Programming Assignment 4 (Binary Strings Without Consecutive Ones, Longest Increasing Subsequence, Bellman-Ford, Dijkstra, and All-Pair Shortest Paths)

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

Instructions For Submissions

- **Each group to have at most 2 members**. Although you can work individually, I encourage you to get a partner.
- One submission per group. **Mention the name of all members.**
- **Submit code and a brief report**. Submission is via Canvas as a single zip file.
- No need to include the algorithm description in the report.

---

1 Overview

We are going to omplement a couple of Dynamic Programming algorithms, and find shortest paths in graphs. To this end, **your task is to implement the following methods**:

- `numberOfBinaryStringsWithNoConsecutiveOnes` in `DynamicProgramming`
- `longestIncreasingSubsequence` in `DynamicProgramming`
- `execute` in `BellmanFord`
- `execute` in `Dijkstra`
- `execute` in `Johnson`
- `execute` in `FloydWarshall`

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 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 and Floyd-Warshall, I have included 3 sample files: `apsp1.txt`, `apsp2.txt`, and `apsp3.txt`. The corresponding graphs are shown below.

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.

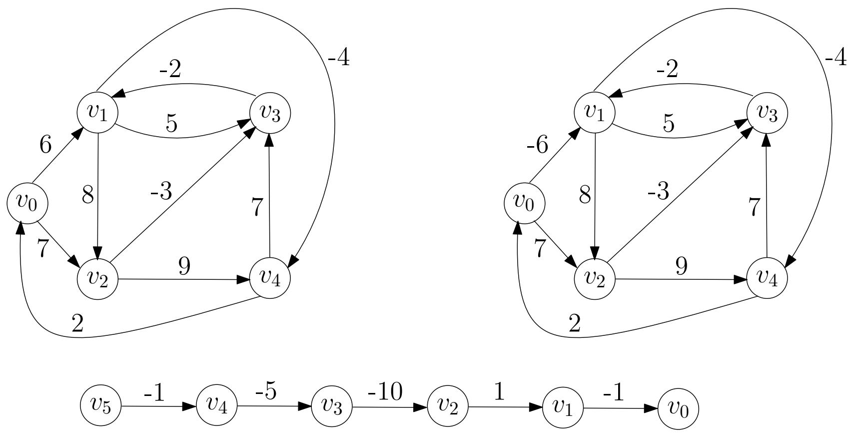


Figure 1: Graphs used for Testing Bellman-Ford Algorithm

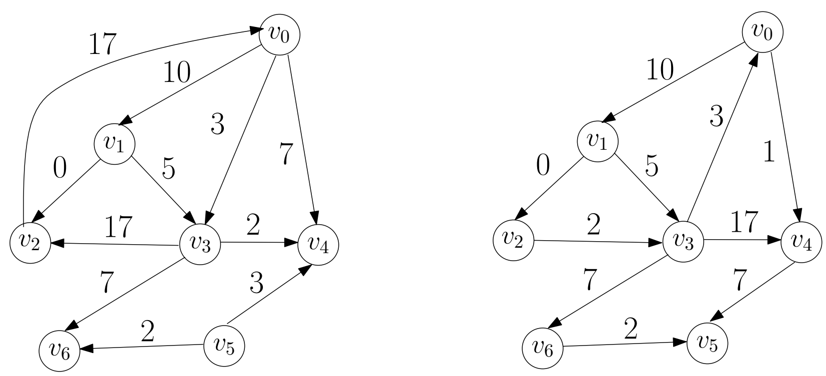


Figure 2: Graphs used for Testing Dijkstra's algorithm

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];**

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.

- **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

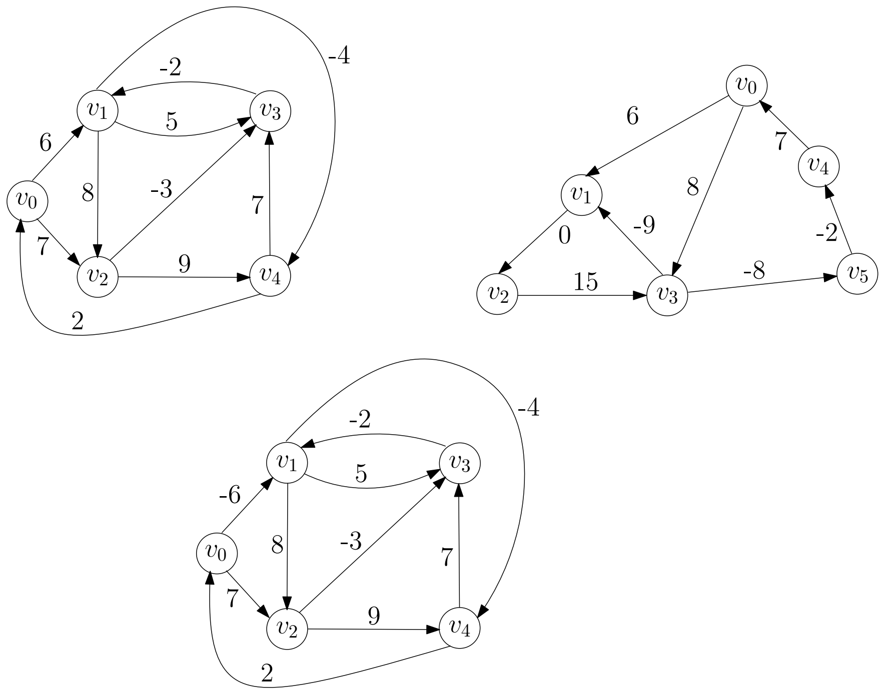


Figure 3: Graphs used for Testing All-Pair Shortest Paths Algorithm

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

You do not need to write code for a priority queue, but you need to use it for Dijkstra's algorithm. Your task will be to create a priority queue and use its main operations – `setPriority`, `getMinimumItem`, and `deleteMinimum`.

- **In C++**, to create an integer priority queue, the syntax is:
  `PriorityQueue<int> pq;` To set the priority of an item i to priority p, the syntax is `pq.setPriority(i, p);` To get the item with the minimum priority, the syntax is `pq.getMinimumItem();` To delete the item with the minimum priority, the syntax is `pq.deleteMinimum();`

- **In Java**, to create an integer priority queue, the syntax is:
  `PriorityQueue<Integer> pq = new PriorityQueue<Integer>();` To set the priority of an item i to priority p, the syntax is `pq.setPriority(i, p);` To get the item with the minimum priority, the syntax is `pq.getMinimumItem();` To delete the item with the minimum priority, the syntax is `pq.deleteMinimum();`

- **In C#**, to create an integer priority queue, the syntax is:
  `PriorityQueue<int> pq = new PriorityQueue<int>();` To set the priority of an item i to priority p, the syntax is `pq.setPriority(i, p);` To get the item with the minimum priority, the syntax is `pq.getMinimumItem();` To delete the item with the minimum priority, the syntax is `pq.deleteMinimum();`

1.6 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.

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 Java, we implement *adjList* as a List of Edge Lists.

2 Dynamic Programming

You are going to write code for a couple of dynamic programming problems. As opposed to a detailed pseudo-code, I have deliberately omitted some of the details. You should check out the following videos for explanations, which is likely to help you with coding and debugging:

- https://drive.google.com/file/d/1yovdEpEphu0sHppyd41LBUAAEQL-GZ5B/view?usp=sharing
- https://drive.google.com/file/d/1TA2GMo_tRgBwGqxDjTnkpm2g6iOkl1bq/view?usp=sharing
- https://drive.google.com/file/d/1DjYYLLe5nU-O3dRaIbOOsVqLaBWo5goV/view?usp=sharing

2.1 Binary Strings with No Consecutive Ones

Complete the `numberOfBinaryStringsWithNoConsecutiveOnes` method to find the number of binary strings of length n with no consecutive ones. If $B(n)$ is the answer for an n-length string, then $B(1) = 2$, $B(2) = 3$, and for any $n > 2$, we have $B(n) = B(n-1) + B(n-2)$.

You must write a bottom-up dynamic program that uses only $O(1)$ space, i.e., it does not use an array for storage, but a few variables is fine.

2.2 Longest Increasing Subsequence

Complete the `longestIncreasingSubsequence` method to find the longest increasing subsequence. Once again you should use a bottom-up dynamic program. Here's a few helpful steps:

- Create two integer arrays LIS and $PRED$ both of lengths len.
- For $i = 0, 1, 2, 3, \ldots, len-1$, do the following:
  - Set $LIS[i] = 1$ and $pred[i] = -1$.
  - Among the indexes $0, 1, 2, \ldots, i-1$, find the index $maxIndex$ such that $arr[maxIndex] < arr[i]$ and $LIS[maxIndex]$ is the maximum of all the values among $LIS[0], LIS[1], \ldots, LIS[i-1]$. If the values $arr[0], arr[1], \ldots, arr[i-1]$ are all greater than $arr[i]$, then let $maxIndex = -1$
  - If $maxIndex \neq -1$, set $LIS[i] = LIS[maxIndex] + 1$ and $PRED[i] = maxIndex$
- Find $lisIndex$, which is the index containing the maximum value in the $LIS[\,]$ array
- Create a dynamic integer array.
- Starting from $lisIndex$ and by using the $PRED$ array, add the values in the longest increasing subsequence to the dynamic array
- Reverse the dynamic array using the given helper function, and then return it.

3 Bellman-Ford

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 ∞.
- Set $dist[source]$ to 0.
- Initialize a boolean $didDistChange$ to $false$.
- For $i = 1$ to $numVertices - 1$ (both inclusive), do the following:
  - Initialize $didDistChange$ 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] $= \infty$), 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's Algorithm

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

We maintain a boolean array $closed[\,]$, where $closed[v] = true$ indicates that vertex v is closed. Thus, to close a vertex, set corresponding entry in $closed$ array to $true$.

$open$ is implemented as a priority queue. You are going to use three methods of the PriorityQueue class: `setPriority(int item, int priority)`, `getMinimumItem()`, `deleteMinimum()`. The first one is used to set the distance of a vertex, and the latter two are used to get and delete the vertex with the minimum distance label. There is an open vertex if the priority queue's size is at least one; to check that use the `getSize()` method of the priority queue.

- Create an integer $distance$ array of size $numVertices$ and set all its cells to ∞. Create a boolean array $closed$ of size $numVertices$ and set all its cells to $false$
- Create an integer priority queue $open$. Add the vertex $source$ to $open$ with priority 0. Set $distance[source]$ to 0.

- While open is not empty, do the following:
  - Let $minVertex$ be an open vertex with the minimum $distance$ value. Use the `getMinimumItem()` of *open* to get this vertex. Delete the minimum in *open* by calling `deleteMinimum()`.
  - Close $minVertex$
  - For each adjacent edge $adjEdge$ of $minVertex$ do the following:[a]
    * Let $adjVertex$ be the destination of $adjEdge$
    * If $adjVertex$ is not closed, do the following:
      · Set $newDist$ to $distance[minVertex] + adjEdge's\ weight$
      · If $newDist < distance[adjVertex]$, then {
        set $distance[adjVertex]$ to $newDist$
        set priority of $adjVertex$ to $newDist$ in $open$
      }

[a]Each adjacent vertex can be found using the $adjList$

5 Johnson's Algorithm

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 $(numVertices - 1)$
- Remove the last row of $adjList$

- Decrement $numVertices$ by one and $numEdges$ by $numVertices$
- 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 Floyd-Warshall

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

- Create a 2d array $allPairMatrix$ having $numVertices$ rows and columns.
- Set all cells of $allPairMatrix$ to ∞
- Set the cells of $allPairMatrix$ lying on the major diagonal to 0
- For each $edge$ in the graph, set $allPairMatrix[edge's\ source][edge's\ destination] = edge's\ weight$
- Run three nested for-loops from $k = 0$ to $k < numVertices$, $i = 0$ to $i < numVertices$, and $j = 0$ to $j < numVertices$. Within the innermost (i.e., j-loop), do the following:
  - If ($allPairMatrix[i][k]$ or $allPairMatrix[k][j]$ equals ∞), then continue
  - Set $tempDist = allPairMatrix[i][k] + allPairMatrix[k][j]$
  - If ($allPairMatrix[i][j] > tempDist$), then set $allPairMatrix[i][j] = tempDist$
- If any major diagonal value of $allPairMatrix$ is less than 0, then return null
- return $allPairMatrix$;

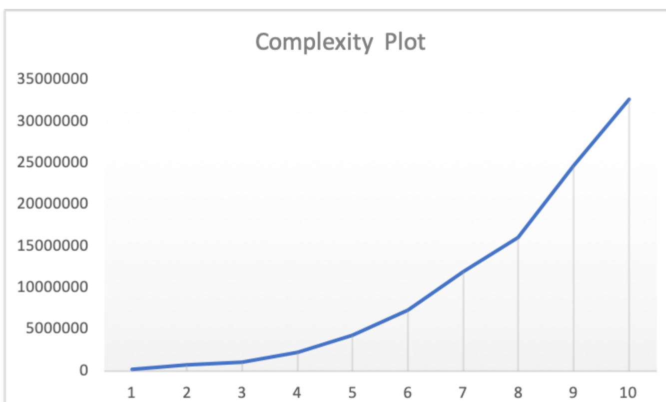
7 Report (10 points)

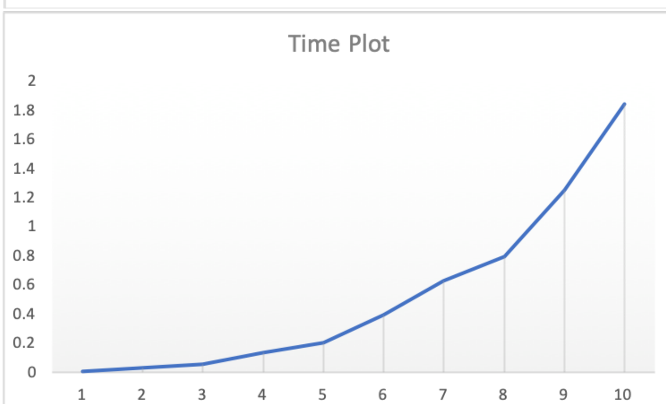
Start by downloading the large graph data from the `RoadNetworks` folder and appropriately set the *DIJKSTRA_FOLDER* variable in `TestTime` file.[2] Answer the following questions:

- What is the complexity of the Dijkstra's algorithm implementation in this assignment?
- Run the `TestTime` file (it'll take some time) and analyze the output in accordance with the complexity above.[3] In particular, do the following:
  - Check the `DijkstraPlot` excel sheet provided to you. The number of vertices and edges are provided in columns C and D for each of the states. Write the formula for the complexity in column E. In particular, if you have claimed that the complexity in the previous question is $O(M^2)$, then formula in cell $E5 = D5 * D5$.
  - Plot the times that you get or each of the states in the cells $E20$ through $E29$
  - Once you fill in the formula/numbers in the columns above, you should obtain two graphs. Do they look similar or different?

Answer these and submit your responses along with the excel that you obtain. Please do not upload the large data files to Canvas.

Here's what my plots look like. They are extremely similar to each other giving a strong indication that my implementation is aligned with the theoretical bound.





[2] Original data from http://users.diag.uniroma1.it/challenge9/download.shtml.

[3] The files are small enough so that the code can completely run, but if for some reason, your computer is unable to handle all the files, then go as far as you can.