1 Introduction .................................................. 2
2 A Recursive Observation .................................. 2
3 Floyd’s Algorithm .......................................... 3
4 Time Complexity ............................................ 5
5 Negative Cycles ............................................. 5
6 Path Reconstruction ........................................ 6
  6.1 Algorithm Modification ................................. 6
  6.2 Shortest Path Extraction .............................. 7
1 Introduction

Suppose we have a directed and weighted graph in which both positive and negative (and zero) weights are allowed and we wish to find a shortest path between any two vertices.

The first thing to note is that because we permitting negative weights we could possibly get paths with arbitrarily low (negative) distances if the graph had a cycle with total negative weight since we could just follow the cycle arbitrarily many times. Thus we will insist that the graph has no cycles with negative total weight.

Note that the graph may have cycles with zero or positive total weight.

2 A Recursive Observation

The core argument which underlies Floyd’s Algorithm is recursive in nature and proceeds as follows:

Suppose our graph has \( n \) vertices labeled \( \{1, 2, \ldots, n\} \). For vertices \( i \) and \( j \) and for \( 1 \leq k \leq n \) denote by \( sp(i, j, k) \) the length of the shortest path from \( i \) to \( j \) permitting intermediate vertices \( \{1, 2, \ldots, k\} \).

Note 2.0.1. Since \( \{1, 2, \ldots, k\} \) are intermediate vertices they do not need to include \( i \) and \( j \).

It follows immediately that because having additional vertices means a possibly shorter (definitely not longer) shortest path we have:

\[
sp(i, j, k) \leq sp(i, j, k - 1)
\]

It follows from this that for vertices \( i \) and \( j \) the shortest path between them either uses \( \{1, 2, \ldots, k - 1\} \) (and not \( k \)) or \( \{1, 2, \ldots, k\} \) (including \( k \)):

\[
sp(i, j, k) = \min (sp(i, j, k - 1), sp(i, k, k - 1) + sp(k, j, k - 1))
\]

Now then, if the shortest path from \( i \) to \( j \) does use \( k \) then it consists of a shortest path from \( i \) to \( k \) permitting intermediate vertices \( \{1, 2, \ldots, k - 1\} \) followed by a shortest path from \( k \) to \( j \) permitting intermediate vertices \( \{1, 2, \ldots, k - 1\} \).

Thus we have:

\[
sp(i, j, k) = \min (sp(i, j, k - 1), sp(i, k, k - 1) + sp(k, j, k - 1))
\]

In addition we have the base case; When \( k = 0 \) we are permitting no intermediate vertices at all and hence:
\[ sp(i, j, 0) = \begin{cases} 
  w(i, j) & \text{If there is a (directed!) edge from } i \text{ to } j \\
  \infty & \text{Otherwise}
\end{cases} \]

### 3 Floyd’s Algorithm

This leads to a dynamic programming algorithm which calculates \( sp(i, j, k) \) for all \( i, j \) first for \( k = 0 \), then \( k = 1 \), then \( k = 2 \), and so on until \( k = n \). The algorithm is as follows:

```plaintext
d = n x n array all infinity
for each edge (u, v):
    d[u][v] = w[u][v]
for each vertex v:
    d[v][v] = 0
for k = 1 to n:
    for i = 1 to n:
        for j = 1 to n:
            if d[i][k] + d[k][j] < d[i][j]:
                d[i][j] = d[i][k] + d[k][j]
        end if
    end for
end for
```

Note that \( d[x][y] \) is the distance specifically from \( x \) to \( y \) and after each iteration of \( k \) (including before the critical third loop happens, treating \( k = 0 \)) we have \( d[x][y] \) storing \( sp(x, y, k) \).

Pay close attention to how this works. We first initialize an array of all \( \infty \), then we update this array using the directed edges in the graph (giving them noninfinite weight) and then we update this array using the vertices (the distance from each vertex to itself is 0).

We then iterate from \( k = 1, 2, ..., n \) and for each iteration we update the entire array such that:

1. Before the \( k = 1 \) iteration starts the array contains the lengths of the shortest paths permitting intermediate vertices \( \{\} \).
2. After the \( k = 1 \) iteration is done the array contains the lengths of the shortest paths permitting intermediate vertices \( \{1\} \).
3. After the \( k = 2 \) iteration is done the array contains the lengths of the shortest paths permitting intermediate vertices \( \{1, 2\} \).
4. ...
5. After the \( k = n \) iteration is done the array contains the lengths of the
shortest paths allowing all the vertices as intermediates.

Let’s see how this works with an example.

**Example 3.1.** Here is a weighted simple directed graph:

```
2
/|
/ |
1
/  
3  1
  |
  3
 7
  |
  |
1
```

We’ll show the array \( d \) as a table. We first initialize and execute the first two `for` loops. At this point \( d[i][j] = \text{sp}(i, j, 0) \):

<table>
<thead>
<tr>
<th>init</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>3</td>
<td>\infty</td>
<td>\infty</td>
</tr>
<tr>
<td>2</td>
<td>\infty</td>
<td>0</td>
<td>1</td>
<td>-2</td>
</tr>
<tr>
<td>3</td>
<td>-3</td>
<td>\infty</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>\infty</td>
<td>\infty</td>
<td>\infty</td>
<td>0</td>
</tr>
</tbody>
</table>

We then perform the \( k = 1 \) iteration. At this point \( d[i][j] = \text{sp}(i, j, 1) \):

- \( d[3][1] + d[1][2] = -3 + 3 = 0 < \infty = d[3][2]. \)

Thus the result is:

<table>
<thead>
<tr>
<th>( k = 1 )</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>3</td>
<td>\infty</td>
<td>\infty</td>
</tr>
<tr>
<td>2</td>
<td>\infty</td>
<td>0</td>
<td>1</td>
<td>-2</td>
</tr>
<tr>
<td>3</td>
<td>-3</td>
<td>0</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>\infty</td>
<td>\infty</td>
<td>\infty</td>
<td>0</td>
</tr>
</tbody>
</table>

We then perform the \( k = 2 \) iteration. At this point \( d[i][j] = \text{sp}(i, j, 2) \):

- \( d[1][2] + d[2][3] = 3 + 1 = 4 < \infty = d[1][3]. \)
- \( d[1][2] + d[2][4] = 3 + -2 = 1 < \infty = d[1][4]. \)

Thus the result is:
We then perform the $k = 3$ iteration. At this point $d[i][j] = sp(i, j, 3)$:


Thus the result is:

<table>
<thead>
<tr>
<th>$k = 3$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>-2</td>
<td>0</td>
<td>1</td>
<td>-2</td>
</tr>
<tr>
<td>3</td>
<td>-3</td>
<td>0</td>
<td>0</td>
<td>-2</td>
</tr>
<tr>
<td>4</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>0</td>
</tr>
</tbody>
</table>

We then perform the $k = 4$ iteration. There are no updates. At this point $d[i][j] = sp(i, j, 4)$:

<table>
<thead>
<tr>
<th>$k = 3$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>-2</td>
<td>0</td>
<td>1</td>
<td>-2</td>
</tr>
<tr>
<td>3</td>
<td>-3</td>
<td>0</td>
<td>0</td>
<td>-2</td>
</tr>
<tr>
<td>4</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>0</td>
</tr>
</tbody>
</table>

4 Time Complexity

It is fairly easy to see that the time complexity here is $\Theta(n^3)$ due to the nested loops. This is best-, worst-, and average-case.

5 Negative Cycles

As mentioned in the introduction, if the graph contains a negative cycle then for any two vertices in that cycle there are arbitrarily shortest (negative length) paths between those vertices. Because of that, asking for “the shortest path” is an irrelevant question. However there are two things to note:

**Note 5.0.1.** If the graph does contain a negative cycle then for any vertex $i$ in the cycle Floyd’s Algorithm will result in $d[i][i]$ being equal to the total (negative) length of that cycle. In this way Floyd’s Algorithm can be used to detect the existence of negative cycles by simply checking the main diagonal of $d$ for negative values.

**Note 5.0.2.** You are encouraged to consider what might the result of Floyd’s
Algorithm might be on graphs with negative cycles for values which are not on the main diagonal.

6 Path Reconstruction

Although Floyd’s algorithm does not in and of itself give the actual paths between the vertices it is easy enough to do this.

6.1 Algorithm Modification

In addition to storing the distances between vertices we fill another array \( p \) for which, once the algorithm is complete, \( p[i][j] \) contains the predecessor of \( j \) along the shortest path from \( i \) to \( j \).

Here is the modified pseudocode:

\[
\begin{align*}
d & = n \times n \text{ array all infinity} \\
p & = n \times n \text{ array all null} \\
\text{for each edge } (u,v): & \quad d[u][v] = w[u][v] \\
& \quad p[u][v] = u \\
\text{for each vertex } v: & \quad d[v][v] = 0 \\
& \quad p[v][v] = v \\
\text{for } k = 1 \text{ to } n: & \quad \\
& \quad \text{for } i = 1 \text{ to } n: \\
& \quad \quad \text{if } d[i][k] + d[k][j] < d[i][j]: \\
& \quad \quad \quad \quad d[i][j] = d[i][k] + d[k][j] \\
& \quad \quad \quad \quad p[i][j] = p[k][j] \\
& \quad \quad \end{\text{if}} \\
& \quad \end{\text{for}} \\
& \end{\text{for}} \\
\end{align*}
\]

In the first for loop for each edge from \( u \) to \( v \) we assign \( p[u][v] = u \). This is essentially saying that if we are permitted no intermediate vertices then along the shortest path from \( u \) to \( v \) the predecessor to \( v \) should be \( u \), which makes sense.

In the second for loop for each vertex \( v \) we assign \( p[v][u] = v \). This is essentially saying that if we are permitted no intermediate vertices then along the shortest path from \( v \) to \( v \) the predecessor to \( v \) should be \( v \), which makes sense.

In the third and main for loop consider what the situation is after the \( k \) iteration (including \( k = 0 \) before the loop starts). We have \( p[i][j] \) storing the predecessor of \( j \) along the shortest path from \( i \) to \( j \) permitting intermediate
vertices $\{1, 2, \ldots, k\}$.

Suppose we are executing the $k + 1$ iteration for a given $i$ and $j$, meaning we are now permitting intermediate vertices $\{1, 2, \ldots, k + 1\}$. If the conditional is satisfied this implies that the shortest path from $i$ to $j$ ought to go through vertex $k + 1$ and hence it will have this form:

$$i \rightarrow \ldots \text{maybe more vertices...} \rightarrow k + 1 \rightarrow \ldots \text{maybe more vertices...} \rightarrow j$$

Note that each of these two shortest (sub)paths permits intermediate vertices $\{1, 2, \ldots, k\}$.

The predecessor for $j$ along this new shortest path from $i$ to $j$ permitting intermediate vertices $\{1, 2, \ldots, k + 1\}$ ought then to be the predecessor of $j$ along the shortest path from $k + 1$ to $j$ permitting intermediate vertices $\{1, 2, \ldots, k\}$ and this is precisely what is stored in $p[k + 1][j]$, hence we ought to assign $p[i][j] = p[k + 1][j]$ which is precisely what the $k + 1$ iteration does.

Observe that if there is no possible path at all from some vertex $u$ to $j$ then $p[u][v]$ will be assigned null at the start and will remain so for the duration.

### 6.2 Shortest Path Extraction

Once this pseudocode executes we can then use the $p$ array to reconstruct a given path easily. Given vertices $u$ and $v$ we simply look at predecessors of $v$ until we get back to $u$. Note that if $p[u][v] = \text{null}$ then this means that no path from $u$ to $v$ was ever found.

```python
u, v = two vertices
path = []
if p[u][v] != null:
    path = [v]
    while v != u:
        v = p[u][v]
        path.prepend(v)
    end while
end if
```

Let’s see how this works with our example from earlier.

**Example 6.1.** Here is our weighted simple directed graph:
We’ll show the array \( p \) as a table. After initialization and the first two for loops we have the following:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>2</td>
<td>None</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>None</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>4</td>
</tr>
</tbody>
</table>

We then perform the \( k = 1 \) iteration. Note that:

- \( d[3][1] + d[1][2] = -3 + 3 = 0 < \infty = d[3][2] \). Thus the shortest path from 3 to 2 permitting intermediate vertices \( \{1\} \) passes through 1 and so we set \( p[3][2] = p[1][2] = 1 \).

Thus the result is:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>2</td>
<td>None</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>4</td>
</tr>
</tbody>
</table>

We then perform the \( k = 2 \) iteration. Note that:

- \( d[1][2] + d[2][3] = 3 + 1 = 4 < \infty = d[1][3] \). Thus the shortest path from 1 to 3 permitting intermediate vertices \( \{1, 2\} \) passes through 2 and so we set \( p[1][3] = p[2][3] = 2 \).
- \( d[1][2] + d[2][4] = 3 + -2 = 1 < \infty = d[1][4] \). Thus the shortest path from 1 to 4 permitting intermediate vertices \( \{1, 2\} \) passes through 2 and so we set \( p[1][4] = p[2][4] = 2 \).
- \( d[3][2] + d[2][4] = 0 + -2 = -2 < 7 = d[3][4] \). Thus the shortest path from 3 to 4 permitting intermediate vertices \( \{1, 2\} \) passes through 2 and so we set \( p[3][4] = p[2][4] = 2 \).

Thus the result is:
We then perform the $k = 3$ iteration. Note that:

- $d[2][3] + d[3][1] = 1 + -3 = -2 < \infty = d[2][1]$. Thus the shortest path from 2 to 1 permitting intermediate vertices \{1, 2, 3\} passes through 3 and so we set $p[2][1] = p[3][1] = 3$.

Thus the result is:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>None</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>4</td>
</tr>
</tbody>
</table>

We then perform the $k = 4$ iteration. There are no updates and so the result is:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>4</td>
</tr>
</tbody>
</table>

Observe now the shortest path from 3 to 4. Since $p[3][4]$ is not null we assign $path = [4]$ and proceed with $u = 3$ and $v = 4$:

- We have $p[3][v] = p[3][4] = 2$ so the predecessor of 4 is 2 and so $path = [2, 4]$ and $v = 2$.

- We have $p[3][v] = p[3][2] = 1$ so the predecessor of 2 is 1 and so $path = [1, 2, 4]$ and $v = 1$.

- We have $p[3][v] = p[3][1] = 3$ so the predecessor of 1 is 3 and so $path = [3, 1, 2, 4]$ and $v = 3$.

- Since $v == u$ we stop.

Thus we have $path = [3, 1, 2, 4]$ which gives the shortest path $3 \rightarrow 1 \rightarrow 2 \rightarrow 4$. 