algorithm

1.2 C++ Helpful Hints

Use DYNAMIC ALLOCATION for declaring any and all arrays/objects. Remember to return an array from a function, you must use dynamic allocation. So, if you want to return an array x having length 10, it must be declared as **int *x = new int[10];** Also, to prevent a memory leak, the array should be deleted after its purpose has been served.

1.3 Multidimensional arrays

A 2d array is one which has fixed number of columns for each row, and *a jagged array* is one which has variable number of columns for each row.

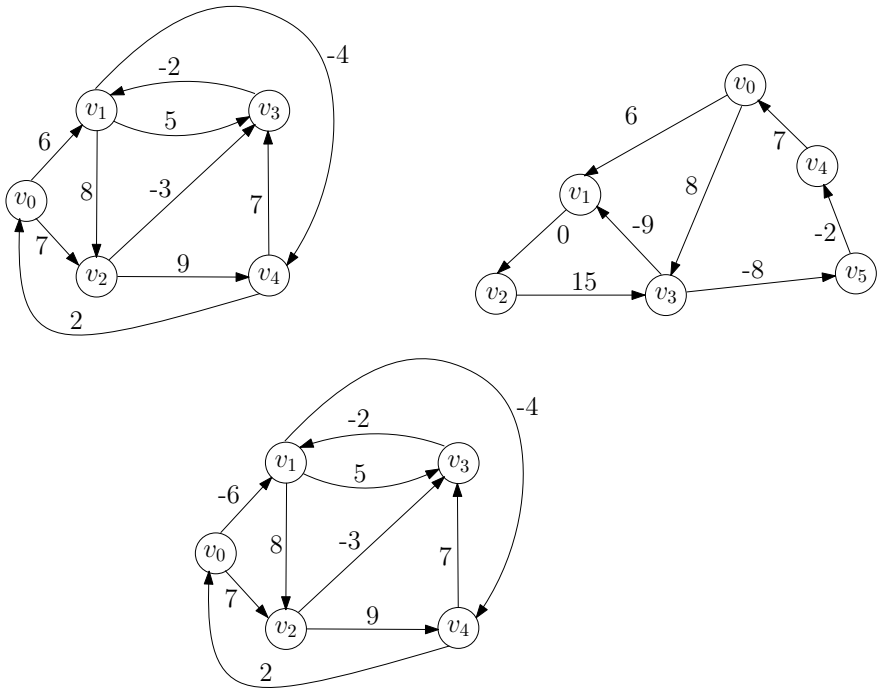

Figure 4: Graphs used for Testing Johnson's Algorithm

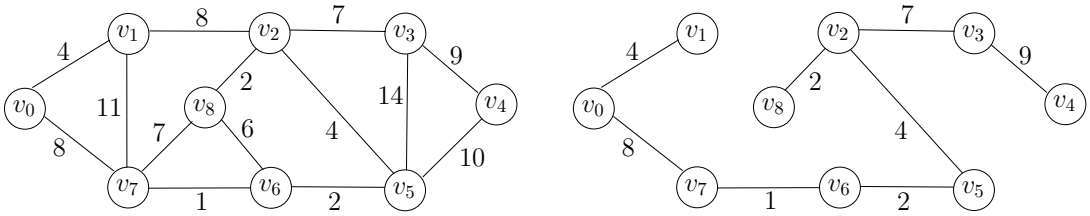

Figure 5: Graph (left) used for testing MST algorithms; corresponding MST on right

- **In C++**, although to create a 2d array/jagged array you don't need dynamic allocation, you'll need it to return the arrays from a function. Therefore, I'll discuss the dynamic allocation version. To create a 2d array/jagged array A that has r rows, the syntax is **int **A = new int*[r]**. To allocate c columns for row index i, the syntax is **A[i] = new int[c]**. Following is a code to return a jagged-array having *numRows* rows and *numCols* columns in each row.

```
int **jagged(int numRows, int *numCols) {
    int **C = new int*[numRows];
    for (int i = 0; i < numRows; i++)
        C[i] = new int[numCols[i]];
    return C;
}
```

- **In JAVA and C#**, to create a 2d array/jagged array A that has r rows, the syntax is **int[ ][ ] A = new int[r][ ]**. To allocate c columns for row index i, the syntax is **A[i] = new**

int[c].[1] Following is a code to return a jagged-array having *numRows* rows and *numCols* columns in each row.

```
int[][] jagged(int numRows, int[] numCols) {
    int[][] C = new int[numRows][];
    for (int i = 0; i < numRows; i++)
        C[i] = new int[numCols[i]];
    return C;
}
```

1.4 Dynamic Arrays

Here, you will use C++/Java/C# implementations of dynamic arrays, which are named respectively **vector**, **ArrayList**, and **List**.

- **In C++**, the syntax to create is `vector⟨int⟩ name`.

  To add a number (say 15) at the end of the vector, the syntax is `name.push_back(15)`.

  To remove the last number, the syntax is `name.pop_back()`.

  To access the number at an index (say 4), the syntax is `name.at(4)`.

  To find the length, the syntax is `name.size()`.

- **In Java**, the syntax to create is `ArrayList⟨Integer⟩ name = new ArrayList⟨Integer⟩()`.

  To add a number (say 15) at the end of the array list, the syntax is `name.add(15)`.

  To remove the last number, the syntax is `name.remove(name.size() - 1)`.

  To access the number at an index (say 4), the syntax is `name.get(4)`.

  To find the length, the syntax is `name.size()`.

- **In C#**, the syntax to create is `List⟨int⟩ name = new List⟨int⟩()`.

  To add a number (say 15) at the end of the vector, the syntax is `name.Add(15)`.

  To remove the last number, the syntax is `name.RemoveAt(name.Count - 1)`.

  To access the number at an index (say 4), the syntax is `name[4]`.

  To find the length, the syntax is `name.Count`.

1.5 Adjacency List: Representing Graphs in Memory

The vertices in the graph are numbered 0 through $n-1$, where n is the number of vertices. We use a two-dimensional jagged array **adjList** (called *adjacency list*) to represent the graph. Specifically, row index i in the array corresponds to the vertex v_i, i.e., row 0 corresponds to v_0, row 1 corresponds to v_1, and so on. Each cell in row i stores an outgoing edge of the vertex v_i. Each edge has 3 properties – *src*, *dest*, and *weight*, which are respectively the vertex from which the edge originates, the vertex where the edge leads to, and the edge weight. To get the number of outgoing edges of the vertex v_i, we simply get the length of the row at index i.

[1] For 2d arrays, you could use: **int A[ ][ ] = new int[r][c]** in JAVA and **int[,] A = new int[r,c]** in C#

In a nutshell, Edge is a class which has three integer variables – *src*, *dest*, and *weight*. The adjacency list, therefore, is a jagged array, whose type is Edge. In C++, we implement *adjList* as a vector of Edge vectors. In Java, we implement *adjList* as an ArrayList of Edge ArrayLists. In C#, we implement *adjList* as a List of Edge Lists.

2 Algorithms on DAG

You will write two algorithms.

2.1 Single Source Longest Paths in a DAG (35 points)

In the lecture notes, you have seen how to compute the shortest paths in a DAG after its topological order has been found. We remarked that the shortest path algorithm can be easily modified to find longest paths. Here, we will implement the same; moreover, we will find longest paths and compute topological order at the same time. See the pseudo-code in the next page.

A Remark about Implementation: Note that we are setting the *dist* array entries to very small values (INT_MIN for C++, $Integer.MIN\_VALUE$ for Java, and $INT32.MinValue$ for C#) to simulate $-\infty$. Let us assume that we have a graph

$$v_0 \rightarrow v_1 \leftarrow v_2$$

where the edge weight from v_0 to v_1 is 5 and the edge weight from v_2 to v_1 is -5. A topological order is v_0, v_2, v_1. If we compute the longest path from v_0, initially $distance[1]$ would be set to

$$distance[1] = distance[0] + 5 = 0 + 5 = 5$$

Now on processing v_2, we will compute $(distance[2] - 5)$, which will be a large positive number due to rounding of numbers. This will lead to $distance[1]$ being erroneously set to a positive number larger than 5 (which is incorrect as there is no way to reach v_2 from v_0).

Hence, if *distance* of the dequeued vertex equals $-\infty$, we simply skip the remaining code inside the outer for-loop. Complete the `longestPaths` function in the `DAG` class using the following steps:

- Call the the `topoSort` function to get the topological order of the graph
- Allocate $numVertices$ cells for an integer array $distance[\,]$
- Use a loop to initialize all cells of $distance[\,]$ to $-\infty$

  In C++, use INT_MIN for $-\infty$. In Java, use Integer.MIN_VALUE for $-\infty$. In C#, use Int32.MinValue for $-\infty$.
- Set $distance[s] = 0$
- for each vertex v in the topological order, do the following:
  - for each outgoing edge $adjEdge$ of the vertex v, do the following:
    - let $adjVertex$ be the destination of $adjEdge$
    - if $(distance[v] \neq -\infty)$, do the following:
      - let $len = distance[v]+$ weight of $adjEdge$
      - if $(len > distance[adjVertex])$, set $distance[adjVertex] = len$

- Return the distance array

2.2 Counting Odd-Even Hops in a DAG (35 points)

Given a start vertex s, we want to compute the number of paths from s to every vertex t, such that the paths have an even number of edges. We also want to compute the number of paths from s to every vertex t, such that the paths have an odd number of edges. Let's see how we can solve this problem via a dynamic program.

We define a function $countEven(v)$ which counts the number of paths from s to v which have an even number of edges. Likewise, define a function $countOdd(v)$ which counts the number of paths from s to v which have an odd number of edges. For any vertex v, let $incoming(v)$ be the edges of the graph that have an edge to v. Then, we have the following base-cases:

- $countEven(s) = 1$. We can go from s to s using zero (an even number) edges.
- $countOdd(s) = 0$. We cannot go from s to s using an odd number of edges.

Now, we have the following recurrences:

- $countEven(v) = \sum_{(u,v) \in incoming(v)} countOdd(u)$. Consider a vertex u which has an edge to the vertex v. If there are x many paths from s to u that have an odd number of edges, then there are exactly x many paths from s to v via the vertex u that have an even number of edges – just extend a path to u by the edge from u to v. So, we just sum the number of odd-hop paths to u for every vertex u which have edges to v.
- $countOdd(v) = \sum_{(u,v) \in incoming(v)} countEven(u)$. Same arguments as in the previous case.

Using these recursion rules, fill up the `countOddEvenHops` function with a bottom-up dynamic program to compute the number of even-hop paths and the number of odd-hop paths from s to every vertex in the graph. Your code must run in $O(V + E)$ time for any credit. It will return the answer as a 2d array which two rows: the first row contains the number of even-hop paths and the second row contains the number of odd-hop paths.

Memoization Hint. You should be able to use a lot of the logic/code-structure from the `longestPaths` function. See how you can adapt with these modified recursion rules.

3 Bellman-Ford (60 points)

Complete the `execute` method in the `BellmanFord` class using the following steps:

- Create an integer array $dist[\,]$ of size $numVertices$.
- Initialize each cell of $dist[\,]$ to ∞. Initialize a boolean $didDistChange$
- Set $dist[source]$ to 0.
- For $i = 1$ to $numVertices - 1$ (both inclusive), do the following:
  - Set $didDistChange$ to $false$
  - For each edge e in the graph, do the following:[a]

        * If $dist$[source of e] is ∞, then continue
        * Set $newDist = dist$[source of e] + weight of e
        * If $newDist < dist$[destination of e], then set $dist$[destination of e] = $newDist$ and set $didDistChange = true$
    - if ($dist$ did not change), return $dist$

- For each edge e in the graph, do the following:
    - If ($dist$[source of e] = ∞), then continue
    - If ($dist$[source of e] + weight of e < $dist$[destination of e]), then return *null*

- return $dist$

[a] Run a loop from $j = 0$ to $j < numVertices$ and a nested loop from $k = 0$ to $k <$ the length of the j^{th} row of $adjList$. An edge is given by the k^{th} cell of the j^{th} row of $adjList$.

4 Dijkstra (no coding/submission required)

You must have already implemented Dijkstra's algorithm in COMPSCI 223: Data Structures; so, we are not doing it again. I have provided the code because it is needed for Johnson's algorithm. Also, you may want to compare it against DJP to understand the difference between the two.

5 Johnson (70 points)

Implement the `execute` method in `Johnson` class.

- Add a blank row to $adjList$. This is creating the dummy vertex. Syntax:
    - **C++:** $adjList.push\_back(vector\langle Edge\rangle())$;
    - **Java:** $adjList.add(new\ ArrayList\langle Edge\rangle())$;
    - **C#:** $adjList.Add(new\ List\langle Edge\rangle())$;
- Run a loop from $i = 0$ to $i < numVertices$. Within the loop,
    - Create an edge e with src as $numVertices$ (which is the dummy vertex), destination as i (which is a vertex in the graph), and weight 0.
    - Add e at the last row of $adjList$, i.e., at index $numVertices$. To add e, first obtain the last row and then add e.
- Increment $numEdges$ by $numVertices$ and $numVertices$ by one
- Create a BellmanFord object for this graph.

  You are simply going to call the BellmanFord class constructor with the argument as *this*. Essentially, you are using polymorphism here. Johnson and BellmanFord classes both extend Graph class; so passing *this* would mean that the Graph part of Johnson object is embedded into BellmanFord object (as desired).

- Obtain the $\Phi[\,]$ array by executing BellmanFord from the dummy $(numVertices - 1)$
- Decrement $numVertices$ by one and $numEdges$ by $numVertices$. Remove the last row of $adjList$.
- If Φ is $null$, then return $null$
- For each edge e in the graph, modify its edge weight using the Φ array as

$$e's\ weight = e's\ weight + \Phi[e's\ source] - \Phi[e's\ destination]$$

- Create a 2d array $allPairMatrix$ having $numVertices$ rows.
- Create a Dijkstra object for this graph. Once again, you are simply going to call the Dijkstra class constructor with the argument as $this$.
- For $i = 0$ to $i < numVertices$, set $allPairMatrix[i]$ to the array returned by executing Dijkstra's algorithm for the source i
- Run a loop from $i = 0$ to $i < numVertices$, and a nested loop from $j = 0$ to $j < numVertices$. Within the inner loop, if $i \neq j$ and $allPairMatrix[i][j] \neq \infty$, then set $allPairMatrix[i][j] = allPairMatrix[i][j] - \Phi[i] + \Phi[j]$
- For each edge in the graph, revert back to its original edge weight using the Φ array.
- Return $allPairMatrix$;

6 DJP (no coding/submission required)

Recall that DJP is essentially the same as Dijkstra's algorithm with the only essential change being in how the label of a node gets updated. So, here's how the codes are different:

- DJP can start at any vertex; the Minimum Spanning Tree may look a little different depending on the vertex we start at (which is okay). We start at the last vertex.
- If we remove u from the priority queue and relax an edge (u, v) in Dijkstra's algorithm, then $distance[v]$ is set to $distance[u] + edgeWeight(u, v)$ if we obtain a smaller estimate. In DJP, on the other hand, we will set $label[v] = edgeWeight(u, v)$ if we obtain a smaller estimate. Here, $label[\,]$ is the equivalent of the $distance[\,]$ array.
- Also, we need to retrieve the edges of the MST at the end. To this end, we use an (dynamic) array (of type `Edge`) where we will store the edges of the minimum spanning tree. Call this (dynamic) array $parent$. Suppose, we relax an edge $e = (u, v)$ and we set $label[v] = edgeWeight(u, v)$, then you will update $parent[v] = e$.
- At the end, $parent$ will contain all the edges of the minimum spanning tree, as well as one extra edge – a null edge for the start vertex – we need to remove it. That's why we started at the end, because removal at the end of a dynamic array takes $O(1)$ time, whereas removal at the beginning takes linear time.