COMS 4995: Parallel Functional Programming
Parallel Bellman-Ford Algorithm

Zhe Hua – zh2261

December 23, 2020

1 Introduction

Bellman-Ford algorithm solves the single-source shortest-path problem. While Bellman-Ford algorithm can solve graphs with negative edge weights, Dijkstra's algorithm cannot. Thus, one important use case of Bellman-Ford algorithm is negative cycle detection. Sequential Bellman-Ford runs in $O(nm)$ time where n is the number of vertices and m is the number of edges in the graph. For sparse graph, the algorithm's time complexity is close to that of Dijkstra's algorithm. However, when the graph is complete, its time complexity becomes $O(n^3)$. In this project, I parallelize Bellman-Ford algorithm for negative cycle detection task. I conduct broadly two levels of parallelization, implement 6 different versions of parallel Bellman-Ford algorithm, and test them on complete graphs with 50, 500, 1000, and 2000 vertices. The experiments demonstrate that parallelization can attain 1.11x, 5.45x, and 6.19x speedups for complete graphs with 50, 500, and 1000 vertices. In addition, the results show that while the version with one level of parallelization (V1) has the best performance for graphs with less than 1000 vertices, for graphs with more than 1000 vertices, the version with two levels of parallelization using vectors (V3-vector) can be even faster, attaining 1.1x the performance of V1.

2 Bellman-Ford Algorithm

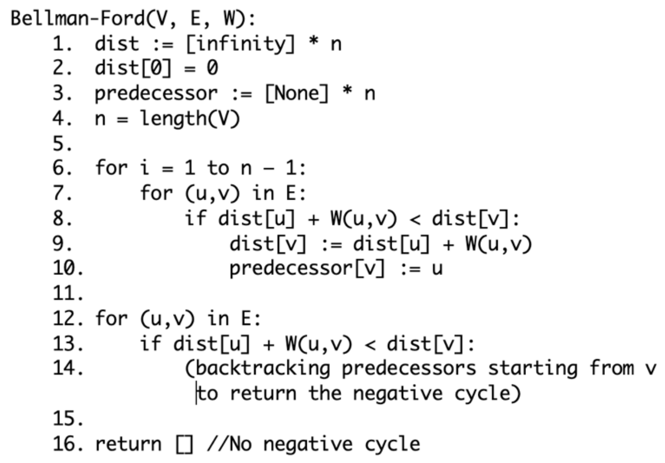
Bellman-Ford algorithm is a dynamic programming algorithm. It simplifies the single-source shortest-path problem by breaking it down into subproblems in a recursive manner. Let s to be the source vertex, v to be the target vertex, E to be the set of all edges, V to be the set of all vertices, and $W(u, v)$ to be the weight of the (u, v) edge. The subproblem is defined as equation (Eq. 1).

$$OPT(i, v) := cost\ of\ shortest\ s \rightsquigarrow v\ path\ using\ at\ most\ i\ edges \qquad \text{(Eq. 1)}$$

The overall problem is then defined as equation (Eq. 2).

$$OPT(i, v) := \begin{cases} 0, & \text{if } i = 0, v = s \\ \infty, & \text{if } i = 0, v \neq s \\ \min\left\{OPT(i-1, v), \min\limits_{u:(u,v)\in E}[OPT(i-1, u) + W(u, v)]\right\}, & \text{if } i > 0 \end{cases} \qquad \text{(Eq. 2)}$$

To find the optimal solution to the overall problem, the algorithm finds the optimal solutions to $n - 1$ subproblems incrementally from $OPT(1, v)$ to $OPT(n - 1, v)$ where $n := |V|$. Each of these $n - 1$ steps to solve the subproblems is commonly called relaxation. To detect negative cycle, the algorithm performs an additional relaxation, since the longest path without a cycle can have at most $|V| - 1$ edges. If a shorter distance is found between the source vertex and any of the target vertex in this additional relaxation, then a negative cycle exists in the graph. A pseudocode to find a negative cycle using Bellman-Ford algorithm is presented below.



```
Bellman-Ford(V, E, W):
    dist := [infinity] * n
    dist[0] = 0
    predecessor := [None] * n
    n = length(V)

    for i = 1 to n - 1:
        for (u,v) in E:
            if dist[u] + W(u,v) < dist[v]:
                dist[v] := dist[u] + W(u,v)
                predecessor[v] := u

    for (u,v) in E:
        if dist[u] + W(u,v) < dist[v]:
            (backtracking predecessors starting from v
             to return the negative cycle)

    return [] //No negative cycle
```

3 Implementation

3.1 Graph Representation

I use a nested IntMap to represent the complete graph. On the first level, the key type is Int, denoting a vertex u, while the value type is an IntMap. On the second level, the key type is Int, denoting a vertex v, while the value type is Int, denoting the weight $W(u, v)$.

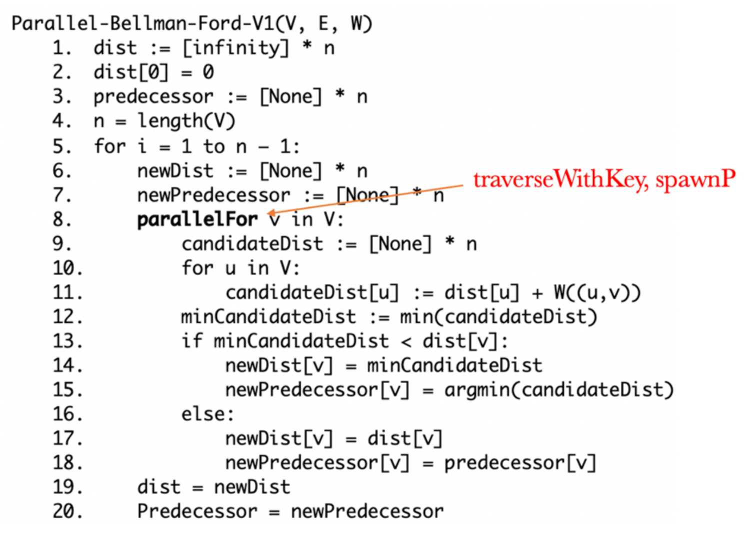
3.2 Random Graph Generation

I build the program generateGraph.hs to write to a text file a complete graph of size s in which the largest negative cycle has length l, where s and l are the inputs to the program. In the output text file for the graph, each line i in the text file is a sequence of comma-separated integers where the j-th integer in the line is the weight of the edge (i, j). The program first builds a complete graph with negative weights but without negative cycle. Then, it randomly generates a list $[c_1, c_2, \dots, c_l]$ of length l denoting the cycle and for all pairs (c_i, c_{i+1}) randomly adds weight w' to the original weight of edge (c_i, c_{i+1}) and subtracts w' from the original weight of edge (c_{i+1}, c_i).

3.3 Parallel Implementation

I build 6 different versions of parallel Bellman-Ford algorithm to find a negative cycle in a complete graph. The first version (V1) uses one level of parallelization, while the other 5 versions (V2, V2-list, V2-vector, V3-list, V3-vector) use two levels of parallelization.

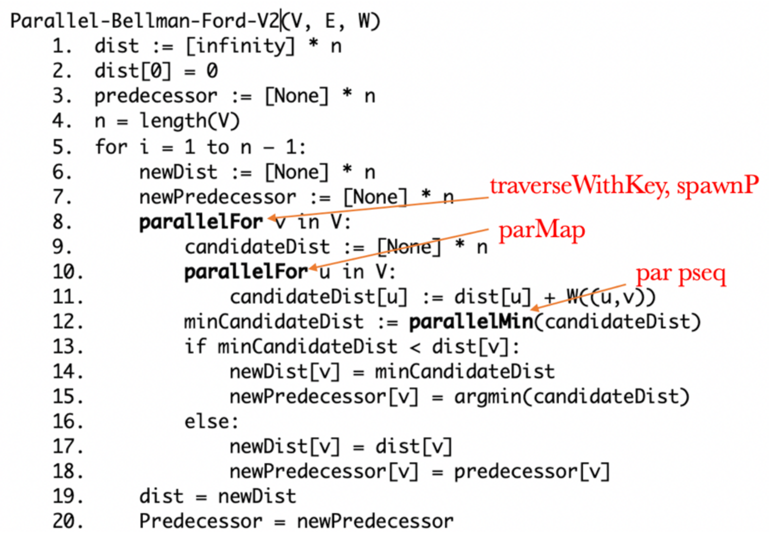
3.3.1 V1



```
Parallel-Bellman-Ford-V1(V, E, W)
    1.  dist := [infinity] * n
    2.  dist[0] = 0
    3.  predecessor := [None] * n
    4.  n = length(V)
    5.  for i = 1 to n - 1:
    6.      newDist := [None] * n
    7.      newPredecessor := [None] * n
    8.      **parallelFor** v in V:
    9.          candidateDist := [None] * n
    10.         for u in V:
    11.             candidateDist[u] := dist[u] + W((u,v))
    12.         minCandidateDist := min(candidateDist)
    13.         if minCandidateDist < dist[v]:
    14.             newDist[v] = minCandidateDist
    15.             newPredecessor[v] = argmin(candidateDist)
    16.         else:
    17.             newDist[v] = dist[v]
    18.             newPredecessor[v] = predecessor[v]
    19.     dist = newDist
    20.     Predecessor = newPredecessor
```

V1 uses only one level of parallelization. It parallelizes the algorithm by splitting the iteration through all edges in the graph into $n := |V|$ chunks. In each chunk, all edges have the same target vertex.

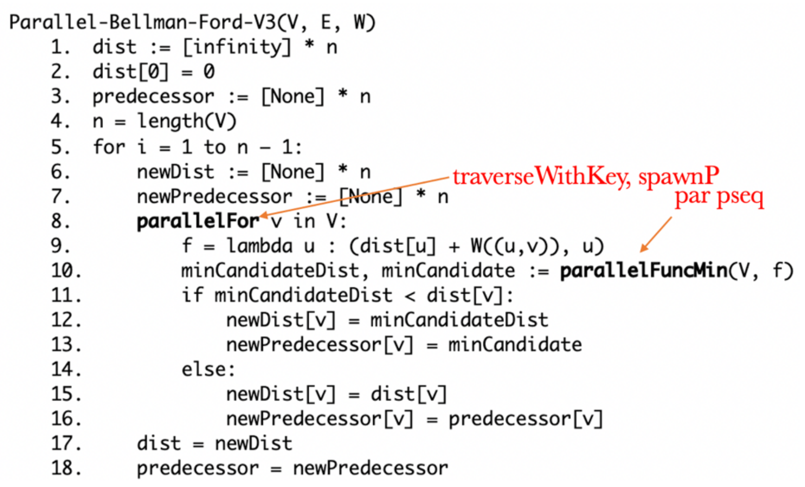
3.3.2 V2, V2-list, V2-vector



```
Parallel-Bellman-Ford-V2(V, E, W)
    1.  dist := [infinity] * n
    2.  dist[0] = 0
    3.  predecessor := [None] * n
    4.  n = length(V)
    5.  for i = 1 to n - 1:
    6.      newDist := [None] * n
    7.      newPredecessor := [None] * n
    8.      **parallelFor** v in V:
    9.          candidateDist := [None] * n
    10.         **parallelFor** u in V:
    11.             candidateDist[u] := dist[u] + W((u,v))
    12.         minCandidateDist := **parallelMin**(candidateDist)
    13.         if minCandidateDist < dist[v]:
    14.             newDist[v] = minCandidateDist
    15.             newPredecessor[v] = argmin(candidateDist)
    16.         else:
    17.             newDist[v] = dist[v]
    18.             newPredecessor[v] = predecessor[v]
    19.     dist = newDist
    20.     Predecessor = newPredecessor
```

The second, third, and fourth versions (V2, V2-list, V2-vector) use two levels of parallelization. The first-level parallelization is the same as the one in V1. The algorithm is parallelized at an additional level by using parMap to iterate all the edges in a chunk. V2 does not parallelize the minimum-finding step on line 12; it finds the minimum sequentially. V2-list finds the minimum by diving the list by half at each level of depth up to a depth of d where d is an input to the program. Similar to V2-list, V2-vector finds the minimum by dividing the list at each level of depth; however, unlike V2-list, it uses the data structure vector which is supposedly faster because splitAt and length operations in Haskell cost $O(1)$ time for vector but $O(n)$ time for list.

3.3.3 V3-list, V3-vector



```
Parallel-Bellman-Ford-V3(V, E, W)
    1.  dist := [infinity] * n
    2.  dist[0] = 0
    3.  predecessor := [None] * n
    4.  n = length(V)
    5.  for i = 1 to n - 1:
    6.      newDist := [None] * n
    7.      newPredecessor := [None] * n
    8.      parallelFor v in V:
    9.          f = lambda u : (dist[u] + W((u,v)), u)
    10.         minCandidateDist, minCandidate := parallelFuncMin(V, f)
    11.         if minCandidateDist < dist[v]:
    12.             newDist[v] = minCandidateDist
    13.             newPredecessor[v] = minCandidate
    14.         else:
    15.             newDist[v] = dist[v]
    16.             newPredecessor[v] = predecessor[v]
    17.     dist = newDist
    18.     predecessor = newPredecessor
```

The fifth and sixth versions (V3-list, V3-vector) use two levels of parallelization. The first-level parallelization is the same as the one in V1. The algorithm is parallelized at the second level by parallelizing the minimum-finding step in the algorithm. While V2-list an V2-vector first iterate through the list to calculate all the new values and then find the minimum of the new values, V3-list and V3-vector directly start the minimum-finding step and calculate the new values while finding the minimum. This is likely to be an improvement over V2-list and V2-vector, because it omits an $O(n)$ iteration (although parallelizable) over all the vertices to calculate the new values. V3-vector uses vector, while V3-list uses list.

4 Results

I perform the experiments on a MacBook Pro with 2.3 GHz 8-Core Intel Core i9 processor and 16 GB 2667 MHz DDR4 memory. The sequential version is the V1 algorithm using 1 core.

4.1 50 vertices

	V1	V2	V2-List	V2-Vector	V3-List	V3-Vector
2 cores	**0.0248s**	0.0258s	0.0327s	0.0293s	0.0272s	0.0255s
4 cores	0.0250s	0.0257s	0.0277s	0.0253s	0.0253s	0.0261s
8 cores	0.0264s	0.0280s	0.0276s	0.0276s	0.0270s	0.0257s

I test the six different versions of parallel Bellman-Ford on 50 complete graphs with 50 vertices and a negative cycle of length 5 randomly generated by generateGraph.hs. The time shown above is the average across 50 graphs. The sequential version takes 0.0275s. The best version V1 (2 cores) achieves a 1.11x speedup. The other versions are slow due to the overhead of creating a second layer of parallelization.

4.2 500 vertices

	V1	V2	V2-List	V2-Vector	V3-List	V3-Vector
2 cores	20.026s	23.080s	25.460s	26.149s	23.687s	21.967s
4 cores	11.096s	12.554s	13.742s	14.097s	12.830s	11.935s
8 cores	**6.326s**	7.746s	8.650s	9.026s	7.493s	7.063s

I test the six different versions of parallel Bellman-Ford on 20 complete graphs with 500 vertices and a negative cycle of length 50 randomly generated by generateGraph.hs. The time shown above is the average across 20 graphs. The sequential version takes 34.487s. The best version V1 (8 cores) achieves a 5.45x speedup. For graph of this size, the overhead of creating a second layer still outweighs the benefit.

4.3 1000 vertices

	V1	V2	V2-List	V2-Vector	V3-List	V3-Vector
2 cores	182.087s	201.616s	222.537s	230.242s	199.271s	183.265s
4 cores	95.238s	108.844s	117.880s	122.676s	107.271s	98.333s
8 cores	**54.976s**	64.715s	70.613s	81.488s	62.018s	56.704s

I test the six different versions of parallel Bellman-Ford on 10 complete graphs with 1000 vertices and a negative cycle of length 100 randomly generated by generateGraph.hs. The time shown above is the average across 10 graphs. The sequential version takes 340.125s. The best version V1 (8 cores) achieves a 6.19x speedup. Versions with second layer of parallelization are catching up with V1 which has only one layer of parallelization. The performance of V3-vector is on a par with that of V1.

4.4 2000 vertices

	V1	V2	V2-List	V2-Vector	V3-List	V3-Vector
8 cores	580.242s	638.057s	773.990s	711.130s	590.045s	**531.204s**

I test the six different versions of parallel Bellman-Ford on 5 complete graphs with 2000 vertices and a negative cycle of length 200 randomly generated by generateGraph.hs. The time shown above is the average across 5 graphs. The sequential version takes too much time and does not finish. The best version V3-vector (8 cores) is 1.1x faster than the second-best version V1. When the graph has 2000 vertices, V3-vector starts to outperform V1.

4.5 Effects of depth

	Depth = 2	Depth = 4	Depth = 6	Depth = 8	Depth = 10
1000 vertices	62.849s	58.786s	**54.881s**	59.448s	60.205s
2000 vertices	589.686s	558.092s	569.507s	**543.906s**	618.661s

Depth is an input used for V2-vector, V2-list, V3-list, and V3-vector. I test V3-vector with different values for depth on 2 complete graphs with 2000 vertices and a negative cycle of length 200 and 2 complete graphs with 1000 vertices and a negative cycle of length 100 randomly generated by generateGraph.hs. The time shown above is the average across 2 graphs. For 1000-vertice complete graphs, the optimal depth is likely to be around 6, while it increases to 8 for 2000-vertice complete graphs. However, the effects of depth are not considerable, possibly due to the fact that the minimum-finding step occupies only a small part of the overall algorithm.

5 Code Listing

5.1 bellmanPar.hs

```
import Data.IntMap.Strict as Map
import Data.Vector as Vector
import Data.List.Split(splitOn)
import Data.List as List
import Control.Parallel
import Control.Parallel.Strategies
import Control.Monad.Par(runPar, spawnP, get)
import System.Exit(die)
import System.Environment(getArgs, getProgName)

getVertices :: IntMap b -> [Key]
getVertices m = Prelude.map (\(i, _) -> i) (Map.toList m)

getVerticesVec :: IntMap b -> Vector Key
getVerticesVec m = Vector.fromList $ getVertices m

getVerticesMap :: IntMap b -> IntMap Key
getVerticesMap m = Map.fromList $ Prelude.map (\(i, _) -> (i,i)) (Map.toList m)

weight :: IntMap (IntMap b) -> Key -> Key -> Maybe b
weight graph i j = do
    rowMap <- Map.lookup i graph
    Map.lookup j rowMap

fromJust :: Num p => Maybe p -> p
fromJust (Just a) = a
fromJust Nothing = 0

myMinList2 :: (Ord a, Num a, Num t, Eq t) =>
                    t -> [(Maybe a, b)] -> Maybe (a, b)
myMinList2 depth xs = myMinList2Helper depth (Vector.fromList xs)
    where
        myMinList2Helper d vxs
            | Vector.length vxs == 0 = Nothing
            | Vector.length vxs == 1 && (fst $ Vector.head vxs) == Nothing = Nothing
            | Vector.length vxs == 1 && (fst $ Vector.head vxs) /= Nothing = Just (fromJust (fst $ Vector.head vxs), (snd
$ Vector.head vxs))
            | d == 0 = myMin (myMinList2Helper 0 $ Vector.fromList [Vector.head vxs]) (myMinList2Helper 0 $ Vector.tail vxs)
            | otherwise = par leftMin (myMin leftMin rightMin)
                where
                    lxs = Vector.length vxs
                    splittedVecs = Vector.splitAt (div lxs 2) vxs
                    leftMin = myMinList2Helper ((-) d 1) (fst splittedVecs)
                    rightMin = myMinList2Helper ((-) d 1) (snd splittedVecs)

myMinList1 :: (Ord a, Eq t, Num t) =>
                    t -> [(Maybe a, b)] -> Maybe (a, b)
myMinList1 _ [] = Nothing
myMinList1 _ [(Nothing, _)] = Nothing
myMinList1 _ [(Just x, i)] = Just (x, i)
myMinList1 0 (x:xs) = myMin (myMinList1 (0 :: Int) [x]) (myMinList1 (0 :: Int) xs)
myMinList1 d xs = leftMin `par` rightMin `pseq` (myMin leftMin rightMin)
    where
        lxs = List.length xs
        leftMin = myMinList1 ((-) d 1) (List.take (div lxs 2) xs)
        rightMin = myMinList1 ((-) d 1) (List.drop (div lxs 2) xs)

myMinList0 :: Ord a => [(Maybe a, b)] -> Maybe (a, b)
myMinList0 [] = Nothing
myMinList0 [(Nothing, _)] = Nothing
myMinList0 [(Just x, i)] = Just (x, i)
myMinList0 (x:xs) = myMin (myMinList0  [x]) (myMinList0 xs)

myMin :: Ord a => Maybe (a, b) -> Maybe (a, b) -> Maybe (a, b)
myMin Nothing Nothing = Nothing
myMin Nothing (Just (x, i)) = Just (x, i)
myMin (Just (x, i)) Nothing = Just (x, i)
myMin (Just (x, i)) (Just (y, j))
    | x <= y = Just (x, i)
    | otherwise = Just (y, j)

bellmanFordParV1 :: IntMap (IntMap Int) -> IntMap Key
bellmanFordParV1 graphMap = Map.mapWithKey getParent (bellmanFordHelper (getVertices graphMap))
    where
        bellmanFordHelper vertices = List.foldl' iterRelaxRound (initDistParentMap 0) [1..((List.length vertices) - 1)]
            where
                initDistParentMap i = Map.insert i (0, i) (Map.fromList [])
                iterRelaxRound dpmap1 _ = runPar $ do
                    m <- Map.traverseWithKey (\_ v -> spawnP (relaxAll v)) (getVerticesMap graphMap)
                    traverse get m
                    where
                        relaxAll target = fromJust2 (myMin (Map.lookup target dpmap1) (myMinList0 relaxResult))
```

```haskell
                            where
                                relaxResult = (Prelude.map relaxTarget vertices)
                                relaxTarget source =
                                    case (prevDist, edgeWeight) of
                                    (Nothing, Nothing) -> (Nothing, source)
                                    (Nothing, Just _) -> (Nothing, source)
                                    (Just _, Nothing) -> (Nothing, source)
                                    (Just pd, Just ew) -> (Just ((+) pd ew), source)
                                    where
                                        prevDist
                                            | a == Nothing = Nothing
                                            | otherwise = Just (fst $ fromJust2 $ a)
                                            where
                                                a = Map.lookup source dpmap1
                                        edgeWeight = weight graphMap source target
                                fromJust2 (Just a) = a
                                fromJust2 Nothing = (0,0)
        getParent _ (_,i) = i

bellmanFordParV2 :: IntMap (IntMap Int) -> IntMap Key
bellmanFordParV2 graphMap = Map.mapWithKey getParent (bellmanFordHelper (getVertices graphMap))
    where
        bellmanFordHelper vertices = List.foldl' iterRelaxRound (initDistParentMap 0) [1..((List.length vertices) - 1)]
            where
                initDistParentMap i = Map.insert i (0, i) (Map.fromList [])
                iterRelaxRound dpmap1 _ = runPar $ do
                    m <- Map.traverseWithKey (\_ v -> spawnP (relaxAll v)) (getVerticesMap graphMap)
                    traverse get m
                    where
                        relaxAll target = fromJust2 (myMin (Map.lookup target dpmap1) (myMinList0 relaxResult))
                            where
                                relaxResult = (parMap rdeepseq relaxTarget vertices) :: [(Maybe Int, Int)]
                                relaxTarget source =
                                    case (prevDist, edgeWeight) of
                                    (Nothing, Nothing) -> (Nothing, source)
                                    (Nothing, Just _) -> (Nothing, source)
                                    (Just _, Nothing) -> (Nothing, source)
                                    (Just pd, Just ew) -> (Just ((+) pd ew), source)
                                    where
                                        prevDist
                                            | a == Nothing = Nothing
                                            | otherwise = Just (fst $ fromJust2 $ a)
                                            where
                                                a = Map.lookup source dpmap1
                                        edgeWeight = weight graphMap source target
                                fromJust2 (Just a) = a
                                fromJust2 Nothing = (0,0)
        getParent _ (_,i) = i

bellmanFordParV3 :: Int -> IntMap (IntMap Int) -> IntMap Key
bellmanFordParV3 depth graphMap = Map.mapWithKey getParent (bellmanFordHelper (getVertices graphMap))
    where
        bellmanFordHelper vertices = List.foldl' iterRelaxRound (initDistParentMap 0) [1..((List.length vertices) - 1)]
            where
                initDistParentMap i = Map.insert i (0, i) (Map.fromList [])
                iterRelaxRound dpmap1 _ = runPar $ do
                    m <- Map.traverseWithKey (\_ v -> spawnP (relaxAll v)) (getVerticesMap graphMap)
                    traverse get m
                    where
                        relaxAll target = fromJust2 (myMin (Map.lookup target dpmap1) (myMinList1 depth relaxResult))
                            where
                                relaxResult = (parMap rdeepseq relaxTarget vertices) :: [(Maybe Int, Int)]
                                relaxTarget source =
                                    case (prevDist, edgeWeight) of
                                    (Nothing, Nothing) -> (Nothing, source)
                                    (Nothing, Just _) -> (Nothing, source)
                                    (Just _, Nothing) -> (Nothing, source)
                                    (Just pd, Just ew) -> (Just ((+) pd ew), source)
                                    where
                                        prevDist
                                            | a == Nothing = Nothing
                                            | otherwise = Just (fst $ fromJust2 $ a)
                                            where
                                                a = Map.lookup source dpmap1
                                        edgeWeight = weight graphMap source target
                                fromJust2 (Just a) = a
                                fromJust2 Nothing = (0,0)
        getParent _ (_,i) = i

bellmanFordParV4 :: Int -> IntMap (IntMap Int) -> IntMap Key
```

```haskell
bellmanFordParV4 depth graphMap = Map.mapWithKey getParent (bellmanFordHelper (getVertices graphMap))
    where
        bellmanFordHelper vertices = List.foldl' iterRelaxRound (initDistParentMap 0) [1..((List.length vertices) - 1)]
            where
                initDistParentMap i = Map.insert i (0, i) (Map.fromList [])
                iterRelaxRound dpmap1 _ = runPar $ do
                    m <- Map.traverseWithKey (\_ v -> spawnP (relaxAll v)) (getVerticesMap graphMap)
                    traverse get m
                    where
                        relaxAll target = fromJust2 (myMin (Map.lookup target dpmap1) (myMinList2 depth relaxResult))
                            where
                                relaxResult = (parMap rdeepseq relaxTarget vertices) :: [(Maybe Int, Int)]
                                relaxTarget source =
                                    case (prevDist, edgeWeight) of
                                    (Nothing, Nothing) -> (Nothing, source)
                                    (Nothing, Just _) -> (Nothing, source)
                                    (Just _, Nothing) -> (Nothing, source)
                                    (Just pd, Just ew) -> (Just ((+) pd ew), source)
                                    where
                                        prevDist
                                            | a == Nothing = Nothing
                                            | otherwise = Just (fst $ fromJust2 $ a)
                                            where
                                                a = Map.lookup source dpmap1
                                        edgeWeight = weight graphMap source target
                                fromJust2 (Just a) = a
                                fromJust2 Nothing = (0,0)
        getParent _ (_,i) = i

bellmanFordParV5 :: Int -> IntMap (IntMap Int) -> IntMap Key
bellmanFordParV5 depth graphMap = Map.mapWithKey getParent (bellmanFordHelper (getVertices graphMap))
    where
        bellmanFordHelper vertices = List.foldl' iterRelaxRound (initDistParentMap 0) [1..((List.length vertices) - 1)]
            where
                initDistParentMap i = Map.insert i (0, i) (Map.fromList [])
                iterRelaxRound dpmap1 _ = runPar $ do
                    m <- Map.traverseWithKey (\_ v -> spawnP (relaxAll v)) (getVerticesMap graphMap)
                    traverse get m
                    where
                        relaxAll target = fromJust2 (myMin (Map.lookup target dpmap1) (minRelaxResult depth vertices))
                            where
                                minRelaxResult _ [] = Nothing
                                minRelaxResult _ [x] = transformForMyMin $ relaxTarget x
                                minRelaxResult 0 (x:xs) = myMin (minRelaxResult 0 [x]) (minRelaxResult 0 xs)
                                minRelaxResult d xs = leftMin `par` rightMin `pseq` (myMin leftMin rightMin)
                                    where
                                        lxs = List.length xs
                                        leftMin = minRelaxResult ((-) d 1) (List.take (div lxs 2) xs)
                                        rightMin = minRelaxResult ((-) d 1) (List.drop (div lxs 2) xs)
                                relaxTarget source =
                                    case (prevDist, edgeWeight) of
                                    (Nothing, Nothing) -> (Nothing, source)
                                    (Nothing, Just _) -> (Nothing, source)
                                    (Just _, Nothing) -> (Nothing, source)
                                    (Just pd, Just ew) -> (Just ((+) pd ew), source)
                                    where
                                        prevDist
                                            | a == Nothing = Nothing
                                            | otherwise = Just (fst $ fromJust2 $ a)
                                            where
                                                a = Map.lookup source dpmap1
                                        edgeWeight = weight graphMap source target
                                transformForMyMin (Nothing, _) = Nothing
                                transformForMyMin (Just x, i) = Just (x, i)
                                fromJust2 (Just a) = a
                                fromJust2 Nothing = (0,0)
        getParent _ (_,i) = i

bellmanFordParV6 :: Int -> IntMap (IntMap Int) -> IntMap Key
bellmanFordParV6 depth graphMap = Map.mapWithKey getParent $ bellmanFordHelper (getVertices graphMap) (getVerticesVec graphMap)
    where
        bellmanFordHelper vertices verticesVec = List.foldl' iterRelaxRound (initDistParentMap 0) [1..((List.length vertices) -
1)]
            where
                initDistParentMap i = Map.insert i (0, i) (Map.fromList [])
                iterRelaxRound dpmap1 _ = runPar $ do
                    m <- Map.traverseWithKey (\_ v -> spawnP (relaxAll v)) (getVerticesMap graphMap)
                    traverse get m
                    where
                        relaxAll target = fromJust2 (myMin (Map.lookup target dpmap1) (minRelaxResult depth verticesVec))
```

```haskell
                            where
                                minRelaxResult d vxs
                                    | Vector.length vxs == 0 = Nothing
                                    | Vector.length vxs == 1 = transformForMyMin $ relaxTarget $ Vector.head vxs
                                    | d == 0 = myMin (minRelaxResult 0 $ Vector.fromList [Vector.head vxs]) (minRelaxResult 0
$ Vector.tail vxs)
                                    | otherwise = leftMin `par` rightMin `pseq` (myMin leftMin rightMin)
                                        where
                                            lxs = Vector.length vxs
                                            splittedVecs = Vector.splitAt (div lxs 2) vxs
                                            leftMin = minRelaxResult ((-) d 1) (fst splittedVecs)
                                            rightMin = minRelaxResult ((-) d 1) (snd splittedVecs)
                                relaxTarget source =
                                    case (prevDist, edgeWeight) of
                                    (Nothing, Nothing) -> (Nothing, source)
                                    (Nothing, Just _) -> (Nothing, source)
                                    (Just _, Nothing) -> (Nothing, source)
                                    (Just pd, Just ew) -> (Just ((+) pd ew), source)
                                    where
                                        prevDist
                                            | a == Nothing = Nothing
                                            | otherwise = Just (fst $ fromJust2 $ a)
                                            where
                                                a = Map.lookup source dpmap1
                                        edgeWeight = weight graphMap source target
                                transformForMyMin (Nothing, _) = Nothing
                                transformForMyMin (Just x, i) = Just (x, i)
                                fromJust2 (Just a) = a
                                fromJust2 Nothing = (0,0)
        getParent _ (_,i) = i

getCycleFromVertex :: IntMap Key -> [Key]
getCycleFromVertex parentMap = 0 : (f parentMap 0 (Map.insert 0 (1 :: Int) (Map.fromList [])))
    where
        f pMap currentVertex visited
            | parent == Nothing = []
            | Map.lookup (fromJust parent) visited /= Nothing = [fromJust parent]
            | otherwise = (fromJust parent) : f pMap (fromJust parent) (Map.insert (fromJust parent) (1 :: Int) visited)
            where
                parent = Map.lookup currentVertex parentMap

pruneCycle :: Eq a => [a] -> [a]
pruneCycle c
    | c == [] = []
    | otherwise = processCycleHelper c (List.last c)
        where
            processCycleHelper cc st
                | List.head cc == st = List.reverse cc
                | otherwise = processCycleHelper (List.tail cc) st

getCycleWeight :: Num p => [Key] -> IntMap (IntMap p) -> p
getCycleWeight c g
    | (List.length c) < 2 = 0
    | otherwise = (+) (fromJust (weight g (List.head c) (List.head tc))) (getCycleWeight tc g)
        where
            tc = List.tail c

getCycle :: Num a =>
                  [[a]] -> (IntMap (IntMap a) -> IntMap Key) -> ([Key], a)
getCycle testcase parF = cyc
    where
        cyc = ((\c -> (c, getCycleWeight c graphM)) . pruneCycle . getCycleFromVertex) parentMap
        parentMap = parF graphM
        graphM = arrayToMap testcase
        arrayToMap arr = Map.fromList $ Prelude.map f (List.zip [0..] arr)
            where
                f (i, row) = (i, Map.fromList $ List.zip [0..] row)

main :: IO ()
main = do
    args <- getArgs
    case args of
        [version, d, filename] -> do
            contents <- readFile filename
            let f line = ((Prelude.map (\x -> read x :: Int)) . (splitOn ",")) line
            let testCase = (Prelude.map f (lines contents))
            let depth = read d :: Int
            case version of
                "2" -> do
                    print "using v2"
                    print (getCycle testCase bellmanFordParV2)
                "3" -> do
                    print ("using v3 with depth == " List.++ d)
                    print (getCycle testCase (bellmanFordParV3 depth))
                "4" -> do
                    print ("using v4 with depth == " List.++ d)
                    print (getCycle testCase (bellmanFordParV4 depth))
                "5" -> do
                    print ("using v5 with depth == " List.++ d)
                    print (getCycle testCase (bellmanFordParV5 depth))
                "6" -> do
                    print ("using v6 with depth == " List.++ d)
                    print (getCycle testCase (bellmanFordParV6 depth))
                _ -> do
                    print "using v1"
                    print (getCycle testCase bellmanFordParV1)
        _ -> do
            pn <- getProgName
            die $ "Usage: " List.++ pn List.++ " <version> <filename>"
```

5.2 generateGraph.hs

```
import Data.Map as Map
import Data.List as List
import System.Random
import Control.Parallel.Strategies
import System.Exit(die)
import System.Environment(getArgs, getProgName)

generateRandomCycle :: Int -> Int -> IO([Int])
generateRandomCycle sz n = do
    g <- newStdGen
    return (List.take n . nub $ (randomRs (0, sz) g :: [Int]))

generateRandomValues :: Int -> Int -> Int -> IO([Int])
generateRandomValues 0 _ _= return []
generateRandomValues n l h = do
    r <- randomRIO (l, h)
    rs <- generateRandomValues ((-) n 1) l h
    return (r:rs)

getOldWeight :: (Num a, Ord k) => [k] -> Map k (Map k a) -> [a]
getOldWeight c g
    | (length c) < 2 = []
    | otherwise = fromJust (weight g (head c) (head tc)) : getOldWeight tc g
        where
            tc = tail c

setWeightToMap :: (Ord a1, Num a2) =>
                        Map a1 (Map a1 a2) -> [a2] -> [a1] -> Map a1 (Map a1 a2)
setWeightToMap g w c
                | (length c) < 2 = g
                | otherwise = setWeightToMap (insertGraph (insertGraph g hc htc hw) htc hc ((-1) * hw)) (tail w) tc
                    where
                        tc = tail c
                        htc = head tc
                        hc = head c
                        hw = head w

generateNewWeightForNegCycle :: Ord k =>
                                      [k] -> Map k (Map k Int) -> IO (Map k (Map k Int), Int)
generateNewWeightForNegCycle cyc graphMap = do
    let oldWeight = getOldWeight cyc graphMap
    addWeights  <- generateRandomValues ((-) (length cyc) 1) 1 100
    let newWeights = zipWith (+) oldWeight addWeights
    return ((setWeightToMap graphMap newWeights cyc), (List.foldr (+) 0 addWeights))

weight :: (Ord k1, Ord k2) =>
                Map k1 (Map k2 b) -> k1 -> k2 -> Maybe b
weight graph i j = do
    rowMap <- Map.lookup i graph
    Map.lookup j rowMap

insertGraph :: (Ord k1, Ord k2) =>
                     Map k1 (Map k2 a) -> k1 -> k2 -> a -> Map k1 (Map k2 a)
insertGraph graph i j w = Map.insert i (Map.insert j w (fromJust2 (Map.lookup i graph))) graph

initGraphMap :: (Num a, Ord k, Enum k, Num k) =>
                      k -> Map k (Map k a)
```

```
initGraphMap sz = Map.fromList $ Prelude.map f [0..((-) sz 1)]
    where
        f i = (i, Map.fromList $ zip [0..((-) sz 1)] (repeat 0))

fromJust :: Num p => Maybe p -> p
fromJust (Just a) = a
fromJust Nothing = 0

fromJust2 :: Ord k => Maybe (Map k a) -> Map k a
fromJust2 (Just a) = a
fromJust2 Nothing = Map.fromList []

getVertices :: Map b1 b2 -> [b1]
getVertices m = Prelude.map (\(i, _) -> i) (Map.toList m)

fillGraphMap :: (Ord b, Num p, Num b) =>
                      Map b (Map b p) -> Map b p -> Map b (Map b p)
fillGraphMap graph initialValueMap = List.foldl' mapRowFunc graph (getVertices graph)
    where
        mapRowFunc graph1 rowI = Map.insert rowI (Map.mapWithKey mapValFunc (fromJust2 (Map.lookup rowI graph))) graph1
            where
                mapValFunc colJ _
                    | rowI == colJ = 0
                    | rowI == 0 = fromJust (Map.lookup colJ initialValueMap)
                    | rowI > colJ = ((*) (-1) (fromJust (weight graph1 colJ rowI)))
                    | otherwise = ((-) (fromJust (weight graph1 0 colJ)) (fromJust (weight graph1 0 rowI)))

main :: IO ()
main = do
    args <- getArgs
    case args of
        [s, l, filename] -> do
            let sz = (read s :: Int)
            let negCycleLen = (read l :: Int)
            let emptyMap = initGraphMap sz
            randomValues <- generateRandomValues ((-) sz 1) (-100) 100
            let initialValues = Map.fromList (zip [0..] (0 : randomValues))
            let graphMap = fillGraphMap emptyMap initialValues
            randomCycle <- generateRandomCycle sz negCycleLen
            (finalNegGraphMap, extraWeight) <- generateNewWeightForNegCycle randomCycle graphMap
            let f gMap = (\(_, rowMap) -> (intercalate "," . Prelude.map show) (Prelude.map (\(_, x) -> x) (Map.toList rowMap)))
gMap
            writeFile (filename ++ "-graph.txt") $ intercalate "\n" ((parMap rdeepseq f (Map.toList finalNegGraphMap)) ::
[[Char]])
            writeFile (filename ++ "-cycle.txt") $ intercalate "," $ Prelude.map show randomCycle
            writeFile (filename ++ "-weight.txt") $ show extraWeight
        _ -> do
            pn <- getProgName
            die $ "Usage: " ++ pn ++ " <size> <negCycleLen> <filename>"
```