Bellman Ford Algorithm

Dynamic Programming

Group members:

Tao Liu

Shuaishuai Li

Jingyao Li

Varun Yadav Keshaboina

Vishnu Vardhan Reddy

Table of Contents

1 Introduction

1.1 What is Bellman-Ford algorithm?

The Bellman-Ford algorithm is a type of dynamic programming algorithm. It can be used to find a shortest path from other vertexes to a single vertex in directed graphs where there are negative weights but no negative cycles. Negative cycles mean cycles that when adding the weights of edges together, the result is negative. In addition, it can also be used to detect negative cycles.

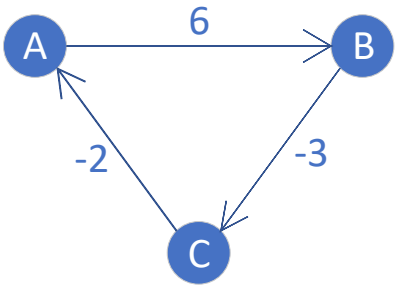


Figure 2: non-negative cycle

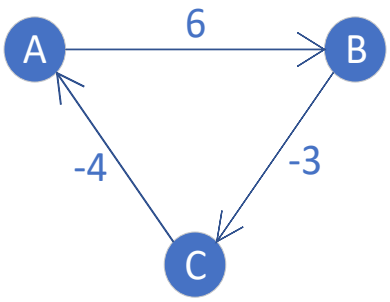


Figure 1: negative cycle

1.2 Why Bellman-Ford instead of Dijkstra's algorithm?

Usually, we would use a greedy algorithm, the Dijkstra's algorithm to find the shortest paths in a weighted graph. But the Dijkstra's algorithm can only be used in non-negative weight graphs. Because this algorithm assumes that when we select the vertex V with shortest path P from unvisited vertexes, we treat P as the shortest path for V. However, if there are negative weights, those Vertexes with longer path may reach V by adding a negative weight. Thus forming a shorter path than P for V. The following is an example:

Unvisited	Distance from s
v	1
u	2
w	∞

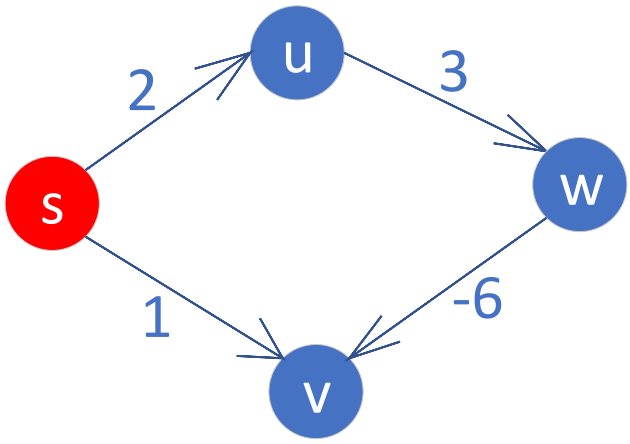


Figure 3: A graph that Dijkstra's does not apply

From figure 3 which uses Dijkstra's algorithm, s is visited while u, w, v are not. Next step we will set v as visited and treat "1" as the shortest distance from s to v. But actually, the path "s-u-w-v" is shorter than "s-v".

1.3 Why negative cycle matters?

Because if there is a negative cycle along the path, then every time we loop around the cycle, the total cost will decrease. Therefore we will keep looping and can never find the shortest path. If there is no negative cycle along the path P, we can confirm that P has no cycles thus all vertexes in P will only repeat once. Why? Because if P has cycles then the cycles must be non-negative and will increase the cost. So P will not go through these cycles. See this example:

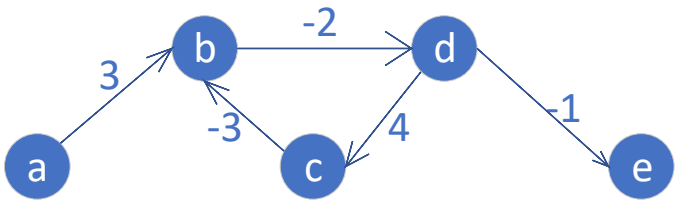


Figure 4: The path from a -> e goes into infinite loop

2 Finding Shortest Path

2.1 The crucial rule

To find shortest path using Bellman-Ford method, we must notice the rule: If a graph G has no negative cycles, then the shortest path P from any two vertexes will have no cycle. So it is clear that P has at most n vertexes and n-1 edges (n is the total number of vertexes in G) (Kleinberg & Tardos, 2006, pp. 292-293).

2.2 Breaking into subproblems

In the algorithm, we want to find a shortest path from all the vertexes v to a certain vertex t in G(V, E). From the rule above, we know the boundary is n-1. Now we try to break the problem into subproblems. We use MinCost(i, v) to denote the cost of the shortest path P from v to t when the path has at most i edges ($0 \leq i \leq n-1$, $v \in V$) (Kleinberg & Tardos, 2006, p. 293). Then MinCost(i, v) can be broken into the following subproblems:

①If P has at most i-1 edges, then MinCost(i, v) = MinCost(i - 1, v);

②If P has i edges, and the outgoing edge from v is (v, w). Then

MinCost(i, v) = weight(v, w) + MinCost(i - 1, w).

③Then choose the smaller one.

MinCost(i, v) = min(MinCost(i - 1, v), $min_{w \in V}$(weight(v, w) + MinCost(i - 1, w))).

2.3 Pseudocode of the algorithm

Now we can write the algorithm based on the subproblems above. We use M[v, i] to represent MinCost(i, v). The function Shortest-Path () aims to find a shortest path from s to t in graph G (Kleinberg & Tardos, 2006, p. 294).

```
1.  Shortest-Path(G, s, t)
2.     V = all vertexes in G
3.     n = V.length
4.     array M[V, from 0 to n-1]
5.     set M[t, 0] = 0
6.     set M[v, 0] = ∞ for other vertexes v except t
7.     for i from 1 to n-1
8.        for v in V
9.           M[v, i] = M[v, i - 1]
10.          for w in outgoing edges of v
11.             temp = weight(v, w) + M[w, i - 1]
12.             if temp < M[v, i]
13.                M[v, i] = temp
14.          endfor
15.       endfor
16.    endfor
17.    return M[s, n - 1]
```

2.4 An example

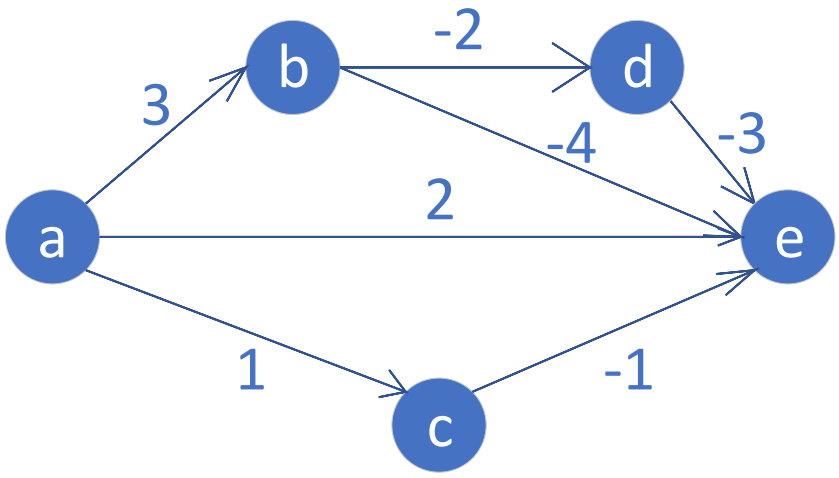


Figure 5: The input graph G

Now let's try to find the shortest path from a to e which means to calculate Shortest-Path(G, a, e). The result will be like this:

i / v	0	1	2	3	4
e	0	0	0	0	0
a	∞	2	-1	-2	-2
b	∞	-4	-5	-5	-5
c	∞	-1	-1	-1	-1
d	∞	-3	-3	-3	-3

Table 1: Array M after n-1 iterations

Pay attention to the way we get M [a, 2] = -1. There are 3 outgoing edges e, c and b from a. So we calculate the following values:

Cost from e = weight (a, e) + M [e, 2 - 1] = 2 + 0 = 2

Cost from c = weight (a, c) + M [c, 2 - 1] = 1 + -1 = 0

Cost from b = weight (a, b) + M [b, 2 - 1] = 3 + -4 = -1

After comparing the above 3 results with M [a, 1]. We set M [a, 2] to the smallest value -1.

2.5 An improvement to the algorithm

The algorithm above has drawbacks. Firstly, it takes space, actually we only interested in M [v, n – 1] instead of other columns. Secondly, it takes O(i*n) time to trace back the shortest path P from array M. For example, according to the array M in **Table 1**. To find the shortest path P (a-b-d-e) from a to e, we must compare "weight(a, b) + M[b, 3]", "weight(a, c) + M[c, 3]" with M[a, 3] to find the next vertex that leads to M[c, 4] = -2. When we find the next vertex b, we have to find the next vertex of b. So on and so forth until we reach e. To improve the algorithm, we will keep 2 arrays. One is M[v] to record the shortest costs from v to t, and another array is First [v] to record the next vertex of v along a shortest path. The improved algorithm will be as follow (Kleinberg & Tardos, 2006, pp. 294-296).

```
1.  Shortest-Path(G, s, t)
2.     V = all vertexes in G
3.     n = V.length
4.     array M[V]
5.     array First[V]
6.     set M[t] = 0
7.     set M[v] = ∞ for other vertexes v except t
8.     for i from 1 to n-1
9.        for v in V
10.          for w in outgoing edges of v
11.             temp = weight(v, w) + M[w]
12.             if temp < M[v]
13.                M[v] = temp
14.                First[v] = w
15.          endfor
16.       endfor
17.    endfor
18.    return M[s]
```

After applying this algorithm to the graph in **Figure 5**, the results are like this:

e	0
a	-2
b	-5
c	-1
d	-3

Table 2: Array M

e	-
a	b
b	d
c	e
d	e

Table 3: Array First

Now we can easily trace back the path. For example, the short path from a to e is a -> b (First[a])-> d (First[b]) -> e (First[d]).

3 Dealing with Negative cycles

3.1 Crucial Observations for detecting negative cycles

From the book (Kleinberg & Tardos, 2006, p. 303), we can find some statements that can help us to solve the problem.

> Statement 1: There is no negative cycle with a path to t if and only if MinCost(n, v) = MinCost(n – 1, v) for all vertexes v.

Because if there is a vertex s whose shortest path P to t meets MinCost(n, s) ≠ MinCost(n – 1, s), that means P has n edges. Which also means P has repeated vertexes and thus has a cycle. This cycle must be negative otherwise P will just chop it off. ***If such a vertex s doesn't exist, that means all vertexes to t will not go through negative cycle***.

However, if a vertex v meets MinCost(n, v) = MinCost(n – 1, v), does that mean the path P from v to t will definitely not have negative cycles? The answer is no. Take the graph in **Figure 6** for example, MinCost(6, a) = MinCost(5, a) but the path from a to c has a negative cycle.

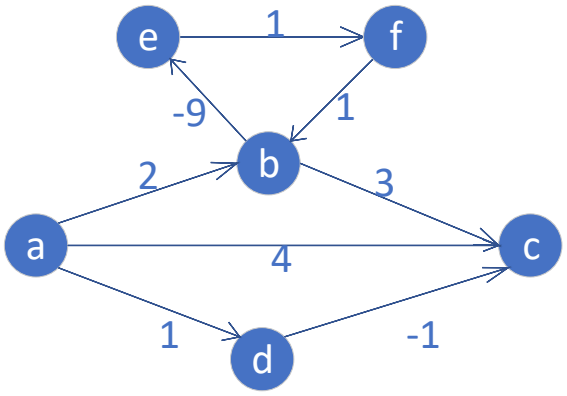


Figure 6: Graph with a negative cycle

3.2 find shortest paths when there are negative cycles

We can manage to find shortest paths for some vertexes even if there are negative cycles in graph G. Firstly, we use Statement 1 to find if there are cycles to t. If no cycles, we can declare that there are shortest paths for each vertex v with a cost of MinCost(n – 1, v). If there are vertexes w that meets MinCost(n, w) ≠ MinCost(n – 1, w), which means Path(w, t) has a negative cycle. We trace back Path (w, t), the Path (w, t) must contain repeated vertexes x, all the vertexes between the repeated x along the Path (w, t) are forming a negative cycle. When we find

all the vertexes W forming negative cycles, we can declare that for any v goes through some vertexes in W, we cannot find their shortest path using our algorithm. The pseudocode will be like this, in order to trace back quickly, we also use array First[v] to record the next vertex of v:

```
1.  Shortest-Path(G, s, t)
2.     V = all vertexes in G
3.     n = V.length
4.     array First[V]
5.     array M[V, from 0 to n]
6.     set M[t, 0] = 0
7.     set M[v, 0] = ∞ for other vertexes v except t
8.     // calculate shortest paths for each v when v has at most 1 to n edges
9.     for i from 1 to n
10.       for v in V
11.          M[v, i] = M[v, i - 1]
12.          for w in outgoing edges of v
13.             temp = weight(v, w) + M[w, i - 1]
14.             if temp < M[v, i]
15.                M[v, i] = temp
16.                First[v] = w
17.          end
18.       end
19.    end
20.    // W is a list of vertexes that forming a negative cycle
21.    list W = []
22.    // check if there are negative cycles to t, if so, find the cycles
23.    for v in V
24.        if M[v, n] != M[v, n - 1]
25.            if v in W
26.                Break
27.            else
28.                // Initially assume all vertexes all not on the path
29.                Visited[V] = false
30.                // trace back the path of v to t until find the cycle
31.                next = v
32.                // loop until find repeated vertex
33.                while next != t
34.                    if Visited[next] == false
35.                        Visited[next] = true
36.                    else
37.                        // add this repeated vertex and the other vertexes between it
38.                        W.add(next)
39.                        temp = First[next]
40.                        While temp != next
41.                            W.add(temp)
42.                            temp = First[temp]
43.                        break
44.                    next = First[next]
45.    end
46.    // check if the shortest path for s go through a negative cycle
47.    next = s
48.    while next != t
49.        if next in W
50.            return "Cannot find shortest path from s to t in our algorithm"
51.        next = First[next]
52.    return M[s, n - 1]
```

3.3 Time Complexity

This part is to analyze the time complexity of the algorithm above, suppose we have m edges in total.

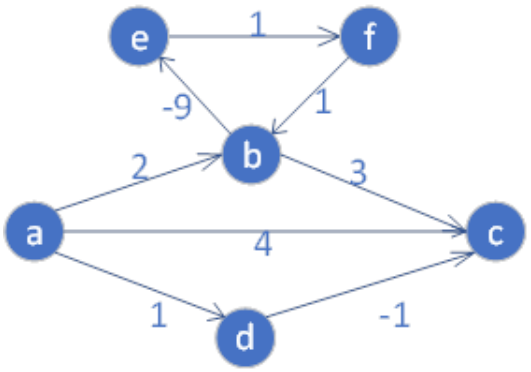
Firstly, the time for calculating M[v, n] is O(nm). Because in every iteration i, we need to calculate all outgoing edges for each v. Suppose v1 has k1 outgoing edges, v2 has k2 and so forth. The time in total will be k1 + k2 + …kn = m. Since there are n iterations, the time complexity is O(nm).

Secondly, the time for finding cycles is O(n + m). Because for each v, we must check its condition, that's O(n) in total. Then for the vertex v with a negative cycle, we will trace back at most n times to find the cycle (n is the length of Path(v, t)). If there are j negative cycles in total, and the length for j are k1, k2 to kj, the total time will be n * k1 +n * k2 + … + n * kj = n * (k1 + k2 + … + kj)<= n * m. So the time complexity is O(n + nm).

Thirdly, the time complexity is O(n*n). Because tracing back the path for s is O(n), and checking for each vertexes on the path is O(k)(k is the number of vertexes in negative cycles). Since k ≤ n, the worst time is O(n*n). Therefore the time complexity in total is O (n^2).

3.4 An example

This is the result after applying the algorithm in 3.2 to the graph in Figure 6 to find a shortest path from a to c.



	0	1	2	3	4	5	6
c	0	0	0	0	0	0	0
a	∞	4	0	0	0	-2	-2
b	∞	3	3	3	-4	-4	-4
d	∞	-1	-1	-1	-1	-1	-1
e	∞	∞	∞	5	5	5	-2
f	∞	∞	4	4	4	-3	-3

Table 3: Array M

	c	a	b	d	e	f
First(v)	-	b	e	c	f	b

Table 4: Array First

When we check M[v, 5] with M[v, 6], we see that M[e, 5] ≠ M[e, 6], so Path(e, c) has 6 edges and has negative cycles. Trace back Path(e, c), which is "e => f (First[e]) => b (First[f]) => e (First[b])". Therefore, e-f-b-e forms a negative cycle. Now we trace back Path(a, c), which is "a => b (First[a])". Since vertex b in the cycle, we can conclude that Path(a, c) has negative cycles.

Therefore, we cannot find a shortest path from a to c in our algorithm. We can find the shortest path from d to c, because Path(d, c) = "d=> c" and it does not go through negative cycles.

4 The Limitation of Bellman-Ford Algorithm

Bellman-Ford algorithm cannot find the shortest path for some vertexes in a graph that has at least one negative cycle. Here is a counter example:

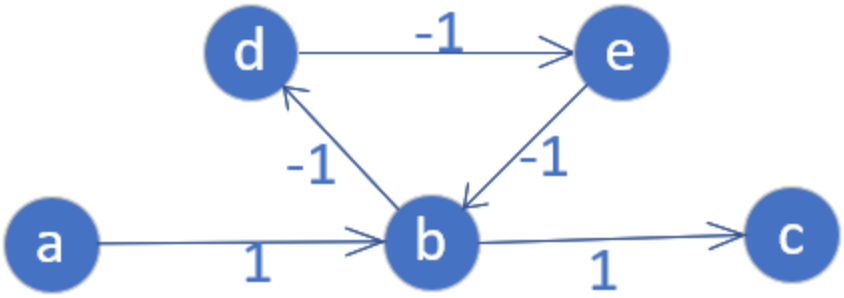


Figure 7: A graph with negative cycles

Proof. As we can see, the graph above contains a negative cycle (b, d, e). For the shortest simple path from b to c, M[b, 4] = -2, which equals the distance of walk (b, d, e, b, c), and the temporary path for b is "b=>d=>e=>b=>c", this violates the restriction of simple path: no duplicated vertexes. Q. E. D.

The single-source shortest simple paths problem (SSSPP) of graphs that have negative cycles is a NP-Complete problem because we can reduce ***Hamiltonian path*** problem to SSSPP.

Proof. For a graph that may have a Hamiltonian path, assign all edges with weight -1. Then we randomly choose one of its vertex v, and compute all single-source shortest simple paths which begin with v. If the input graph has a Hamiltonian path, v will in the path.

For all other nodes x, if the number of edges in the shortest simple path from v to x is n – 1 (n is the number of vertexes in the graph) then we find a Hamiltonian path from v to x.

The reduction above is linear time O(m) where m is the number of edges, hence we can reduce Hamiltonian path problem to SSSPP in polynomial time.

Given a graph G (V, E), a path from vertex s to t in G and a distance D. We can validate if the given path is valid in G and its distance is not larger than D in polynomial time. We just need to search in G's edge set and test if the distance sum of edges in the path is not longer than D. Both of them are polynomial. Hence SSSPP is NP-Complete. Q. E. D.

A naive algorithm for SSSPP is brutal search, which simply enumerate all possible combinations of edges in the input graph and replace the answer with the path if the path is shorter than current answer. The time complexity is $O(2^m)$ where m is the number of edges.

References

Kleinberg, J., & Tardos, E. (2006). *Algorithm Design.* Boston: Pearson Education, Inc.