# Shortest Paths II: Bellman-Ford And Floyd-Warshall

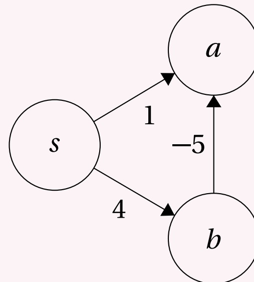
## 1 Shortest Paths in the Presence of Negative Weights

Dijkstra's algorithm allowed us to find single-source shortest paths in graphs with no negative weight edges. This is highly applicable for many real world problems since real-world graphs most often have non-negative weights. For example, we can run Dijkstra's on a graph of locations with edge weights corresponding to distances, to find shortest paths between locations.

However, we sometimes encounter graphs that have negative edge weights. In such scenarios, Dijkstra's may not give us the correct single-source shortest paths.

***Remark: Modifying the Graph to Remove Negative Edge Weights***

Consider the following graph:



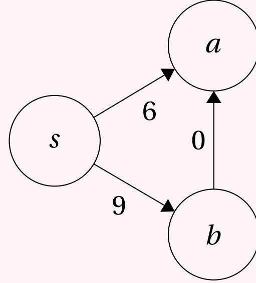

When we run Dijkstra's to find all single-source shortest paths from $s$, we get that the shortest path to vertex $a$ is of weight 1 ($s \to a$) while in reality, the shortest path should be of weight $-1$ ($s \to b \to a$).

One way that we might try to deal with negative weight edges is to add a constant $c$ to all the edge weights where

$$c \geq \max_{(u,v)\in E} |w(u,v)|$$

In this way, we guarantee that all edge weights are non-negative and so we hope that we can just run Dijkstra's on the new graph to find our single-source shortest paths.

Unfortunately, this does not necessarily give us the correct shortest paths. For example, in the above graph, if we add $c = 5$ to all edge weights, we obtain the following graph:




Running Dijkstra's on this modified graph to find the single-source shortest paths from $s$, we note that again, we get that the shortest path to vertex $a$ is $s \to a$ not $s \to b \to a$.

Why did this happen? We realize that adding a constant amount to every edge makes paths with more edges cost disproportionately more. Thus, if the shortest path to a vertex took more edges than other paths to that vertex, we could end up with the incorrect shortest path!

Fortunately, we are still able to find shortest paths on graphs with negative edges, using the **Bellman-Ford** algorithm. This algorithm is able to find single-source shortest paths on any graph that does not contain a **negative weight cycle**.

***Definition: Negative Weight Cycle***

A negative weight cycle is a cycle whose edge weights *sum up to* a negative value.

Note that this does not necessarily imply that all of the edges within the cycle have negative weights, only that their total sums to a negative value!

If there is such a cycle in a graph, we cannot find a shortest path, since we would be able to go around the cycle arbitrarily many times to create arbitrarily short paths, and thus the shortest path is undefined. Luckily, the Bellman-Ford algorithm will also allow us to identify when this is the case.

# 2 The Bellman-Ford Algorithm

## 2.1 The $k$-hop Distances Problem

We will develop the Bellman-Ford algorithm via a *Dynamic Programming* approach by reducing it to a related problem known as $k$-hop distances.

While Dijkstra's no longer works on graphs with negative weight edges, we can still utilize some of the key ideas from the algorithm when coming up with subproblems. We recall that to find the shortest path from the source vertex to a vertex $v$, Dijkstra's takes the minimum cost to go from the source vertex to some already visited vertex $u$, and then adds on the weight of the additional edge to go from $u$ to $v$. By taking the minimum over all possible choices of $u$, Dijkstra's is able to find the shortest path because the shortest path from the source to $v$ must be built from the shortest path between the source and some previously visited vertex $u$.

With negative weight edges, most of these statements still remain true. Particularly, the shortest path from the source to $v$ is *still* built from shorter shortest paths.

***Theorem: Subpaths Property***

The **subpaths property** states that any subpath of the shortest path between two vertices is itself a shortest path between its endpoints.

However, as we noted in the previous lecture, when negative weight edges are present, Dijkstra's priority order no longer guarantees that at the time that we first process vertex $v$ we have found this shorter shortest path!

Indeed, it seems difficult to come up with a processing order that will ensure that when we first process a vertex, we have already found the necessary subpath(s). Doing so feels like it would require knowing the shortest path to that vertex, which brings us back to our original problem. However, there is a relationship between the shortest path and the subpaths that we can utilize to our advantage. Specifically, the shortest path must have more *edges* than its subpaths.

Hence, if we find single-source shortest paths with at most 1 edge, these paths must be potential subpaths for the single-source shortest paths with at most 2 edges and so on. In general, the single-source shortest paths with at most $k$ edges, are the potential subpaths for the single-source shortest paths with at most $k+1$ edges.

With this in mind, we define the **$k$-hop shortest path** between two vertices as follows:

***Definition: $k$-hop Shortest Path***

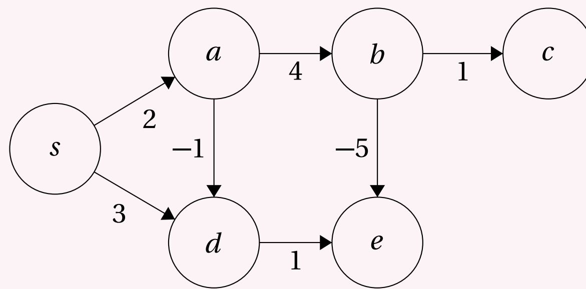
A **$k$-hop shortest path** between $u$ and $v$ in graph $G$ is a shortest path from $u$ to $v$ that contains at most $k$ edges. We will also refer to the weight of the $k$-hop shortest path as the **$k$-hop distance**.

***Remark: Other Use Cases for $k$-hop Shortest Paths***

$k$-hop shortest paths can be very useful in situations where the number of edges we can use is restricted. For example, say you're flying across the country, but you don't want to take more than two flights because it creates too many connections. To find the shortest valid path from your origin to your destination, you would want to find the 2-hop shortest path.

***Example: $k$-hop Shortest Paths***

Let's find the 2-hop shortest paths from $s$ to every vertex on the following graph:




The shortest path from $s$ to itself has weight 0, since starting and ending at $s$ is the only option. Likewise, there is only one path from $s$ to $a$, and it contains one edge. Thus, the 2-hop distance to $a$ is 2. Likewise, the 2-hop distance to $b$ is 6 – we take the path from $s$ through $a$ to $b$.

$d$ and $e$ are a little more complicated. There are 2 paths to $d$, and both of them use at most 2 hops. The path from $s$ to $d$ directly has cost 3, while the path through $a$ has cost 1. The 2-hop distance to $d$ is then 1, since this is the shorter of the two paths. There is one path to $e$ that uses 2 or fewer hops – it is to go from $s$ to $d$ and then to $e$. We cannot use $s, a, d, e$ or $s, a, b, e$ to find a 2-hop shortest path since each of those paths contains three edges. As a result, we get that the 2-hop distance to $e$ is 4.

Finally, there is the matter of $c$. There are no paths to $c$ that use 2 or fewer edges. We therefore say that the 2-hop distance from $s$ to $c$ is $\infty$.

How do $k$-hop distances help us solve the general shortest path problem? The following fact explains it.

***Claim: Maximum Number of Edges in Shortest Path***

In a graph $G$ with no negative weight cycles, given any vertices $u$ and $v$, there exists a shortest path from $u$ to $v$ that uses at most $n-1$ edges.

*Proof.* Assume for the sake of contradiction that the shortest path from $u$ to $v$ uses $n$ edges, and therefore passes through $n+1$ vertices (including $u$ and $v$). By the pigeonhole principle, some vertex $x$ must be visited twice along this path. This portion of the path between the first and second visit of $x$ forms a cycle.

Since by assumption, the graph has no negative weight cycles, this cycle has a nonnegative weight. Hence, we can remove it from the path, and the remaining part of the path will have equal or lesser weight than the original – meaning that the original is not a shortest path, which is a contradiction. □

## 2.2 A Dynamic Programming Formulation of $k$-hop Distances

Using $k$-hop shortest paths, we can now define our set of subproblems as

$$\texttt{Dist}(v, k) = \text{the } k\text{-hop distance from the source vertex to } v$$

for all $v \in V$ and $k \geq 0$.

Importantly, owing to the claim above, if the graph does not contain negative cycles, the answer to our original problem, the weight of the shortest path to each vertex $v$ can then be derived from the $k$-hop distances as the $(n-1)$-hop distances, i.e., as

$$\delta(s, v) = \texttt{Dist}(v, n-1) \qquad \text{for } v \text{ in } V.$$

Let's say we know the distances $\texttt{Dist}(v, k-1)$ for all $v$. How can we compute $\texttt{Dist}(v, k)$? The key insight here is that a path of $k$ vertices can be broken down into a path of $k-1$ vertices, followed by one more edge. This means that to find $\texttt{Dist}(v, k)$, we can consider all possible vertices $u$ such that $(u, v)$ is the final edge in the path. Of course, based of the subpaths property, we want to use the shortest $(k-1)$-hop path as well.

It follows that to find $\texttt{Dist}(v, k)$, we can try all possible choices of $u \in N^-(v)$. For each one, we consider the shortest $(k-1)$-hop path to $u$, and add on the weight of the final edge $(u, v)$. Remember that $N^-(v)$ means the set of "in-neighbors" of $v$, i.e., the vertices $u$ such that $(u, v) \in E$. We then get the recurrence:

$$\texttt{Dist}(v, k) = \begin{cases} 0 & \text{if } k = 0 \text{ and } v = s, \\ \infty & \text{if } k = 0 \text{ and } v \neq s, \\ \min\left(\texttt{Dist}(v, k-1), \min\limits_{u \in N^-(v)} \left(\texttt{Dist}(u, k-1) + w(u, v)\right)\right) & \text{otherwise.} \end{cases}$$

Directly implementing the above recurrence gives the following sequential algorithm. We will use a slightly different graph representation to make the algorithm efficient. Instead of a standard adjacency list, we will assume the graph is given a *reverse* adjacency list, i.e., it has all of the *in-edges* rather than out-edges of each vertex. This is necessary since that matches how our recurrence above is written.

***Algorithm: k-hop Shortest Paths***

```
// Assume a reverse adjacency list representation
type rev_adjacency_list = sequence<sequence<(int,real)>>

fun khopPaths(in_adj : rev_adjacency_list, source: int, K : int)
    -> sequence<real>:
  n = |adj|
  dist = [[∞ for _ in 0...K] for _ in 0...n-1]
  dist[source][0] ← 0

  for k in 1...K:
    for v in 0...n-1:
      dist[v][k] ← dist[v][k-1]
      for u, w in in_adj[v]:
        dist[v][k] ← min(dist[v][k], dist[u][k-1] + w)

  return [dist[v][K] for v in 0...n-1]
```

This algorithm computes the $k$-hop distances for any $k \geq 0$, and it turns out *two* values of $k$ are relevant to us for finding shortest paths.

We could just return the $(n-1)$-hop distances, call those the shortest paths and be done. This would be correct if we assume the algorithm is only ever used on graphs without negative cycles. But unfortunately, if it were called on a graph with negative cycles, it would run and just return a bogus answer.

As an improvement, ideally our shortest path algorithm returns an *option* with `NONE` representing finding a negative weight cycle and `SOME` $\left([\texttt{Dist}(v, n-1) \text{ for } v \text{ in } V]\right)$ returned otherwise. But how do we determine if there is a negative cycle in our graph?

***Claim: Identifying Negative Weight Cycles***

There exists a vertex $v$ such that the $n$-hop shortest path from $s$ to $v$ is shorter than the corresponding $(n-1)$-hop shortest path from $s$ to $v$ if and only if there exists a negative weight cycle reachable from $s$.

*Proof.* The forward direction follows easily from the previous claim. Taking the contrapositive of the previous claim, we get that if the shortest path from $u$ to $v$ uses more than $n-1$ edges, then the graph has a negative weight cycle. If the $n$-hop shortest path to $v$ is shorter than the $n-1$-hop shortest path, the shortest path to $v$ must use at least $n > n-1$ edges. Thus, the graph must have a negative weight cycle.

For the other direction, suppose there exists a negative-weight cycle reachable from $s$. We must prove that the $n$-hop distances are not equal to the $(n-1)$-hop distances. For this, note that at any point during the algorithm, since the $k$-hop distances are strictly a function of the $(k-1)$-hop distances for all $k$, if for any $k$, the $(k+1)$-hop distances are all equal to the $k$-hop distances, then so are the $(k+2)$-hop distances, and so on for all larger values of $k$.

Suppose there exists a negative-weight cycle of length $\ell$ (i.e., it contains $\ell$ edges) and consider any vertex $v$ in the cycle. Since the cycle has negative weight, then for any value of $k$, note that $(k+\ell)$-hop distance to $v$ must be smaller than the $k$-hop distance to $v$, since we can travel around the cycle and accumulate a negative amount of weight. Therefore the $(n+\ell)$-hop distances differ from the $n$-hop distances, and hence the $n$-hop distances differ from the $(n-1)$-hop distances by the above observation. □

Hence, our shortest path algorithm will find the $n$-hop distances and see whether any of them are shorter than the corresponding $n-1$-hop distances. If they are, it can return `NONE` to indicate that the input was invalid, else it can return the distances.

## 2.3 Sequential Bellman-Ford

Putting these ideas together, we arrive at the classic Bellman-Ford shortest path algorithm. A possible *sequential* implementation of Bellman-Ford is given below. It computes the $n$-hop distances and returns `NONE` if any of the distances decrease during hop $n$ since this indicates the presence of a negative cycle.

A few simplifications from the above dynamic programming solution are possible. First, we don't actually need the reverse adjacency list. Since it is more common to represent graphs as adjacency lists with outgoing edges, we note that instead of a vertex needing to iterate over its in-neighbours, each vertex can simply update its out-neighbor's distance estimate instead.

The Bellman-Ford algorithm also makes one additional optimization to save time and space. Unlike in the $k$-hop shortest paths algorithm, we do not store the results for all values of $k$, only the most recently calculated $k$-hop distances which are overwritten in place each iteration.

As a result, the Bellman-Ford algorithm actually no longer calculates the $k$-hop distances exactly. This is because earlier updates to vertices will now be used in the updates of later vertices within the same "hop". Hence, at any point in the algorithm, it knows either the $k$-hop distances or an even shorter path.

***Algorithm: Sequential Bellman-Ford***

```
// Assume a directed adjacency list representation
type adjacency_list = sequence<sequence<(int,real)>>

fun bellman_ford(adj : adjacency_list, source: int)
    -> option<sequence<real>>:
  n = |adj|
  dist = [∞ for _ in 0...n-1]
  dist[source] = 0

  for k in 1...n:
    for u in 0...n-1:
      for v, w in adj[u]:
        if dist[u] + w < dist[v]:
          if k == n: return NONE  // negative cycle detected!
          dist[v] ← dist[u] + w

return SOME(dist)
```

***Theorem: Cost of Sequential Bellman-Ford***

The cost of sequential Bellman-Ford is $O(nm)$.

*Proof.* The algorithm does $n$ iterations, each of which then considers each vertex and each of its neighbours. The cost is therefore

$$\sum_{k=1}^{n} \sum_{v \in V} d^+(v).$$

The sum of the outdegrees of every single vertex is just counting each edge, so this is just $nm$ and hence the cost of the algorithm is $O(nm)$. □

## 2.4 Parallel Bellman-Ford

What parts of Bellman-Ford can we perform in parallel? We realize that for a given $k$, the calculation of the $k$-hop distances are independent across the vertices and only dependent on the $(k-1)$-hop distances. As a result, we can parallelize calculating the $k$-hop distances for each vertex for a given $k$.

Additionally, we notice that the potential shortest distances we derive from taking the $(k-1)$-hop shortest distances to the neighbors are also independent, and we need to find the minimum of them, so this can be expressed using a parallel `map-reduce` pattern. This leads to the following *parallel* implementation of Bellman-Ford. Since it needs to reduce over the in-neighbours, this version once again requires a *reverse* adjacency list as the graph format to be efficient.

***Algorithm: Parallel Bellman-Ford***

```
// Assume a reverse adjacency list representation
type rev_adjacency_list = sequence<sequence<(int,real)>>

fun bellman_ford(in_adj : rev_adjacency_list, s : int)
    -> option<sequence<real>>:
  n = |in_adj|

  fn hop(dist : sequence<T>, k : int) =>
    map((fn (v : int) =>
      min(dist[v],
          reduce(min, ∞,
                 map(fn (u, w) => dist[u] + w, in_adj[v]))
          ), 0...n-1))

  init = parallel [0 if u == s else ∞ for u in 0...n-1]
  dist = fold_left(hop, init, 1...n-1)

  if hop(dist, n) != dist: return NONE  // Negative cycle!
  return SOME(dist)
```

The iteration over hops is purely sequential, but within it, each hop is computed from the previous one in parallel. This code is easy to write in a purely functional style since it can be expressed as a left fold over the hop function.

***Theorem: Cost of Parallel Bellman-Ford***

The span of parallel Bellman-Ford is $O(n \log n)$.

*Proof.* Since we must calculate the $k$-hop distances sequentially, we must at least make $O(n)$ iterations. Within each iteration, we calculate the $k$-hop distances for the vertices in parallel. For each vertex, we consider all of its neighbors (of which there are at most $O(n)$) and perform an $O(1)$ operation to calculate the distance of the path via each neighbor in parallel. This takes $O(1)$ span. We then take a minimum over the corresponding distances which we can perform

via a `reduce` giving us a $O(\log n)$ span for calculating each $k$-hop distance. Thus, our overall span is $O(n \log n)$. □

# 3 All-Pairs Shortest Paths: Floyd-Warshall

Lastly, we turn our attention to the **all-pairs** variant of the shortest paths problem. Given a weighted directed graph, we want to find the shortest path distance between *every* pair of vertices. The easiest way to solve this problem of course is to just use a single-source shortest paths algorithm and run it $n$ times, once from each start vertex.

> ***Theorem: All-pairs shortest paths via SSSP***
>
> All-pairs shortest paths can be solved in $O(nm \log n)$ work and $O(m \log n)$ span for graphs with non-negative weights, or in $O(mn^2)$ work and $O(n \log n)$ span for graphs with negative or non-negative weights.

*Proof.* Use Dijkstra's algorithm or the Bellman-Ford algorithm $n$ times in parallel. □

## 3.1 APSP via Dynamic Programming

Our algorithm for APSP will, like our last two algorithms, also be based on dynamic programming. All three of our previous algorithms for shortest paths were based on the idea of taking an existing path and extending it by one additional edge.

We could envision a similar approach for APSP, and might consider formulating a dynamic programming algorithm based on subproblems like:

$$\text{Dist}(u, v, \ell) := \begin{cases} \text{the weight of the shortest path between} \\ u \text{ and } v \text{ consisting of } \ell \text{ edges.} \end{cases}$$

This would work and result in a cost of $O(mn^2)$. However, it turns out this isn't a very novel approach, its actually just the same as Bellman-Ford again!

The nice thing about paths is that there are lots of ways of breaking them up into pieces. The previous strategy was to just add a single edge at a time, but that seems inefficient because every time we want to add an edge, we have to look at every possible second-last vertex.

Another way to break paths up is to chop them into two paths! A path from $u$ to $v$ can be broken at any point along the path $k$ into the two paths $u$ to $k$ and $k$ to $v$. How exactly does this let us decompose the problem into *smaller problems* though? We can not just define our subproblems as the shortest path between $u$ and $v$ and then solve them by trying all intermediate vertices $k$, because this would assume that we already had all the shortest paths between all $u$ and all possible $k$ to begin with, i.e., the problems aren't guaranteed to be smaller problems.

To make the problems smaller, we need one crucial observation: In a valid shortest path, there is no reason to use the same vertex twice! So, when we decide to break a shortest path $u$ to $v$ into two paths $U$ to $k$ and $k$ to $v$, we don't need $k$ to occur inside either of those paths! Let's therefore try adding new intermediate vertices one at a time.

**Define our subproblems** The idea is that we want to build paths out of increasingly larger sets of intermediate vertices. When vertex $k$ is introduced, we can stitch together a path from $u$ to $k$ and $k$ to $v$ to build a path from $u$ to $v$. Those smaller paths do not need to contain $k$ since there is no point in using $k$ more than once. So, we will use the subproblems

$$\text{Dist}(u, v, k) := \begin{cases} \text{weight of the shortest path from } u \text{ to } v \\ \text{using intermediate vertices } \{1, 2, \ldots, k\} \end{cases}$$

**Deriving a recurrence** We need to consider two cases. For the pair $u, v$, either the shortest path using the intermediate vertices $\{1, 2, \ldots, k\}$ goes through $k$ or it does not. If it does not, then the answer is the same as it was before $k$ became an option. If $k$ now gets used, we can *break the path at* $k$ and use the optimal substructure to glue together the two shortest paths divided at $k$ to get a new shortest path. Writing the recurrence using this idea looks like this.

$$\text{Dist}(u, v, k) = \min\{\text{Dist}(u, v, k-1), \text{Dist}(u, k, k-1) + \text{Dist}(k, v, k-1)\}.$$

Our base case will just be

$$\text{Dist}(u, v, 0) = \begin{cases} 0 & \text{if } u = v, \\ w(u, v) & \text{if } (u, v) \in E, \\ \infty & \text{otherwise.} \end{cases}$$

**Analysis** We have $O(n^3)$ subproblems and each of them takes $O(1)$ time to evaluate, so this takes $O(n^3)$ time! Notice that the key improvement here over the previous algorithm was that for each subproblem, we only needed to make a single binary decision: use $k$ or don't use $k$, which is much more efficient than our earlier algorithm which had to try *every vertex*.

## 3.2 The Floyd-Warshall Algorithm

One downside of the above algorithm is that it uses a lot of space, $O(n^3)$, which is a factor $n$ larger than the input graph. This is bad if the graph is large. Can we reduce this? There are two ways. First, notice that the subproblems for parameter $k$ only depend on the subproblems with parameter $k-1$. So, we don't actually need to store all $O(n^3)$ subproblems, we can just keep the last set of subproblems and compute bottom-up in increasing order of $k$.

Here's an even simpler but more subtle way to optimize the algorithm. Similar to Bellman-Ford, we don't actually need to keep the intermediate results for all previous values of $k$. We can simply update in-place as we go, as this can only improve the distance estimates and will never make them worse. This is called the Floyd-Warshall algorithm.

Unlike all of our previous algorithms which work best with an adjacency list representation, the Floyd-Warshall algorithm works best with an *adjacency matrix*. This is because it isn't updating paths by extending them by a single edge anymore, so individual edges are less relevant. What it needs is the fastest way possible to look up the distance between any pair of vertices, which is what an adjacency matrix is optimized for.

*Algorithm: Floyd-Warshall*

```
// assume a directed adjacency matrix representation
type adj_matrix = sequence<sequence<real>>

fun floyd_warshall(D : adj_matrix):
  // After each iteration, D[u][v] = weight of the shortest u->v
  // path that's allowed to use vertices in the set 1..k
  for k in 0...n-1:
    for u in 0...n-1:
      for v in 0...n-1:
        D[u][v] ← min(D[u][v], D[u][k] + D[k][v]);
  return D
```

So what happened here, it looks like we just forgot the $k$ parameter of the DP, right? It turns out that this algorithm is still correct, but now it only uses $O(n^2)$ space because it just keeps a single 2D array of distance estimates. Why does this work? Well, compared to the by-the-book implementation, all this does is allow the possibility that $D[u][k]$ or $D[k][v]$ accounts for vertex $k$ already, but a shortest path won't use vertex $k$ twice, so this doesn't affect the answer!

## 3.3 Parallel Floyd-Warshall

Like Bellman-Ford, most steps of Floyd-Warshall are independent and can be parallelized. Fundamentally, the $n$ steps of $k$ are required to be sequential, but the computation of `D[u][v]` for each $u$ and $v$ can be ran independently in parallel using the values from the previous $k$.

*Algorithm: Parallel Floyd-Warshall*

```
fun floyd_warshall(D : adj_matrix):
  fn next_k(P : adj_matrix, k : int) =>
    tabulate(fn u => tabulate(fn v =>
      min(P[u][v], P[u][k] + P[k][v]),
    0...n-1), 0...n-1)
  return fold_left(next_k, D, 0...n-1)
```

We write this functionally by using a left fold over a function that takes the adjacency matrix and computes the next set of distances by considering $k$ as an intermediate vertex.

***Theorem: Cost of Parallel Floyd-Warshall***

Parallel Floyd-Warshall costs $O(n^3)$ work and $O(n)$ span.

*Proof.* Folding over the $n$ values of $k$ is sequential, but the inner calculation for each value of $u$ and $v$ costs $O(1)$ and is done in parallel, so the work is $O(n^3)$ and the span is $O(n)$. □

As an exercise, think about how to extend this algorithm to detect the presence of negative-weight cycles. Currently it just returns a matrix of distances, some of which may be incorrect because of the existence of negative-weight cycles.