# BELLMAN-FORD ALGORITHM

The **Bellman-Ford algorithm** is a graph search algorithm that finds the shortest path between a given source vertex and all other vertices in the graph. This algorithm can be used on both weighted and unweighted graphs.

Like Dijkstra's shortest path algorithm , the Bellman-Ford algorithm is guaranteed to find the shortest path in a graph. Though it is slower than Dijkstra's algorithm, Bellman-Ford is capable of handling graphs that contain negative edge weights, so it is more versatile. It is worth noting that if there exists a negative cycle in the graph, then there is no shortest path. Going around the negative cycle an infinite number of times would continue to decrease the cost of the path (even though the path length is increasing). Because of this, Bellman-Ford can also detect negative cycles which is a useful feature.

The Bellman-Ford algorithm, like Dijkstra's algorithm, uses the principle of relaxation to find increasingly accurate path length. Bellman-Ford, though, tackles two main issues with this process.

1. If there are negative weight cycles, the search for a shortest path will go on forever.
2. Choosing a bad ordering for relaxations leads to exponential relaxations.

The detection of negative cycles is important, but the main contribution of this algorithm is in its ordering of relaxations. Dijkstra's algorithm is a greedy algorithm that selects the nearest vertex that has not been processed. Bellman-Ford, on the other hand, relaxes *all* of the edges.

## Algorithm Psuedo-code

The pseudo-code for the Bellman-Ford algorithm is quite short.

*This is high level description of Bellman-Ford written with pseudo-code, not an implementation*

```
for v in V:
    v.distance = infinity
    v.p = None
source.distance = 0
for i from 1 to |V| - 1:
    for (u, v) in E:
        relax(u, v)
```

The first `for` loop sets the distance to each vertex in the graph to infinity. This is later changed for the source vertex to equal zero. Also in that first `for` loop, the `p` value for each vertex is set to nothing. This value is a pointer to a predecessor vertex so that we can create a path later.

The next `for` loop simply goes through each edge `(u, v)` in `E` and relaxes it. This process is done `|V| - 1` times.

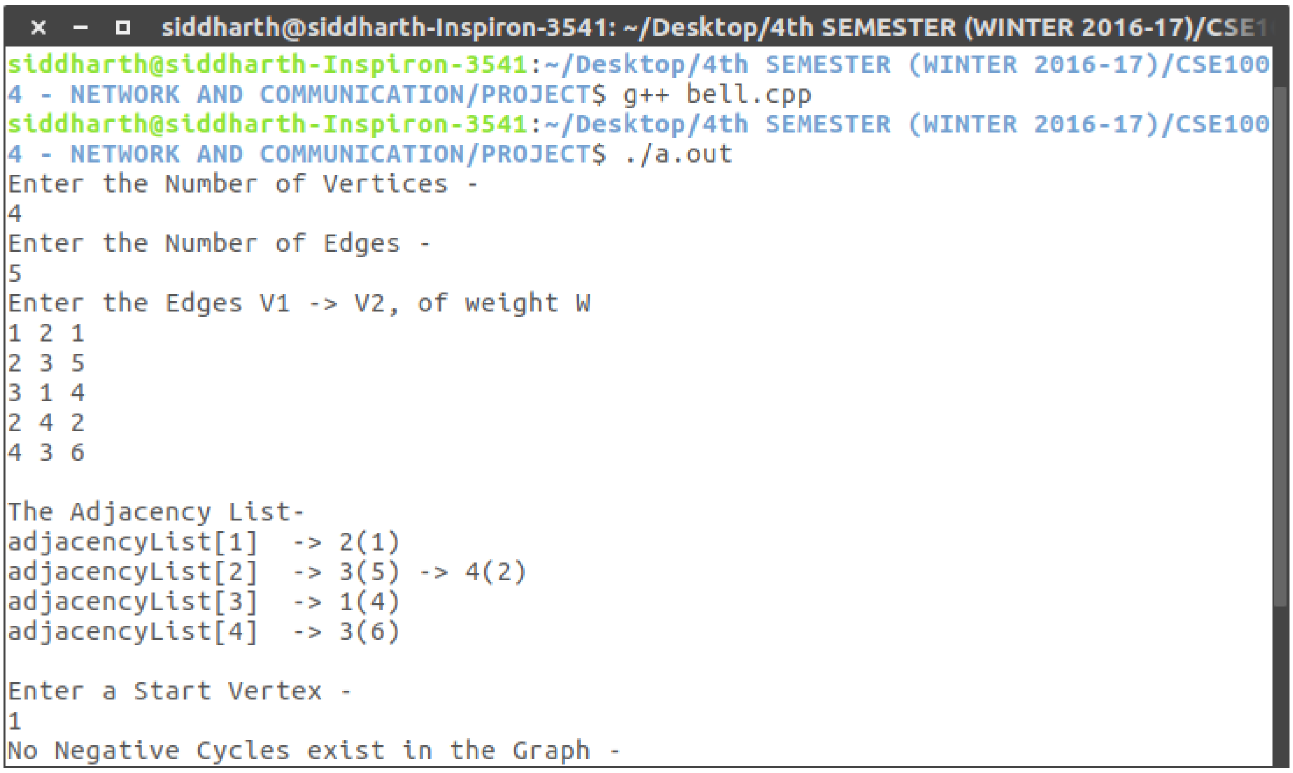
## Relax Equation

```
relax(u, v):
        if v.distance > u.distance + weight(u, v):
              v.distance = u.distance + weight(u, v)
              v.p = u
```

## Detecting Negative Cycle

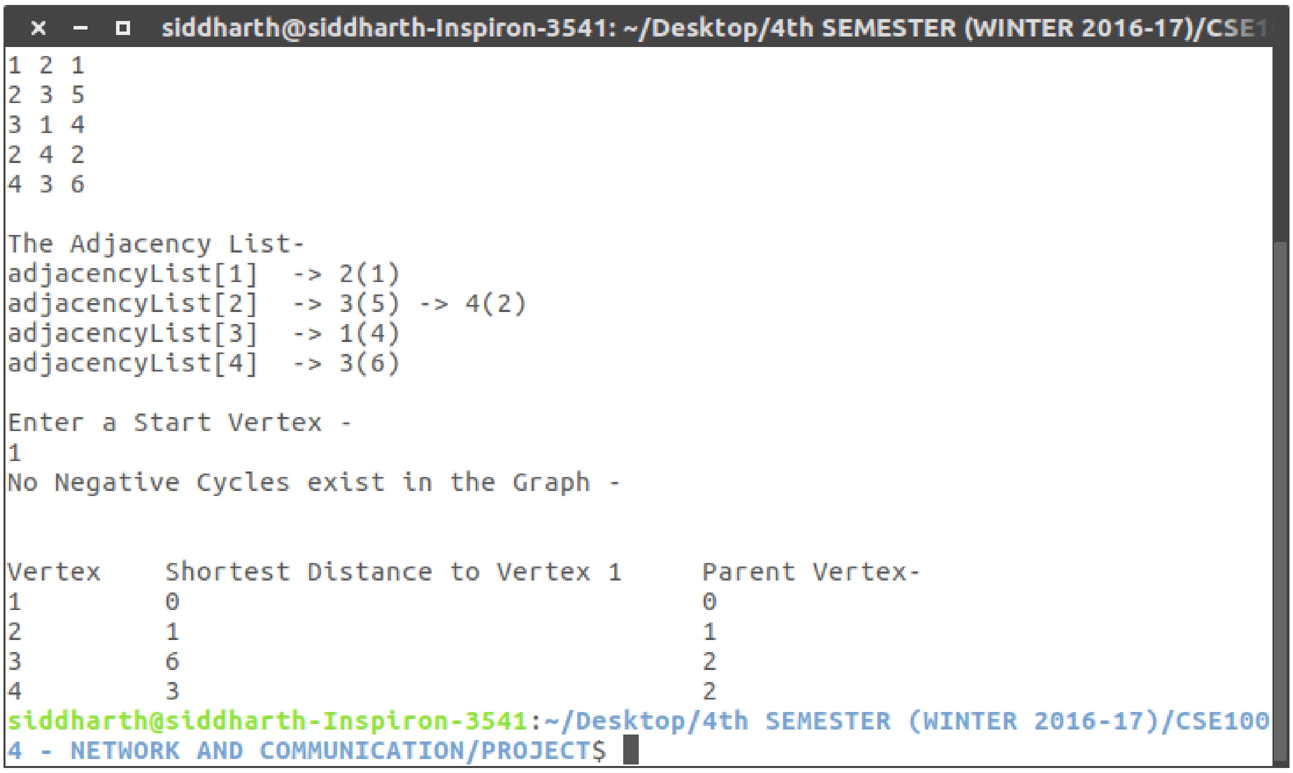
```
for v in V:
        v.distance = infinity
        v.p = None
  source.distance = 0
  for i from 1 to |V| - 1:
        for (u, v) in E:
              relax(u, v)
  for (u, v) in E:
        if v.distance > u.distance + weight(u, v):
              print "A negative weight cycle exists"
```

## Implementation in C++



siddharth@siddharth-Inspiron-3541: ~/Desktop/4th SEMESTER (WINTER 2016-17)/CSE1

```
siddharth@siddharth-Inspiron-3541:~/Desktop/4th SEMESTER (WINTER 2016-17)/CSE100
4 - NETWORK AND COMMUNICATION/PROJECT$ g++ bell.cpp
siddharth@siddharth-Inspiron-3541:~/Desktop/4th SEMESTER (WINTER 2016-17)/CSE100
4 - NETWORK AND COMMUNICATION/PROJECT$ ./a.out
Enter the Number of Vertices -
4
Enter the Number of Edges -
5
Enter the Edges V1 -> V2, of weight W
1 2 1
2 3 5
3 1 4
2 4 2
4 3 6

The Adjacency List-
adjacencyList[1]  -> 2(1)
adjacencyList[2]  -> 3(5) -> 4(2)
adjacencyList[3]  -> 1(4)
adjacencyList[4]  -> 3(6)

Enter a Start Vertex -
1
No Negative Cycles exist in the Graph -
```

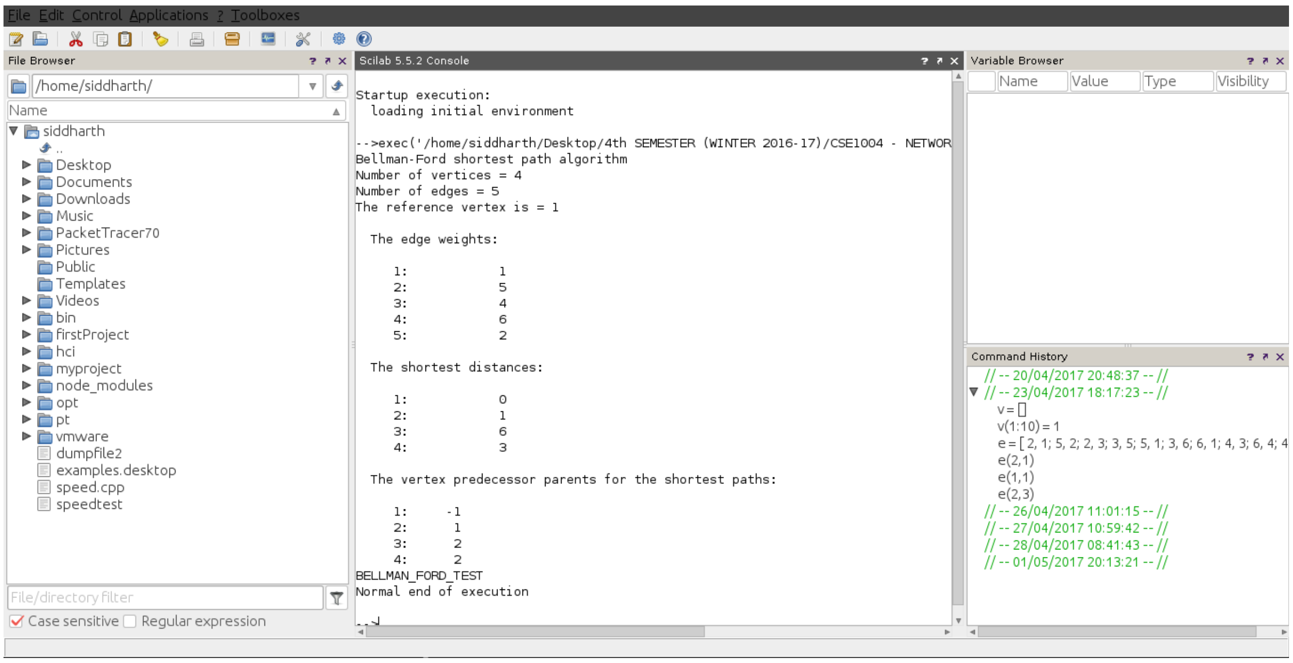


siddharth@siddharth-Inspiron-3541: ~/Desktop/4th SEMESTER (WINTER 2016-17)/CSE1

```
1 2 1
2 3 5
3 1 4
2 4 2
4 3 6

The Adjacency List-
adjacencyList[1]  -> 2(1)
adjacencyList[2]  -> 3(5) -> 4(2)
adjacencyList[3]  -> 1(4)
adjacencyList[4]  -> 3(6)

Enter a Start Vertex -
1
No Negative Cycles exist in the Graph -


Vertex    Shortest Distance to Vertex 1     Parent Vertex-
1         0                                 0
2         1                                 1
3         6                                 2
4         3                                 2
siddharth@siddharth-Inspiron-3541:~/Desktop/4th SEMESTER (WINTER 2016-17)/CSE100
4 - NETWORK AND COMMUNICATION/PROJECT$
```

## Implementation in Scilab

```
function [ v_weight, predecessor ] = bellman_ford ( v_num, e_num, source, e,
e_weight )

  r8_big = 1.0E+30;
  v_weight = [];
  predecessor = [];
  v_weight(1:v_num) = r8_big;
  v_weight(source) = 0;

  predecessor(1:v_num) = -1;

  for i = 1 : v_num
    for j = 1 : e_num
      u = e(j,2);
      v = e(j,1);
      t = v_weight(u) + e_weight(j);
      if ( t < v_weight(v) ) then
        v_weight(v) = t;
        predecessor(v) = u;
      end
    end
  end

  for j = 1 : e_num
    u = e(j,2);
    v = e(j,1);
    if ( v_weight(u) + e_weight(j) < v_weight(v) )
      mprintf ( '\n' );
      mprintf ( 'BELLMAN_FORD - Fatal error!\n' );
      mprintf ( '  Graph contains a cycle with negative weight.\n' );
      error ( 'BELLMAN_FORD - Fatal error!' );
    end
  end

  return
endfunction


function i4vec_print ( n, a, ti )

  mprintf ( '\n' );
  mprintf ( '%s\n', ti );
  mprintf ( '\n' );

  for i = 1 : n
    mprintf ( '%6d: %6d\n', i, a(i) );
  end

  return
endfunction


function r8vec_print ( n, a, ti )

  mprintf ('\n' );
  mprintf ( '%s\n', ti );
  mprintf ( '\n' );
  for i = 1 : n
    mprintf ( '%6d: %12g\n', i, a(i) );
  end

  return
endfunction


function bellman_ford_test01 ( )

  e_num = 5;
  v_num = 4;

  e = [ 2, 1; 3, 2; 1, 3; 3, 4; 4, 2];
  e_weight = [ 1;5;4;6;2];

  source = 1;

  mprintf ( 'Bellman-Ford shortest path algorithm\n' );

  mprintf( 'Number of vertices = %d\n', v_num);
  mprintf ( 'Number of edges = %d\n', e_num);
  mprintf( 'The reference vertex is = %d\n', source);

  r8vec_print ( e_num, e_weight, '  The edge weights:' );

  [ v_weight, predecessor ] = bellman_ford ( v_num, e_num, source, e,e_weight );

  r8vec_print ( v_num, v_weight, '  The shortest distances:' );
  i4vec_print ( v_num, predecessor, '  The vertex predecessor parents for the shortest
paths:' );

  return
endfunction

function bellman_ford_test ( )

  bellman_ford_test01;
  mprintf ( 'BELLMAN_FORD_TEST\n' );
  mprintf ( 'Normal end of execution\n' );


  return
endfunction
bellman_ford_test;
```



```
Startup execution:
  loading initial environment

-->exec('/home/siddharth/Desktop/4th SEMESTER (WINTER 2016-17)/CSE1004 - NETWOR
Bellman-Ford shortest path algorithm
Number of vertices = 4
Number of edges = 5
The reference vertex is = 1

  The edge weights:

     1:            1
     2:            5
     3:            4
     4:            6
     5:            2

  The shortest distances:

     1:            0
     2:            1
     3:            6
     4:            3

  The vertex predecessor parents for the shortest paths:

     1:      -1
     2:       1
     3:       2
     4:       2
BELLMAN_FORD_TEST
Normal end of execution

-->
```

```
// -- 20/04/2017 20:48:37 -- //
// -- 23/04/2017 18:17:23 -- //
    v = []
    v(1:10) = 1
    e = [ 2, 1; 5, 2; 2, 3; 3, 5; 5, 1; 3, 6; 6, 1; 4, 3; 6, 4; 4
    e(2,1)
    e(1,1)
    e(2,3)
// -- 26/04/2017 11:01:15 -- //
// -- 27/04/2017 10:59:42 -- //
// -- 28/04/2017 08:41:43 -- //
// -- 01/05/2017 20:13:21 -- //
```