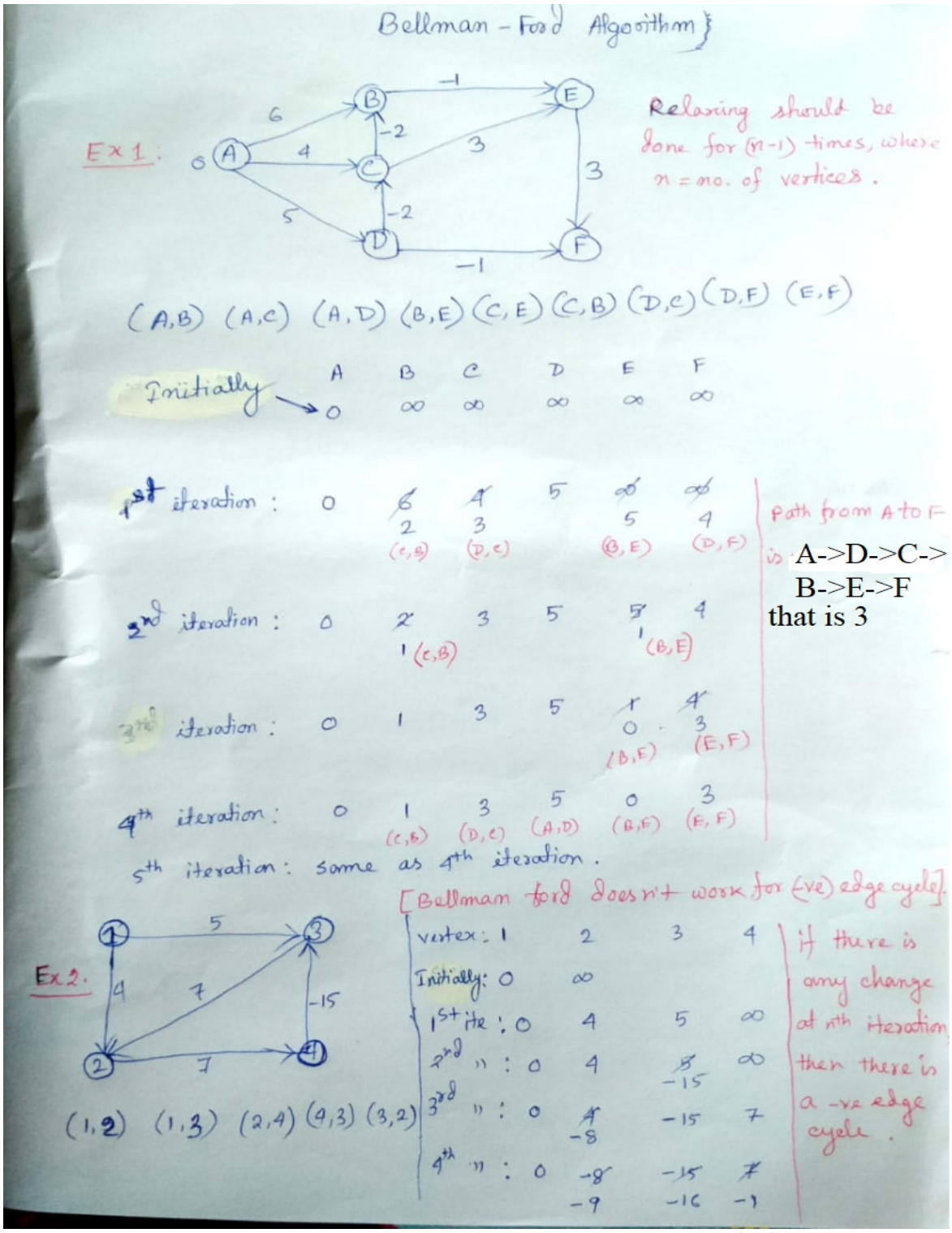
***Bellman-ford algorithm***
***(Single Source shortest path algorithm)***

The Bellman-Ford Algorithm is a dynamic programming algorithm for the single-sink (or singlesource) shortest path problem. It is slower than Dijkstra's algorithm, but can handle negativeweight directed edges, so long as there are no negative-weight cycles.



It starts with a starting vertex and calculates the distances of other vertices which can be reached by one edge. It then continues to find a path with two edges and so on.
The **Bellman-Ford algorithm** follows the bottom-up approach.

Bellman-Ford works better (better than Dijksra's) for distributed systems. Unlike Dijkstra's where we need to find the minimum value of all vertices, in Bellman-Ford, edges are considered one by one.

**Algorithm**
Following are the detailed steps.

Input: Graph and a source vertex src
Output: Shortest distance to all vertices from src. If there is a negative weight cycle, then shortest distances are not calculated, negative weight cycle is reported.

**1)** This step initializes distances from the source to all vertices as infinite and distance to the source itself as 0. Create an array dist[] of size |V| with all values as infinite except dist[src] where src is source vertex.

**2)** This step calculates shortest distances. Do following |V|-1 times where |V| is the number of vertices in given graph.
….. Do following for each edge u-v
……If dist[v] > dist[u] + weight of edge uv, then update dist[v]
…….dist[v] = dist[u] + weight of edge uv

**3)** This step reports if there is a negative weight cycle in graph. Do following for each edge u-v
……If dist[v] > dist[u] + weight of edge uv, then "Graph contains negative weight cycle"
The idea of step 3 is, step 2 guarantees the shortest distances if the graph doesn't contain a negative weight cycle. If we iterate through all edges one more time and get a shorter path for any vertex, then there is a negative weight cycle

The following algorithm takes as input a directed weighted graph and a starting vertex. It produces all the shortest paths from the starting vertex to all other vertices.

**Algorithm 1:** Bellman-Ford Algorithm

```
Data: Given a directed graph G(V, E) , the starting vertex S, and
      the weight W of each edge
Result: Shortest path from S to all other vertices in G
D[S] = 0;
R = V - S;
C = cardinality(V);
for each vertex k ∈ R do
    D[k] = ∞;
end
for each vertex i = 1 to (C - 1) do
    for each edge (e1, e2) ∈ E do
        Relax(e1, e2);
    end
end
for each edge (e1, e2) ∈ E do
    if D[e2] > D[e1] + W[e1, e2] then
        Print("Graph contains negative weight cycle");
    end
end
Procedure Relax (e1, e2)
for each edge (e1, e2) in E do
    if D[e2] > D[e1] + W[e1, e2] then
        D[e2] = D[e1] + W[e1, e2];
    end
end
```

After the initialization step, the algorithm started calculating the shortest distance from the starting vertex to all other vertices. This step runs $(|V| - 1)$ times. Within this step, the algorithm tries to explore different paths to reach other vertices and calculates the distances. If the algorithm finds any distance of a vertex that is smaller then the previously stored value then it relaxes the edge and stores the new value.

Finally, when the algorithm iterates $(|V| - 1)$ times and relaxes all the required edges, the algorithm gives a last check to find out if there is any negative cycle in the graph.

If there exists a negative cycle then the distances will keep decreasing. In such a case, the algorithm terminates and gives an output that the graph contains a negative cycle hence the algorithm can't compute the shortest path. If there is no negative cycle found, the algorithm returns the shortest distances.

**Why would one ever have edges with negative weights in real life?**

Negative weight edges might seem useless at first but they can explain a lot of phenomena like cashflow, the heat released/absorbed in a chemical reaction, etc.

For instance, if there are different ways to reach from one chemical A to another chemical B, each method will have sub-reactions involving both heat dissipation and absorption.

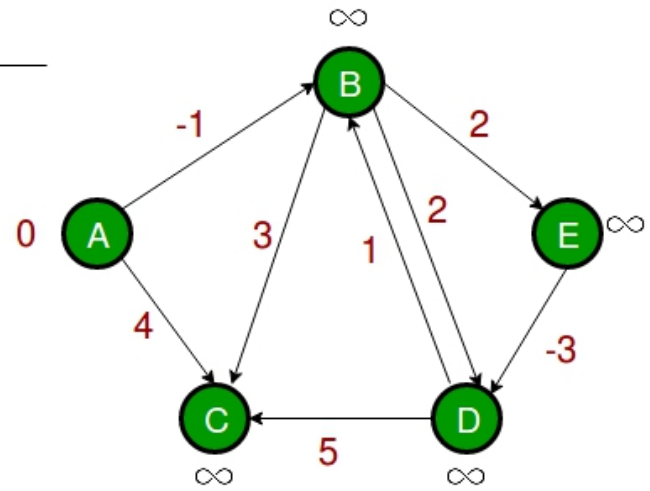
If we want to find the set of reactions where minimum energy is required, then we will need to be able to factor in the heat absorption as negative weights and heat dissipation as positive weights.

**Example**

Let us understand the algorithm with following example graph. The images are taken from this source.

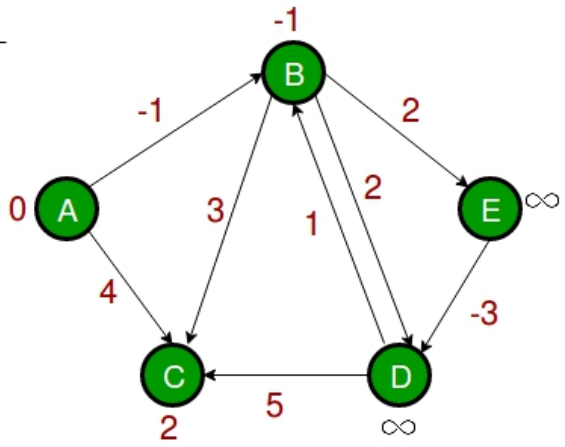
Let the given source vertex be 0. Initialize all distances as infinite, except the distance to the source itself. Total number of vertices in the graph is 5, so all edges must be processed 4 times.

| A | B | C | D | E |
|---|---|---|---|---|
| 0 | ∞ | ∞ | ∞ | ∞ |



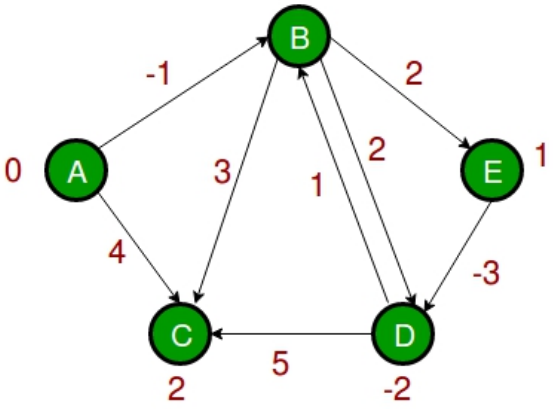

Let all edges are processed in the following order: (B, E), (D, B), (B, D), (A, B), (A, C), (D, C), (B, C), (E, D). We get the following distances when all edges are processed the first time. The first row shows initial distances. The second row shows distances when edges (B, E), (D, B), (B, D) and (A, B) are processed. The third row shows distances when (A, C) is processed. The fourth row shows when (D, C), (B, C) and (E, D) are processed.

| A | B | C | D | E |
|---|---|---|---|---|
| 0 | ∞ | ∞ | ∞ | ∞ |
| 0 | -1 | ∞ | ∞ | ∞ |
| 0 | -1 | 4 | ∞ | ∞ |
| 0 | -1 | 2 | ∞ | ∞ |




The first iteration guarantees to give all shortest paths which are at most 1 edge long. We get the following distances when all edges are processed second time (The last row shows final values).

| A | B | C | D | E |
|---|---|---|---|---|
| 0 | ∞ | ∞ | ∞ | ∞ |
| 0 | -1 | ∞ | ∞ | ∞ |
| 0 | -1 | 4 | ∞ | ∞ |
| 0 | -1 | 2 | ∞ | ∞ |
| 0 | -1 | 2 | ∞ | 1 |
| 0 | -1 | 2 | 1 | 1 |
| 0 | -1 | 2 | -2 | 1 |




The second iteration guarantees to give all shortest paths which are at most 2 edges long. The algorithm processes all edges 2 more times. The distances are minimized after the second iteration, so third and fourth iterations don't update the distances.

Excercise: