Approximate All-Pairs Shortest Paths

Charles Y. Zheng and Jinshu Wang
Department of Statistics
Stanford University
Stanford, CA 94305
{snarles, jinshuw}@stanford.edu

Arzav Jain
Department of Computer Science
Stanford University
Stanford, CA 94305
arzavj@stanford.edu

Abstract

This document describes in detail our approximate algorithm for all-pairs-shortest-paths, Block Floyd-Warshall. It will be later incorporated into the Appendix for the paper isomap.pdf.

1 Introduction

1.1 Definitions

Let $G = (V, E)$ be a weighted directed graph with weights $w_{i,j}$ for edges $(i \to j) \in E$. Define the adjacency matrix A as a square matrix with dimension $n = |V|$, and entries

$$A_{ij} \begin{cases} w_{i,j} & \text{if } (i \to j) \in E \\ 0 & \text{if } i = j \\ \infty & \text{if } (i \to j) \notin E \end{cases}$$

Further assume that A is strongly connected; that is, for all $i, j \in A$, there exists a *path* $(v_0 \to v_1 \to \ldots \to v_l)$ with $v_0 = i$, $v_l = j$, $l \geq 0$ such that for all $k \in 0, \ldots, l$, $(v_k \to v_{k+1}) \in E$. For any given path $p = (v_0 \to \ldots \to v_l)$, define the weight of the path $w(p)$ as the sum of the edges

$$w(p) = \sum_{i=1}^{l} e_{v_i, v_{i+1}}$$

and define the *length* of the path $\ell(p)$ as the number of edges in the path, $\ell(p) = l$. For $i, j \in V$ let $\mathcal{P}_{i,j}$ denote the set of all directed paths from i to j, and $\tilde{\mathcal{P}}_{i,j}$ denote the set of *shortest directed paths* from i to j, defined by

$$\tilde{\mathcal{P}}_{i,j} = \{\tilde{p} \in \mathcal{P}_{i,j} : w(\tilde{p}) = \min_{p \in \mathcal{P}_{i,j}} w(p)\}$$

Let $\ell_{i,j}$ be the *minimum length* of the shortest directed path in $\mathcal{P}_{i,j}$,

$$\ell_{i,j} = \min_{p \in \tilde{\mathcal{P}}_{i,j}} \ell(p)$$

Finally, let ℓ_G be the *shortest-path diameter* of the graph G, defined as

$$\ell_G = \max_{i,j \in V} \ell_{i,j}$$

Given an $n \times m$ matrix A and an $n \times m$ matrix B, define the entrywise minimum $C = \min(A, B)$ by

$$C_{ij} = \min(A_{ij}, B_{ij})$$

Meanwhile, given an $n \times k$ matrix A and a $k \times m$ matrix B, define the *min-plus* product $C = A \otimes B$ by

$$C_{i,j} = \min_{l=1}^{k} A_{il} + B_{lj}$$

for $i = 1, \ldots, n$ and $j = 1, \ldots, m$. For a square matrix A, let A^k denote the min-plus kth power of A,

$$A^k = \underbrace{A \otimes A \otimes \cdots \otimes A}_{k \text{ times}}$$

1.2 All-pairs shortest paths

We consider the problem of *all-pairs shortest-paths* (APSP) as the following: given an $n \times n$ adjacency matrix A describing graph (G, V), compute the $n \times n$ matrix S if all shortest-path distances, with

$$S_{ij} = \min_{p \in \mathcal{P}_{i,j}} w(p)$$

Define the k-shortest path matrix $S^{(k)}$ by

$$S_{ij}^{(k)} = \begin{cases} \min_{p \in \mathcal{P}_{i,j} : \ell(p) \leq k} w(p) & \text{if } \min_{p \in \mathcal{P}_{i,j}} \ell(p) \leq k \\ \infty & \text{otherwise} \end{cases}$$

Note that $A = S^{(1)}$; it is easily verified that $A^k = S^{(k)}$ for all $k \geq 1$. Furthermore, we have $S^{(\ell_G)} = S$ since for any pair $i, j \in V$ there exists a shortest path with length at most ℓ_G.

This motivates the iterative squaring algorithm for shortest pairs.

Algorithm 1 Iterative Squaring for APSP

```
function ITERSQUARE(Adjacency matrix A)
    S ← A
    for k = 1, ..., ⌈log_w(ℓ_G)⌉ do
        S ← S^2
    end for
    Return S
end function
```

On a single machine, the cost of each squaring operation is $O(n^3)$. Hence the computational cost of iterative squaring is $O(\log(\ell_G)n^3)$. Since ℓ_G is at most n, the worst-case cost is therefore $O(\log(n)n^3)$. The Floyd-Warshall algorithm improves substantially on this cost:

Algorithm 2 Floyd-Warshall algorithm for APSP

```
function FLOYDWARSHALL(Adjacency matrix A)
    S ← A
    for k = 1, ..., n do
        Let S_{·,k} denote the kth column and S_{k,·} denote the kth row of S
        S ← min(S, S_{·,k} ⊗ S_{k,·})
    end for
    Return S
end function
```

The cost of computing $\min(S, S_{\cdot,k} \otimes S_{k,\cdot})$ in each iteration is $O(n^2)$. Since there are n iterations, the cost is therefore $O(n^3)$, which is strictly better than iterated squaring regardless of the shortest-path-diameter of the graph. However, we will see that in a distributed setting, Floyd-Warshall is no longer strictly better.

1.3 Distributed setting

Consider a network of p worker nodes arranged in an $\sqrt{p} \times \sqrt{p}$ grid. We denote the entire grid by Λ and a particular worker by $\Lambda[i, j]$. For any $n \times n$ square matrix A, we say that A is stored on the grid if $\Lambda[i, j]$ stores corresponding $n/\sqrt{p} \times n/\sqrt{p}$ block of A. The crucial property of this block structure is that for any pair of matrices A, B, stored on the grid, then if the number of rows of A match the number of rows of B, then the number of rows of each block of A matches the number of rows of each corresponding local block of B, and that the analagous property holds if the number of columns of A matches the number of columns of B.

Hence our setup resembles the computing grids studied in the supercomputing literature[1] except that we make different assumptions about communication costs, which are more appropriate to distributed computing "in the cloud". Given our motivation of developing algorithms for cloud computing, our main objective is to measure the total *waiting time* needed for the entire calculation to complete, and secondarily the total *expense* as measured by number of workers times waiting time. These measures of cost are the most appropriate for cloud computing, where waiting time may be a crucial factor (e.g. for streaming applications) and where financial cost corresponds to worker-hours.

In particular, we separate the cost of communication and computation, so that sending messages is assumed to cost zero CPU; however, we only allow one message to be sent or recieved at any time, and a machine that is sending is not open for recieving and vice versa. However, mutual disjoint pairs of workers can communicate simultaneously. Also, we assume a homogenous cost of communication between any pair of workers. Let a message M consist of w words. The time $T(M)$ it takes for a worker $\Lambda[i, j]$ to send a message M to another worker $\Lambda[k, l]$ consisting of w words is determined by the latency of the network κ_L and the inverse transmission rate κ_T and is given by

$$T(M) = \kappa_L + \kappa_T w$$

Following these assumptions, the cost for a single worker to broadcast a message to q other workers is given by

$$T_B(M) \approx \log(q)(\kappa_L + \kappa_T w)$$

since approximately $\log(q)$ rounds of one-to-one transmission are needed. If multiple messages M are broadcasted simultaneously (possible from the same source), the one-to-one communication rounds can be arranged in a way so that the waiting time for the entire process is only a constant larger than the time for the $\max_M T_B(M)$. Let therefore make the simplifiying assumption that the cost for the simultaneous broadcast of b messages, all of size w, and each to q recipients is

$$T_B(M_1, \ldots, M_b) \approx \log(q)(\kappa_L + \kappa_T w) + \kappa_S b$$

where κ_S is a constant describing the additional time per message.

We take the computational cost for a single worker to complete one operation to be κ_C. On a single core, we take the time of entrywise minimum to be $\kappa_C nm$ and the time of min-plus multiplication to be $2\kappa_C nmk$, for input matrices of the dimensions as described in the respective definitions.

We define the following algorithms for distributed entrywise minimum and distributed min-plus multiplication.

Algorithm 3 Distributed Entrywise Minimum

function DISTRIBUTEDMIN($n \times m$ matrices A, B)
  Let $A[i, j]$ denote the block of A owned by worker $\Lambda[i, j]$, and similarly define $B[i, j]$
  for $i, j = 1, \ldots, \sqrt{p}$ in parallel **do**
    Define $C[i, j] = \min(A[i, j], B[i, j])$
  end for
  Result C is stored on grid Λ
end function

Algorithm 4 Distributed Min-Plus multiplication

function DISTRIBUTEDMPM($n \times k$ matrix A, $k \times m$ matrix B)
  Let $A[i, \cdot]$ denote the blocks of A owned by workers $\Lambda[i, \cdot]$, and similarly define $B[\cdot, j]$
  for $i = 1, \ldots, \sqrt{p}$ and $j = 1, \ldots, \sqrt{p}$, asynchronously **do**
    Broadcast $A'_i = A[i, \cdot]$ to all workers $\Lambda[i, \cdot]$
    Broadcase $B'_j B[\cdot, j]$ to all workers $\Lambda[\cdot, j]$
  end for
  for $i, j = 1, \ldots, \sqrt{p}$ in parallel **do**
    Set $C[i, j] \to A'_i \otimes B'_j$
  end for
  Result C is stored on grid Λ
end function

Here the notation A'_i, B'_j is intended to distinguish broadcasted copies from local matrices. The algorithm DISTRIBUTEDMPM only works if each block $A[i, \cdot]$ and $B[\cdot, j]$ fit in worker memory. Otherwise, one has to split the job into smaller parts.

Algorithm 5 Large-scale distributed Min-Plus multiplication

function DISTRIBUTEDMPM2($n \times k$ matrix A, $k \times m$ matrix B)
  Let $A = [A_1, \ldots, A_q]$ where each A_i fits in memory.
  Let $B^T = [b_1^T, \ldots, b_q^T]$ where each b_i fits in memory.
  Intialize distributed $C = 0$ with dimension $n \times m$
  for $k = 1, \ldots q$ **do**
    $C \leftarrow$ DISTRIBUTEDMIN(C, DISTRIBUTEDMPM(A_k, b_k))
  end for
  Result C is stored on grid Λ
end function

We now state the cost of of these algorithms assuming that the inputs are already in place on the network.

- DISTRIBUTEDMIN requires no communication, hence the cost is the cost of communication. Each worker must complete a local entrywise min, hence the waiting time is $\kappa_C nm/p$.
- DISTRIBUTEDMP requires a simultaneous broadcast of $2\sqrt{p}$ messages, each message having size at most $k \max(n, m)/\sqrt{p}$ and $\sqrt{p}$ recipients. Afterwards, each worker must complete a local min-plus multiplication. Hence, the waiting time is bounded by

$$\frac{1}{2}\log(p)(\kappa_L + \kappa_T k \max(n, m)/\sqrt{p}) + \kappa_C nmk/p$$

We will analyze the cost of DISTRIBUTEDMPM2 in the following analyses.

Henceforth we use DISTRIBUTEDMIN, DISTRIBUTEDMPM, and DISTRIBUTEDMPM2 as building blocks for our distributed versions of ITERSQUARE and FLOYDWARSHALL. Up to a change in notation, our distributed Floyd-Warshall is essentially the same as described in Kumar [1].

Algorithm 6 Distributed Iterative Squaring

function D-ITERSQUARE(Adjacency matrix A)
  $S \leftarrow A$
  for $k = 1, \ldots, \lceil \log_w(\ell_G) \rceil$ **do**
    $S \leftarrow$ DISTRIBUTEDMPM2(S, S)
  end for
  Result S stored on grid Λ
end function

Let us assume that each worker can store at least $3n^2/p$ words in RAM. It follows that p chunks are needed in DISTRIBUTEDMPM2, which therefore involves an alternating sequence of p calls

each of DISTRIBUTEDMPM and DISTRIBUTEDMIN, where each call can only begin as soon as the previous call is completed. The entire procedure requires $d_G = \lceil \log_2(\ell_G) \rceil$ such calls of DISTRIBUTEDMPM2. Therefore the total waiting times of D-ITERSQUARE, as a function of n, ℓ_G, p is

$$\begin{aligned} T_{D-IterSquare}(n, d_G, p) =& d_G p \left[\frac{1}{2} \log(p)(\kappa_L + \kappa_T (n/p)(n/\sqrt{p})) + 2\sqrt{p}\kappa_S + \kappa_C n^2 (2(n/p) + 1)/p \right] \\ =& \frac{p \log(p) d_G}{2} \kappa_L + 2 d_G p^{3/2} \kappa_S + \frac{n^2 d_G}{\sqrt{p}} \kappa_T + \frac{d_G n^2 (2n + p)}{p} \kappa_C \end{aligned}$$

Compare to distributed Floyd-Warshall:

Algorithm 7 Distributed Floyd-Warshall

function D-FLOYDWARSHALL(Adjacency matrix A)
  $S \leftarrow A$
  for $k = 1, \ldots, n$ **do**
    Let $S_{\cdot,k}$ denote the kth column and $S_{k,\cdot}$ denote the kth row of S
    Set $S' \leftarrow$ DISTRIBUTEDMPM$(S_{\cdot,k}, S_{k,\cdot})$
    $S \leftarrow$ DISTRIBUTEDMIN(S, S')
  end for
  Return S
end function

Distributed Floyd-Warshall involves an alternating sequence of n calls each to DISTRIBUTEDMPM and DISTRIBUTEDMIN where each call must be completed in sequence. Therefore the total waiting time is

$$T_{D-FloydWarshall}(n, d_G, p) = \frac{n \log(p)}{2} \kappa_L + 2n\kappa_S + \frac{n^3 \log(p)}{\sqrt{p}} \kappa_T + \frac{3n^3}{p} \kappa_C$$

It is then evident that for small log-diamterers d_G, large latencies κ_L and sufficiently large p, that $T_{D-FloydWarshall}$ can exceed $T_{D-IterSquare}$. Note that this depends on knowing a good upper bound for d_G, which is possible in many applications.

2 Block Floyd-Warshall

Note that both Floyd-Warshall and Iterative Squaring may be viewed as special cases of a more general algorithm, *Block Floyd-Warshall* with parameters K and L.

The parameter K determines the number of both horizontal and vertical *blocks* to split A. When $K = n$, each block is a single row or column of A, as seen in the Floyd-Warshall update step. When $K = 1$, each block is the matrix A itself, as seen in iterative squaring. The parameter L determines the number of *outer loops*. Floyd-Warshall has no outer loop, so $L = 1$. Iterative squaring has $L = d_G$ outer loops.

Algorithm 8 Block Floyd-Warshall

function BLOCKFLOYDWARSHALL(Adjacency matrix A)
  Let $M = n/k$.
  $S \leftarrow A$
  for $l = 1, \ldots, L$ **do**
    for $k = 1, \ldots, M$ **do**
      Let $S = [S_1, \ldots, S_M]$, and $S^T = [s_1^T, \ldots, s_M^T]$
      $S \leftarrow \min(S, S_k \otimes s_k)$
    end for
  end for
  Return S
end function

For any given parameter K, Block Floyd-Warshall computes the APSP matrix given $L \geq n/K$, as stated in the following theorem.

2.1 Probabilistic Guarantees

2.2 Distributed Block Floyd-Warshall

References

[1] V. Kumar and V. Singh. Scalability of parallel algorithms for the all-pairs shortest-path problem. *Journal of Parallel and Distributed Computing*, 13(2):124–138, Oct. 1991.