CSCE 668, Distributed Algorithms and Systems
Spring 2019
Project Report
Feiyan Yu, Boquan Tao, Ziyi Liu

Introduction

Our project is from the paper "A deterministic distributed algorithm for weighted all pairs shortest paths through pipelining". Shortest path is a popular research field in distributed algorithm. In this paper, the authors presents an algorithm which can calculate the shortest path from different sources. The algorithm solves (h, k)-SSP problem, which means that there are k sources which we can arbitrarily assign and every shortest path does not exceed h hop length. If we assign k sources to be all nodes in the graph, then we get APSP(all pairs shortest paths) problem. In an older paper also proposed by the same authors, another algorithm is presented to solve APSP which requires single source shortest path as a sub-routine and introduces the concept of blocker set. The new algorithm therefore can generalize better. This algorithm, like the older version, needs to assume synchronization. The round number needed to complete the (h, k)-SSP problem is $2\sqrt{\Delta kh} + k + h$. If we want to solve ASAP and set k to be n, then the round number is $2n\sqrt{\Delta} + 2n$. All shortest distances are at most Δ. Δ is a parameter which we need to give to the algorithm. A simple way of getting Δ is to input the longest path in the graph. The old algorithm will finish in $O(n^{\frac{3}{2}})$ rounds which is asymptotically worse than $2n\sqrt{\Delta} + 2n$.

The algorithm runs in CONGEST model in which every link's bandwidth is bounded. Every node only knows its incident edges. The graph can be directed or undirected. The weight of each edge is non-negative. The old algorithm can tolerate negative edges as long as there are no negative cycles.

Algorithm

An innovative feature of this algorithm is the use of κ instead of just weighted distance. More specifically, $\kappa = d * \gamma + l$, where $\gamma = \sqrt{kh/\Delta}$. So κ inherits from weighted distance d and hop length l. Another interesting part is that every node keeps a list $list_v$ which stores entry Z. Z is of the form $[\kappa, d, l, x]$, where x is the source node. Although some path can never be shortest paths, the storage of it can enable the algorithm to terminate in $O(\sqrt{\Delta kh})$. Because there are many variables, it is clearer to see in a table.

The entries in $list_v$ are sorted in the order (d, l, x). Initially, every node sets their distance to sources infinity. The sources set its distance to itself 0, and add the entry $(0, 0, 0, v)$ to its own $list_v$. All other non source nodes have an empty $list_v$ at the beginning. $Z.v$ represents the number of entries in $list_v$ for source x which are below and at Z. The message sent is in the form $(Z, Z.flag - d^*, Z.v)$. When receiving a message Z^-, it first constructs a new entry Z based on that message. The hop length will increment by 1, the distance will add the weight on that edge and κ will change accordingly. If the new distance is smaller than the current shortest distance for that source, we then set the shortest distance to the new distance. And if there are less than $Z^-.v$ entries in $list_v$ for source x whose key value κ are smaller than $Z.\kappa$, then we just

insert that Z to $list_v$. If not, this new Z will never be a shortest path entry and we can just ignore it.

Because $flag - d^*$ and p are also attached to entry Z, in our implementation, we add those two in our Z. So now Z is in the form: $(\kappa, d, l, x, flag - d^*, p)$. The graph in presented in a matrix, where -1 means there is no edge between the two nodes and non-negative number shows the weight.

Table 1: Global parameters.

S	set of sources
k	number of sources
h	maximum number of hops in a shortest path
Δ	maximum weighted distance of a shortest path
n	number of nodes
γ	parameter equal to $\sqrt{kh/\Delta}$

Table 2: Local variables at node v.

d_x^*	current shortest path distance from x to v
$list_v$	list at v for storing the SP and non-SP entries

Table 3: Variables for entry $Z = (\kappa, d, l, x)$ in $list_v$.

κ	$\kappa = d * \gamma + l$
d	weight (distance) of the path associated with this entry
l	hop-length of the path associated with this entry
x	start node (i.e. source) of the path associated with this entry
p	parent node of v on the path associated with this entry
v	number of entries for source x at or below Z in $list_v$ (not stored explicitly)
$flag - d^*$	flag to indicate if Z is the current SP entry for source x
pos	position of Z in $list_v$ in a round r

Algorithm 1 $(h, k) - SSP$ algorithm

Initialization for node v:

1: for each $x \in S$ do $d_x^* \leftarrow \infty$
2: if $v \in S$ then $d_x^* \leftarrow 0$; add entry $Z = (0, 0, 0, v)$ to $list_v$; $Z.flag - d^* \leftarrow true$

Algorithm at node v for round r:

1: send: if there is an entry Z with $\lceil Z.\kappa + pos(Z) = r \rceil$
2: then compute $Z.v$ and form the message $M =< Z, Z.flag - d^*, Z.v >$ and send M to all neighbors.
3: receive: let I be the set of incoming messages
4: for each $M \in I$ do
5: let $M = (Z^- = (\kappa^-, d^-, l^-, x), Z^-.flag - d^*, Z^-.v)$ and let the sender be y
6: $\kappa \leftarrow \kappa^- + w(y, v) * \gamma + 1$; $d \leftarrow d^- + w(y, v)$; $l \leftarrow l^- + 1$
7: $Z \leftarrow (\kappa, d, l, x)$; $Z.flag - d^* \leftarrow false$; $Z.p \leftarrow y$
8: let Z^* be the entry for x in $list_v$ such that $Z^*.flag - d^* = true$ if such an entry exists.
9: if $Z^-.flag - d^* = true$ and $l <= h$ and $((d < d_x^*)$ or $(d = d_x^*$ and $Z.l < Z^*.l)$ or $(d = d_x^*$ and $Z.l = Z^*.l$ and $Z.p < Z^*.p)$
10: then $d_x^* \leftarrow d$; $Z.flag - d^* \leftarrow true$
11: if Z^* exists
12: $Z^-.flag - d^* = false$
13: INSERT(Z)
14: else
15: if there are less than $Z^-.v$ entries for x with $key \leq Z.\kappa$ then INSERT(Z)

INSERT(Z):

1: insert Z in $list_v$ in sorted order of (κ, d, x)
2: if $\exists$ an entry Z' for x in $list_v$ such that $Z'.flag - d^* = false$ and $pos(Z') > pos(Z)$ then
3: find Z' with smallest $pos(Z')$ such that $pos(Z') > pos(Z)$ and $Z'.flag - d^* = false$
4: remove Z' from $list_v$

Implementation

---

1. Implementation of Graph API

   This algorithm is based on graph, so we built a graph class with some APIs for implementing the main algorithm. We used adjacency matrix to show the weights in the graph. What we need is to find the neighbors for each vertex and get the weight of each edge.

2. Simulation of global round number in synchronous case

   As for the tool in this project, DistAlgo, has a nature of asynchronous condition. So, it could not resolve the synchronous case directly, also, users could not add a global variables directly for each process. However, the requirement of this algorithm needs every process to know the round number even if this process does not receive any message in this round. So, after each round of sending

message of shortest path, we implemented a synchronization step for making sure all the processes could successfully get into the same round.

Our strategy was simple at this step. We made all the processes send messages to every process contains the local round number. After receiving all the messages with the same round number as the local, the local round number increment by 1.

Algorithm 2 Round Number Synchronization for process p_i

Require:

Initially $p_i.round = 0$ and $p_i.received\_count = 0$

First Send message:

1: send message $M =< p_i.round >$ to the all n processes(including itself) and wait for receiving messages

Then Receive message M:

2: **if** $M.round == p_i.round$ **then**
3: $p_i.received\_count = p_i.received\_count + 1$
4: **else**
5: put M into buffer for later review
6: **end if**
7: **if** $p_i.received\_count == n$ **then**
8: $p_i.round = p_i.round + 1$
9: Terminate
10: **end if**

Termination: Without any failures, all message should be received. So, before terminate, the process should receive n messages. If all the n message has the same round number, the algorithm terminate.

Correctness: Without any failures, this synchronization could ensure that all the processes terminates, all process are at the same round.

Proof: Lemma 1: At any stage, $p_i.round - p_j.round \leq 1$. Proof: If $p_i.round - p_j.round > 1$, that means p_i have received n message $M.round = p_i.round - 1 > o_j.round$. Because there is no duplicated message and no failure, the n message must include a message with round number $M.round = p_i.round - 1$ send by p_j. It raises contradiction because p_j could not send a message with round number greater than its round.

So, the after the slowest processor terminates, and has not send the synchronization message at the next round, all the processes have the same round number. Because during the period between the slowest process terminates and it send the synchronization message for the next round, all processors has the same round number, we could use that time to implement our algorithm. Even if the actual situation is asynchronous, all the process uses the same and correct round number as a parameter. We could think that round number is global now.

3. Implementation of (h,k)-SSP algorithm

   The implementation of the algorithm strictly followed the steps of the (h,k)-SSP in the paper. Because the round number only serve as the condition to determine whether p_i should send message or not, we just user the time interval when all process has the same round number to check it and send (h,k)-SSP message.

   The overall step of this implementation is showed below:

Algorithm 3 (h,k)-SSP(hop, r_{limits}, Δ, Graph, Sources)

Require:

1: Initialization of loacal variable.
2: **while** $p_i.round < r_{limit}$ **do**
3: ASPS_algorithm()
4: Synchronization()
5: **end while**

Results

We test our algorithm on different graphs and compare the result with another APSP algorithm - Dijkstra's algorithm. As for the comparison, we use some example at Visualgo .

The first example is a weighted directed acyclic graph, and the topology of the graph is as figure 1.

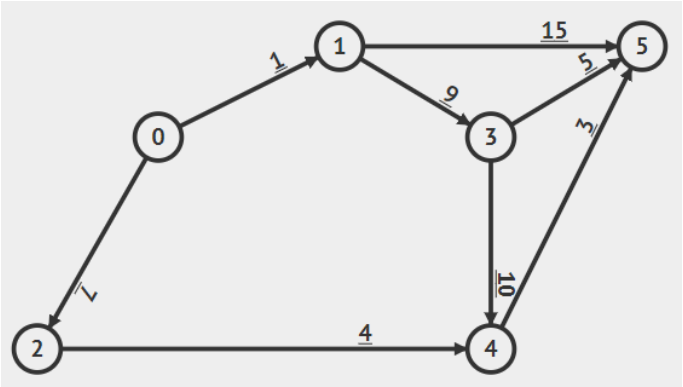


Figure 1: Test graph 1

The following is the mapping between the processor's name and the node id.

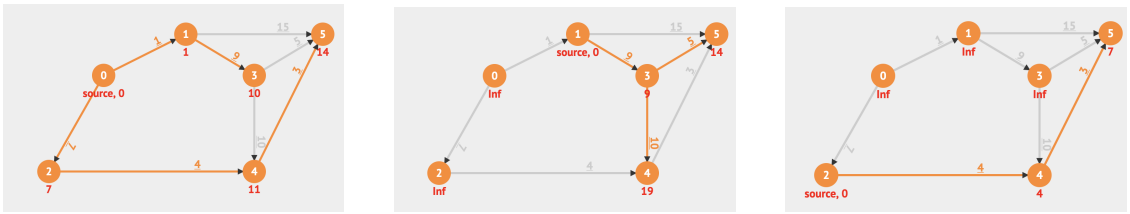
$$
\begin{aligned}
&< P : bfc04 > : 0, < P : bfc02 > \quad : 1, \\
&< P : bfc07 > : 2, < P : bfc03 > \quad : 3, \\
&< P : bfc05 > : 4, < P : bfc06 > \quad : 5.
\end{aligned}
$$

Then the following table shows the results of our algorithm.

processor	$\langle P : bfc04 \rangle$	$\langle P : bfc02 \rangle$	$\langle P : bfc07 \rangle$	$\langle P : bfc03 \rangle$	$\langle P : bfc05 \rangle$	$\langle P : bfc06 \rangle$
$\langle P : bfc04 \rangle$	0	inf	inf	inf	inf	inf
$\langle P : bfc02 \rangle$	1	0	inf	inf	inf	inf
$\langle P : bfc07 \rangle$	7	inf	0	inf	inf	inf
$\langle P : bfc03 \rangle$	10	9	inf	0	inf	inf
$\langle P : bfc05 \rangle$	11	19	4	10	0	inf
$\langle P : bfc06 \rangle$	14	14	7	5	3	0

Table 4: Result of our algorithm on graph 2

Then we compare the results of the shortest path of single sources (corresponding to each column in the table) from Visualgo's Dijkstra's algorithm.



(a) source: node 0 (b) source: node 1 (c) source: node 2

(d) source: node 3 (e) source: node 4 (f) source: node 5

Figure 2: Result of graph 1 from visualgo

The second example is a weighted undirected tree, and the topology of the graph is as figure 3.

The following is the mapping between the processor's name and the node id.

$$
\begin{aligned}
< P : 47003 > : 0, < P : 47007 > \quad &: 1, \\
< P : 47004 > : 2, < P : 47002 > \quad &: 3, \\
< P : 47005 > : 4, < P : 47006 > \quad &: 5.
\end{aligned}
$$

Then table 5 shows the results of our algorithm.

Then figure 4 shows results of the shortest path of single sources (corresponding to each column in the table) from Visualgo's Dijkstra's algorithm.

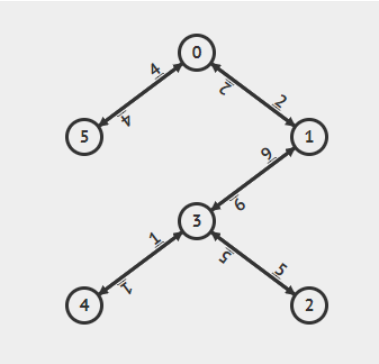
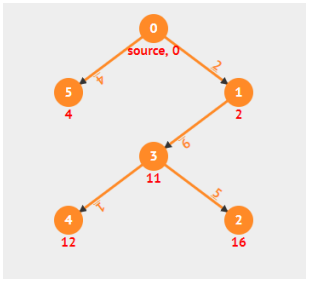


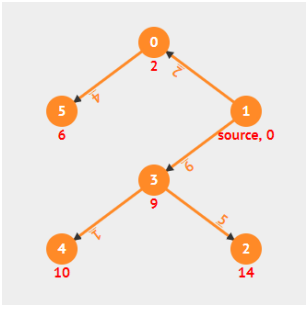
Figure 3: Test graph 2

processor	$\langle P : bfc04 \rangle$	$\langle P : bfc02 \rangle$	$\langle P : bfc07 \rangle$	$\langle P : bfc03 \rangle$	$\langle P : bfc05 \rangle$	$\langle P : bfc06 \rangle$
$\langle P : bfc04 \rangle$	0	2	13	8	9	4
$\langle P : bfc02 \rangle$	2	0	11	6	7	6
$\langle P : bfc07 \rangle$	13	11	0	5	6	inf
$\langle P : bfc03 \rangle$	8	6	5	0	1	12
$\langle P : bfc05 \rangle$	9	7	6	1	0	inf
$\langle P : bfc06 \rangle$	4	6	inf	12	13	0

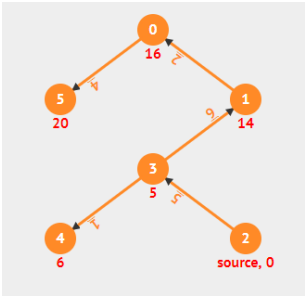
Table 5: Result of our algorithm on graph 3



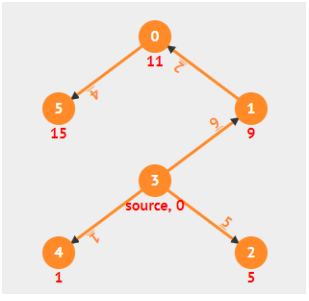
(a) source: node 0



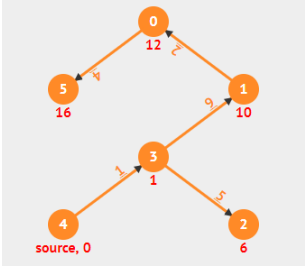
(b) source: node 1



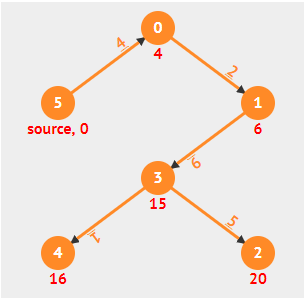
(c) source: node 2



(d) source: node 3



(e) source: node 4

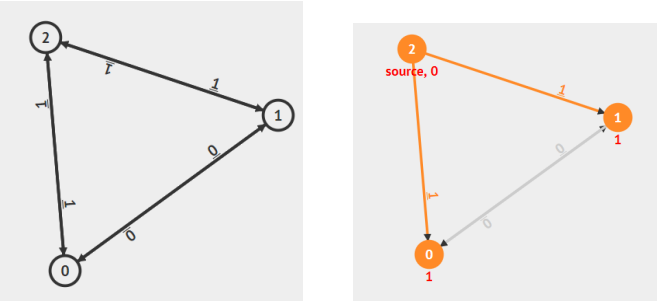


(f) source: node 5

Figure 4: Result of graph 2 from visualgo

The following three examples are two negative weighted examples and a graph with a zero weight edge.

In the graph with a zero weight edge, the algorithm will give the shortest path without including the zero weight unless the zero weight edge is necessary for the path. In result, we also give the parent of node on the shortest path.



(a) original graph (b) source: node 2

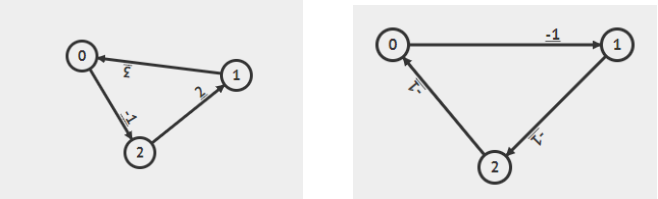
Figure 5: graph with a zero weight edge

The following is the output of the algorithm. As for source node 2, both node 0 and node one's immediate parent is node 2, which means since the zero weight edge is not necessary for the shortest path, it is not included.

---

source $\langle P : 81004\rangle$:2
$\langle P : 81002\rangle$: 0's parent = $\langle P : 81004\rangle$
$\langle P : 81003\rangle$: 1's parent = $\langle P : 81004\rangle$
output = $\langle P : 81004\rangle$: 1
output = $\langle P : 81004\rangle$: 1
output = $\langle P : 81004\rangle$: 0

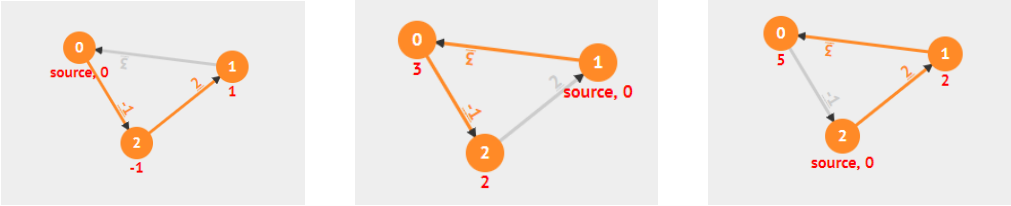
In graphs with negative weight edges, the algorithm is not able to give the correct answer.



(a) graph with only one negative edge (b) graph with negative cycle

Figure 6: Two graphs with negative weight edges

Judging from the output of the algorithm and comparing with the correct result, the algorithm gives the wrong result.



(a) source: node 0 (b) source: node 1 (c) source: node 2

Figure 7: Visualgo result for the first negative weight example

---

$\langle P : 11002\rangle$: 0, $\langle P : 11003\rangle$: 1,$\langle P : 11004\rangle$: 2
$\langle P : 11002\rangle$:OUTPUT: output = $\langle P : 11002\rangle$: 0, $\langle P : 11003\rangle$: 3, $\langle P : 11004\rangle$: 5
$\langle P : 11003\rangle$:OUTPUT: output = $\langle P : 11002\rangle$: inf, $\langle P : 11003\rangle$: 0, $\langle P : 11004\rangle$: 2
$\langle P : 11004\rangle$:OUTPUT: output = $\langle P : 11002\rangle$: -1, $\langle P : 11003\rangle$: 2, $\langle P : 11004\rangle$: 0

It is known that there is no answer to the shortest path on negative weight cycles, but some algorithms like Bellman Ford's algorithm could detect negative cycles in the graph. After testing the negative weight cycle on this algorithm, it gives the same answer no matter how many rounds or hops we set. The answer is following.

$\langle P : 4c002\rangle$: 0, $\langle P : 4c003\rangle$: 1,$\langle P : 4c004\rangle$: 2
$\langle P : 4c002\rangle$:OUTPUT: output = $\langle P : 4c002\rangle$: 0, $\langle P : 4c003\rangle$: inf, $\langle P : 4c004\rangle$: -1
$\langle P : 4c003\rangle$:OUTPUT: output = $\langle P : 4c002\rangle$: -1, $\langle P : 4c003\rangle$: 0, $\langle P : 4c004\rangle$: inf
$\langle P : 4c004\rangle$:OUTPUT: output = $\langle P : 4c002\rangle$: inf, $\langle P : 4c003\rangle$: -1, $\langle P : 4c004\rangle$: 0

Conclusion

This algorithm is a strong and simple algorithm for finding shortest paths for k sources. The round for all pairs shortest paths when k is set to n is $2n\sqrt{\Delta} + 2n$, which is linear to n. Although every node has to keep a list $list_v$ locally, the number of entries in the list is at most $\sqrt{\Delta h/k} + 1$, which is acceptable.

We use DistAlgo to implement this algorithm and the results are correct. This algorithm avoids going through edges with big weights because at every round, an entry $Z = (\kappa, d, l, x)$ would be sent only when $\lceil Z.\kappa + pos(Z) = r \rceil$, so if the edge weight is too big, κ would also be very big, then this entry will be sent at a very late round. This strategy helps early termination and avoids sending useless message. But this strategy makes the algorithm not usable to negative edges. When a negative edge is present, then κ can be negative. When the algorithm is checking whether $\lceil Z.\kappa + pos(Z) = r \rceil$, this equation can never be satisfied because this entry is less than round number r and r is always increasing.

Assessment of DistAlgo

As stated in the official GitHub, DistAlgo is a very high-level language for distributed algorithms. The language is based on python.

To begin with, the installation of DistAlgo is simple. Although we found some problem when installing DistAlgo with Python 3.7, it works great on Python 3.6. It could be easily installed with pip. Running DistAlgo is a little bit inconvenient since we can only use its compiler. When running, we found that DistAlgo also generates an interpreted python file, which could help us to understand how to DistAlgo works. DistAlgo does have some official example algorithms, like leader election, mutual exclusion, Paxos algorithm, to help understand the language, and we found them helpful. Among all the advantages, the greatest one must be how close the language is to the pseudocode in papers, which enables us to implement algorithms without simulating a distributed environment.

Nevertheless, there are some problems we found during implementation. First of all is that the official documents are too concise, or may be incomplete. We cannot find enough information about the keywords, syntax, and how to implement a distributed algorithm by simply reading the documents. Only after following official examples can we understand how to implement an algorithm. Moreover, we found that we cannot run the algorithm with more than about 80 processors, which will lead the program to crash. It is acceptable for just implementing algorithms in distributed papers, but may not be good enough for industrial scales. One last problem is that there is no synchronous environment embedded in the language, so we have to simulate synchronous over asynchronous at first to implement our proposed algorithm.

To summarize, although not so friendly to beginner, DistAlgo is a great tool for learning and understanding algorithms in publications but needs further improvement to solve more complicated and larger-scale problems.