Algorithms for Massive Data Science	Lecture date: Mar.20, 2019
Instructor: Ben Moseley	Scribe: Yuyan Wang

1 Introduction to Massively Parallel Model

1.1 Massively Parallel Model (MPC)

In Massively Parallel Model, the input size is $O(n)$ and the number of machines is $O(m)$. The memory assigned to each machine is $\tilde{O}(\frac{n}{m})$. We can think of m as satisfying $m = n^{\epsilon}$ for some constant $\epsilon > 0$. It is desirable to make the model work for all small constant $\epsilon > 0$, in which case both the memory on each machine and number of machines is sublinear. They are both small compared to the large data size. The total memory will then be $\tilde{O}(n)$.

In MPC we want to minimize the number of rounds. For instance, people are used to want to solve interesting problems in $O(1)$ or even 2 rounds on MapReduce. Polylogarithmic number of rounds are usually feasible today. Another thing people want in MPC is to keep runtime per machine polynomial in RAM model.

1.2 Extra Memory

Sometimes it is also useful to allow some extra memory in the MPC setting. Instead of $\tilde{O}(n)$, the total memory is $O(n^{1+\frac{1}{2}\epsilon})$, and the machine number m is at least n^{ϵ} for $0 \leq \epsilon < 2$. In this case the memory on each machine is $O(n^{1-\frac{1}{2}\epsilon})$, ensured to be still sublinear. Although the bound on total input size is relaxed, this relaxation may help us discover good algorithms. ,'

2 Partitioning Algorithms

2.1 Rough Sketch

In this approach the data is partitioned evenly across machine arbitrarily (or randomly). A *sketch* is computed on each machine, and all the sketches are communicated to one machine. This machine will then use the sketches received to compute the final solution.

This approach tends to work when there is little to no dependence across the partitions. For example, it works well for word count, as is mentioned in previous lecture, since the word count on every partitioned data is quite independent of each other. But surprisingly, it may also work even when there are many dependencies.

2.2 Semi-Formal Definition of Partitioning Algorithms

Input is a set U of elements. For a set $U' \subseteq U$, a *coreset* S is defined to be a representative sketch of U'. S is usually much smaller than U'.

The key property of a coreset is that there is an algorithm which, given access only to S, can compute a good solution for all of U'.

The generic algorithm is given as follows:

1. Let m be the number of machines.
2. Partition the points in U across the machines. Each machine receives a $|U|/m$ sized portion of the input (the partition is typically arbitrary, but needs to be randomized in some cases).
3. Machine i constructs a coreset S_i on the set it receives such that $|S_i|$ is much smaller than $|U|/m$.
4. The sets S_i are sent to a single machine and a final solution is constructed from $\cup_i S_i$.

There are a few key properties that we want with the algorithm. First, the coresets returned by the machines should allow for reconstruction of a good solution to the original problem. Critically, this holds even though each machine saw a sublinear portion of the input. Second, memory per machine is at least $\sum_{i\in[m]} |S_i|$, this guarantees that all the coresets returned can fit into the memory of a single a machine.

3 Application: Minimum Spanning Trees of Dense Graphs (MST)

We are given $G = (V, E)$ on $|V| = n$ vertices and $|E| = \ell$ edges, and edge weights $w_e \in \mathbb{R}^+$ for every $e \in E$.

We want to select a tree $T \subset E$ such that the graph $G = (V, T)$ is connected, and the total weight of T, $\sum_{e\in T} w_e$ is minimized. Such a tree is called a *minimum-weight spanning tree*. We will assume that ℓ is large compared to n, and machines have memory at least $\tilde{\Omega}(n)$, enough to store the entire output on a singe machine.

We want to use partitioning based approach. At first sight it may seem problematic to do arbitrary partitioning: intuitively, edges omitted in one partition may have been crucial for connectivity in another partition. However, we will show below that this is not the case for the MST problem. See the following subsection for description of a partition algorithm. The algorithm and analysis came from [**?**].

3.1 Partitioning Algorithm

1. Arbitrarily partition E into m parts, $E_1, E_2, \ldots, E_m$ with $|E_i| = \tilde{O}(\ell/m)$ for each $i \in [m]$.
2. On each partition, for each connected component of $G_i = (V, E_i)$, find the minimum cost spanning tree. Let T_i be the set of trees spanning each component of G_i.
3. Assign $\cup_i T_i$ to one machine. Denote $T_f = \cup_i T_i$, and $G_f = (V, E_f)$ be the resulting graph. Let T be the minimum spanning tree of G_f. Return T. T is the MST we want.

3.2 Algorithm Analysis

In order to prove this algorithm fits into the MPC setting, we need to prove two things: that this algorithm does return an MST for $G = (V, E)$, and that this algorithm doesn't run out of the memory on each machine, i.e., both the G_i's and the final graph G_f can satisfy the memory requirement.

Theorem 1 *The partition algorithm described returns a MST for $G(V, E)$.*

For our convenience, let's assume for now that all the w_e's are distinct. Before proving Theorem 1, we need the following lemma, which intuitively says for any i, all the edges thrown away in G_i are "unimportant" for G.

Lemma 2 *Let T be the MST for G. For any $e \notin \cup_i T_i$, $e \notin T$.*

Proof: Without loss of generality, assume $e \notin T_i$. Since T_i is a forest containing the collection of spanning trees for every component of $G_i = (V, E_i)$, adding e to T_i creates a cycle, call it C.

We claim that e has the maximum weight in C. If not, C contains some edge e' such that $w_{e'} > w_e$, but $w_{e'} \in T_i$. Then the collection of trees $T_i \cup \{e\} \setminus \{e'\}$ gives us another collection of spanning trees with smaller weight, contradicting the fact that T_i is MST for G_i.

Since this cycle C is in G_i, C also exists in G. We prove $e \notin T$ by contradiction. Suppose $e \in T$. Removing e from T partitions the vertices V into two sets V_1 and V_2, such that e belongs to the cutset $\delta(V_1, V_2)$. Since C is a cycle and $C \cap \delta(V_1, V_2) \neq \emptyset$, there must at least be another edge $e' \in C$ that is also in the cutset $\delta(V_1, V_2)$. We've already proved $w_e > w_{e'}$, so $T \cup \{e'\} \setminus \{e\}$ gives us a spanning tree with smaller total weights. This contradicts the assumption that T is MST. □
Applying Lemma 2 to G_f, we know that the MST in G can only contain edges in G_f, in other words, a MST in G_f is also the MST in G.

Now we've proved that the algorithm is correct, what remains is to show that it satisfies the memory requirement.

Lemma 3 *The graph G_f contains at most $m(n-1)$ edges.*

Proof: Each of the m partitions outputs a spanning forest on n nodes, which has at most $n-1$ edges. □

Theorem 4 *Say that n is the number of nodes in the graph and $\ell = n^{1+\beta}$ is the number of edges. Assume the algorithm is given machines of memory n^{1+c} per machine for any constant $0 < c < \beta$. There is a $O(\frac{\beta}{c})$ round MapReduce algorithm that computes a MST.*

Proof: Choose the number of machines to be $m = \frac{\ell}{n^{1+c}} = \frac{n^{1+\beta}}{n^{1+c}} = n^{\beta-c}$. The memory needed to store G_f is $O(n^{\beta-c} \cdot n) = O(n^{1+\beta-c})$, sublinar in the input size.

If $\beta - c > c$ then we can repeat β/c times until this holds. □

Note that to get $O(\frac{\beta}{c})$ rounds we need the memory to be superlinear in n. If it is n polylogarithmic this leads to a polylogarithmic round algorithm.

4 Approximation Algorithm Basics

Next we want to design approximation algorithms using the partition approach. First let's go over some basics for approximation algorithms.

Approximation Ratio: In this course we will be concerned with the approximation ratio of algorithms. The approximation ratio tells us how close our algorithms solution is compared to the optimal solution. Our goal is to find algorithms that are polynomial time and have the smallest approximation ratio possible.

The following is a list of terms often used in designing approximation algorithms.

- Objective function: The value of a feasible solution we want to optimize. It measures the quality of the solution. For example, in the MST problem the objective function is the summation of weights in the tree: $\sum_{e \in T} w_e$.
- $OPT(I)$: Given an instance I and an objective this is the value of the optimal solution.
- $A(I)$: Given an instance I, the value returned by our algorithm for the instance.
- α-approximation: For a maximization problem, if and only if $A(I) \geq \alpha OPT(I)$ for all I where $0 \leq \alpha \leq 1$; for a minimization problem, if and only if $A(I) \leq \alpha OPT(I)$ for all I where $\alpha \geq 1$.

The α-approximation Guarantees how far we are from optimal for every instance of a problem. This is a worst case bound, of inherit interest theoretically and is a good indicator for practice. Thus, we want to find an efficient algorithm (hopefully polynomial time) with the best α.

Pros and Cons of Approximation Algorithms:
Pros:

- Shows the varying difficulty of problems. The approximation ratio α can be very different from $O(1)$ to $O(n)$.
- The analysis can reveal easy and difficult cases for a problem.
- The ratio is robust for all instances.
- Allows for reduction between problems.
- Reveals underlying algorithmic challenges of the problems. It either helps to improve heuristics in practice, or can be used themselves.
- The theory reveals connections to areas of mathematics and cs.

Cons:

- Worst cases can focus on pathological instances. Ignores average case and practical instances.
- Typically no tradeoff of running time and quality of the solution.
- Limited to problems that have a clean formulation.
- Cannot be used for decision problems or those that are inapproximable (we will get into inapproximability later).

Approximation algorithms are often used to trade-off another resource. Usually this resource is time, but others are also possible. Here are some scenarios where approximation algorithms are commonly used.

1. Problems that take large polynomial running time to find optimal solutions. For example, matching takes too long for graphs like a social network. A simple greedy 1/2-approximation is known in linear time.

2. Big data problems which take a lot of space. For example, matching in the web graph. Usually we cannot store all the data. So one approach is to use stream algorithms.

3. NP-Hard Problems. To the best of our knowledge it will take exponential time to find the optimum.

4. Problems where it's expensive to code highly sophisticated algorithms. Say, maybe the programmer time is too high. In this case maybe a simple approximation will suffice.

5. Online problems. These are the problem where you do no know the entire input in advance. For example, consider job scheduling on multiple machines. We want to optimize some service quality. It is hard to solve if you don't know all the jobs in advance.

5 Application: An Approximation Algorithm for Traveling Salesman Problem (TSP)

In Traveling Salesman Problem, we are given a graph $G = (V, E)$, and cost $c(e)$ for all $e \in E$. The aim is to find a *Hamiltonian cycle* of minimum length. A Hamiltonian cycle is a cycle that visits each vertex exactly *once*.

This problem can be shown to be NP-Hard by being reduced from a Hamiltonian Cycle problem. In the Ham Cycle problem, you just want to know if a some general graph has a Hamiltonian Cycle. Thus, any solution to TSP is a solution to Hamiltonian Cycle problem. In fact, the situation is much worse. It is shown that TSP cannot be approximated within any polynomial factor.

Theorem 5 *The Traveling Salesperson Problem cannot be approximated within any polynomial factor unless $P = NP$.*

In order to solve TSP we need to relax the problem. We can either allow vertices to be visited multiple times or require a metric condition on G. Here we assume that the cost $c(e)$ is a metric and describe the *Metric-TSP* problem: in this problem, G is a complete graph, and $c(e)$ satisfies these conditions:

- $c(e) \geq 0$
- $c(uw) \leq c(uv) + c(vw)$, for any $u, v, w \in V$

Notice that a Hamilton cycle is close to a spanning tree, in the sense that if we throw away one edge in the Hamilton cycle, the rest gives us a spanning tree (in fact a path). Based on this intuition, finding the MST in the graph G might be close to finding the minimum weight Hamilton cycle. In fact, the optimal MST objective function value lower-bounds the optimal TSP, since the TSP cost is at least the cost of the path, which is at least the cost of MST. Now we design the following algorithm for Metric-TSP, called TSP-MST:

1. Compute a MST T of G.

2. Doubling edges of T, delete all other edges and call the resulting graph G'. By basic graph theory, since every vertex in this graph has even degree, G' contains an Euler tour. Note this is a tour of G since it passes all the vertices.

3. Obtain a Hamiltonian cycle by taking "shortcuts" in the tour: begin with arbitrary point and follow the Euler tour, and whenever the tour is going back to a point already visited before, take the shortcut from the current node to the next node that has not been visited so far. If all nodes are already visited, go back to the starting node and complete the Hamilton cycle.

Theorem 6 *TSP-MST is a 2-approximation for Metric-TSP.*

Proof: Obviously, TSP-MST returns a Hamilton cycle, since it's guaranteed to visit all vertices, and never visit the same vertex twice. The Hamilton cycle is reduced from an Euler cycle with cost $2\sum_{e\in T} c(e)$, and by triangle inequality, every "shortcut" can never make the total cost go up, so the resulting Hamilton cycle has a cost at most $2\sum_{e\in T} c(e)$. On the other hand, the MST is a lower bound on the optimal TSP cost, since the optimal TSP cost is bounded by the optimal MST cost, this is proved to be a 2-approximation. □

Knowing this, we can use the MST problem to obtain a $O(1)$ approximation for TSP, assuming the input is a dense graph.