PS-MWC: A SCALABLE, DISTRIBUTED MAXIMUM WEIGHT CLIQUE ALGORITHM USING APACHE SPARK

A Thesis Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Qian Qiu

August 2018

PS-MWC: A SCALABLE, DISTRIBUTED

MAXIMUM WEIGHT CLIQUE ALGORITHM

USING APACHE SPARK

_____

Qian Qiu

APPROVED:

_____

Dr. Christoph F. Eick, Chairman

_____

Dr. Larry Shi

_____

Dr. Zhu Han

_____

Dean, College of Natural Sciences and Mathematics

ACKNOWLEDGMENT

I would like to express my sincere gratitude to my advisor, Dr. Christoph F. Eick, for his guidance over the past year. I am grateful for everything he taught me and his time, support, and patience. I also would like to thank my parents and my husband for always encouraging and supporting me.

PS-MWC: A SCALABLE, DISTRIBUTED

MAXIMUM WEIGHT CLIQUE ALGORITHM

USING APACHE SPARK

An Abstract of a Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Qian Qiu

August 2018

ABSTRACT

The maximum weight clique problem (MWCP), in which we search for a clique with the maximum sum of vertices' weights, is a long-standing problem in graph theory. It has a wide range of applications in wireless network telecommunications, railroad dispatching, social-network analysis, and hotspot discovery.

The MWCP is a challenging NP-Hard problem; therefore, sequential algorithms are not able to solve it for graphs with millions of nodes and edges. To alleviate this problem, we propose a scalable and distributed approach, PS-MWC, to solve the MWCP utilizing Apache Spark, an in-memory cluster computing framework. PS-MWC consists of two main phases: a partitioning phase and a solving phase. In the partitioning phase, we partition the original problem into smaller sub-problems in parallel, such that each sub-problem is small enough to be efficiently solved using a sequential MWCP algorithm. In the solving phase, we solve the sub-problems and get candidate solutions in parallel. We apply efficient pruning strategies in both phases, to discard sub-problems if they have no chance to beat the best candidate solution found so far. Finally, we output the best solution among all candidate solutions.

We evaluated the performance of PS-MWC on a benchmark of MWCP. Experimental results showed that PS-MWC found the same solutions as state-of-the-art algorithms such as FastWClq and WLMC. We also examined the benefits of our partitioning strategy, the relationship between the number of vertices of graphs and the runtime of PS-MWC, as well as the pruning efficiency. The results showed that after partitioning, sizes of sub-problems were much less than that of the original graphs. Moreover, there was an almost linear relationship between sizes of original graphs and the runtime. High efficiency of the pruning strategy was reflected in the results that more than 99.80% of sub-problems had been pruned. For a graph with more than 3 million vertices and more than 117 million edges, PS-MWC took 1349 and 575 seconds on an EMR cluster with 64 and 128 cores, respectively.

CONTENTS

CHAPTER 1

INTRODUCTION

Clique problems are fundamental problems in graph theory. Given an undirected graph $G = (V, E)$, where V denotes the set of vertices and E denotes the set of edges, a clique C is a subset of V in which vertices are all pairwise adjacent in graph G. The maximum clique problem (MCP) is to find the maximum clique, a clique which contains the maximum number of vertices in graph G. The maximum weight clique problem (MWCP), is a generalization of the MCP. It assumes that there is a weight function w which assigns a weight to each vertex. The MWCP is to find a clique with maximum total weight in G. In the scenario that all weights of vertices are all equal, the MWCP is equivalent to the MCP.

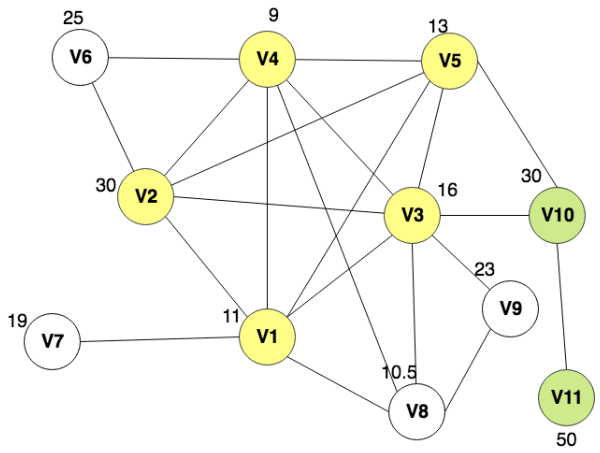

Figure 1.1. In the given weighted undirected graph G, the maximum clique is the set $\{v_1, v_2, v_3, v_4, v_5\}$, which is marked in yellow. The maximum weight clique in G is the set $\{v_{10}, v_{11}\}$, which is marked in green

Figure 1.1 gives an example of the MWCP and the MCP. In the undirected graph G, each circle denotes one vertex. Each vertex is assigned a unique id as an identification, and a weight, which is denoted by the number depicted next to it. Take v_1 for example, its id is 1, and its weight is 11. It turns out that the set of vertices $\{v_1, v_2, v_3, v_4, v_5\}$ constitutes a maximum clique in G, because we cannot find another clique which has more vertices than it in G. The maximum clique is

marked in yellow. When it comes to the MWCP, the set of vertices $\{v_{10}, v_{11}\}$, with total weight value 80, constitutes a maximum weight clique in G, because we cannot find another clique in G which has greater total weight than it. The maximum clique is marked in green in Figure 1.1.

The maximum weight clique problem is equivalent to the maximal weighted independent set problem (MWISP). An independent set in a graph $G = (V, E)$ is a set $I \subseteq V$ that there is no pair of nodes in I linked by an edge in E [1]. With this definition, we can easily conclude that an independent set in G is a clique in $\bar{G}$, which is the complement graph of G.

The maximum weight clique problem has a wide range of applications in many fields, such as wireless network telecommunications, railroad dispatching, and social-network analysis. For example, in the case of wireless networks, it can be used for node clustering, by finding a maximal-weighted independent set which can act as cluster-heads [1]. Furthermore, it can be used for scheduling in wireless networks, by finding an independent set of nodes which can transmit maximal amount of data with no interference between arbitrary pairs of nodes [2]. In railroad dispatching, it can be used to find a maximum set of non-conflicting train routes [3]. It can also be used to remove overlapping hotspots in Hotspot Discovery Frameworks [22, 23].

The MCP and MWCP are NP-Hard problems, which means we don't know of polynomial-time algorithms for them. Due to its wide applications, many researchers have investigated MCP and tried to solve it within an acceptable time. However, compared with MCP, fewer algorithms have been proposed for solving MWCP. The reason might be that MWCP is more complicated and some powerful techniques for MCP are not applicable or ineffective for solving MWCP since the weights of vertices need to be considered [6].

In recent years, the proliferation of massive data sets establishes the need for scalable algorithms to solve MWCP for massive graph data. Furthermore, several frameworks like Hadoop MapReduce and Apache Spark can facilitate distributed and scalable computation on computing clusters. However, quite few distributed or parallel algorithm for the MWCP have been proposed.

To the best of our knowledge, only one recent paper [16] proposes a parallel tabu search algorithm for MWCP. This tabu search algorithm has a limitation that it doesn't guarantee to find the best result. Moreover, while exploring the search space, tabu search spends a serious amount of time calculating the fitness value of each neighbor solution, so it is more suitable for dense graphs with a small search space. The maximum number of vertices in its benchmark instances is only 4000. In this research, we propose a parallel and scalable algorithm, PS-MWC, which aims to find the maximum weight clique in massive graphs on a cluster. This algorithm is implemented using the Apache Spark framework [28], a high-performance fault-tolerant in-memory computing framework. Particularly, we present an algorithm which partitions the input graph into sub-problems in parallel. The sub-problems have smaller sizes than the original graph, and are independent with each other, so they can be distributed among the nodes in a cluster and be solved separately. Afterwards, the sub-problems are solved by any existing sequential algorithms for the MWCP. During the partitioning and solving phases, we employ an effective pruning strategy that discards sub-problems when they cannot contain a better clique than the best solution found so far. In this thesis, we choose FastWClq [6] as the sequential algorithm in the solving phase.

The main ideas of PS-MWC include:

1. Initializing the value of $global\_max\_weight$, a global variable which is used to keep track of the weight of the best clique we find so far, with the weight of a clique we rapidly get from the input graph;
2. Partitioning the input graph into sub-problems, which can be solved independently. The maximum weight clique of the original graph is contained in at least one of these sub-problems;
3. Solving sub-problems to get candidate solutions in parallel and updating $global\_max\_weight$ when a better solution is found;

4. Pruning sub-problems with $global\_max\_weight$ during the partitioning step as well as the solving step;

Compared with the parallel tabu search algorithm [16], PS-MWC can adopt any exact algorithm for the MWCP in the solving step. This guarantees the optimality of the solution obtained. Moreover, PS-MWC is capable of solving massive sparse graphs with millions of vertices. In contrast to the partitioning strategies used in parallel algorithms for MCP [20, 21], partitioning in PS-MWC is done in parallel by each vertex sending messages to its neighbor and merging messages it received. The partitioning procedure is implemented with functions provided by GraphX library in Apache Spark.

Our implementation uses the Apache ZooKeeper service to maintain and update the value of $global\_max\_weight$ in a znode that can be accessed by each executor. Moreover, we improve the efficiency of our program according to Spark Tuning Guide [24], by tuning the data serialization strategy, choosing optimal data structures, and tuning the garbage collection strategy. Furthermore, we conducted several experiments to evaluate the solution quality, partitioning effect, runtime difference between different strategies, the pruning efficiency, and the scalability. We evaluated the performances of PS-MWC on a benchmark of MWCPs. Experiments showed that PS-MWC found the same solutions as FastWClq [6] and WLMC [5]. We also examined the benefits of our partitioning strategy, the relationship between the number of vertices of graphs and the runtime of PS-MWC, as well as the pruning efficiency. The results showed that after partitioning, the sizes of sub-problems were much less than that of the original graph. Moreover, there was an almost linear relationship between the size of original graph and the runtime of PS-MWC. High efficiency of the pruning strategy was reflected in the results that more than 99.80% of sub-problems had been pruned. For a graph with more than 3 million vertices and more than 117 million edges, PS-MWC took 1349 and 575 seconds on an EMR cluster with 64 and 128 cores, respectively.

This thesis is organized as follows. Chapter 2 introduces the background knowledge. Then, we explain our distributed algorithm PS-MWC in detail in Chapter 3. Chapter 4 describes the implementation of PS-MWC as well as the tuning techniques for improving its efficiency. Experimental evaluations of PS-MWC are presented in Chapter 5. Finally, we provide concluding remarks and discuss future work in Chapter 6.

CHAPTER 2

BACKGROUND

In this section, we briefly discuss related work first. Then, we present some background knowledge of the techniques which have been used for implementing PS-MWC: Apache Spark, GraphX library, and the Apache ZooKeeper service. Section 2.3 provides the scenario of applying PS-MWC in a hotspot discovery framework.

2.1 Related Work

The main algorithms for MWCP fall into two types: exact algorithms which guarantee the optimality of solutions, and heuristic algorithms which aim to find (near-)optimal solutions when the search space is too large.

Östergård [11] proposed a branch and bound MWCP algorithm based on error-correcting codes. He also developed an improved branch and bound algorithm [12] in which the vertices were processed based on an order provided by a vertex coloring of the input graph. Yamaguchi et al., [13] presented an algorithm which computes the upper bound of the MWCP based on the longest path in a directed acyclic graph generated by the input graph. Fang et al., [7] developed an algorithm based on Maximum Satisfiability Reasoning to improve the upper bound, and applied Top-k failed literal detection to improve the upper bound. Because all previous exact algorithms for MWCP exhibited slow performances on large graphs, Jiang et al. [5] proposed an exact branch and bound algorithm, called WLMC, which preprocesses the input graph data to derive a vertex ordering, removes vertices which cannot be contained in any optimal solution, and reduces the number of branches in the search space by incremental vertex-weight splitting.

As for heuristic algorithms, Wang et al. [4] introduced two heuristics and develop a local search algorithm called LSCC+BMS for MWCP. BMS is a probabilistic heuristic called Best from Multiple Selection, which returns the best element from multiple samples. Based on the same heuristic, Cai et al., [6] proposed an algorithm called FastWClq for solving massive graphs,

which interleaves between clique construction and graph reduction. Moreover, they proposed a heuristic function for estimating the benefit of adding a vertex, and a graph reduction algorithm. Kumlander [8, 9] proposed a backtrack tree search algorithm in which the vertices are ordered using a heuristic coloring method. Shimizu improved Kumlander's algorithm and developed VCTable [14] and OT-Clique [15] algorithms which adopt the optimal table method.

The above proposed approaches for MWCP are all sequential, which means they are not scalable. For all these approaches except WLMC [5], LSCC+BMS [4], and FastWClq [6], when the size of input graph increases, the computation cannot be completed within reasonable time. Even though WLMC [5], LSCC+BMS [4], and FastWClq [6] were designed with purpose of solving MWCP for large graphs, they cannot deal with massive graphs which are distributed in multiple nodes, since they can work only on single computer. This calls for scalable and distributed algorithms for solving MWCP for massive graphs. Recently, Kiziloz and Dokeroglu [16] developed a parallel tabu search algorithm for MWCP. This tabu search algorithm has a limitation that it doesn't guarantee to find the best result. Moreover, while exploring the search space, tabu search spends a serious amount of time calculating the fitness value of each neighbor solution, so it is more suitable for dense graphs with a small search space. The maximum number of vertices in the benchmarks they use was only 4000.

While there are few scalable algorithms for MWCP, some researchers have proposed algorithms for solving MCP on a cluster. Pardolos et al. [17] introduced a parallel maximum clique algorithm based on MPI. Because MPI does not provide fault tolerance, several researchers proposed MapReduce solutions for MCP. Lin et al. [18] proposed a MapReduce algorithm for MCP, which adopts a random partitioning strategy. Xiang et al. [19] also proposed a MapReduce algorithm, which partitions a graph into subgraphs based upon graph coloring and uses a MCR algorithm developed by Tomita et al. [20]. However, the MapReduce model has several drawbacks, e.g., it needs to save intermediate outputs data to the disk after computation, which

results in possibly unnecessary I/O operations. More specifically, the results of *map()* function are written to disk as intermediate outputs, which are then read and processed by *reduce()* function, and the final outputs are stored to disk. To avoid those limitations, Elmasry et al. [21] proposed a scalable algorithm for MCP using Apache Spark to distribute the computational load. They designed a multi-phase partitioning strategy, which requires ordering all of vertices by degree. Furthermore, the partitioning was done by sequentially removing vertices from the input graph. This algorithm was designed for dense graphs, that contain thousands of vertices and a few million edges. It is not suitable for sparse real-world graphs that contain millions of vertices.

2.2 Technology

2.2.1 Apache Spark

Apache Spark is a popular open-source cluster computing framework featuring SQL query, graph processing, and machine learning. One important advantage of Spark is it provides in-memory computing. It can load input data into memory which can be accessed by multiple processes for multiple times; hence, unnecessary disk I/O overhead is avoided. This advantage makes Spark more suited for iterative applications than MapReduce, which needs to load data from and write data to disk for each iteration. It also has high usability, due to different programming language APIs: Python, Scala, R, and Java. Moreover, Spark supports fault tolerance computation.

Every Spark application consists of a driver program. A driver program runs *main* function implemented by users and executes parallel operations on a cluster [28]. Running on master node, it connects to a cluster manager for distributing resources, splits the application into tasks, and schedules them to run on executors. An executor is a distributed agent responsible for executing tasks on worker nodes [29]. A task is the smallest individual unit of execution launched to compute a RDD partition [29].

A fundamental processing unit in Spark is Resilient Distributed Datasets (RDD), an immutable distributed dataset across the cluster which is resilient to data storage failure. RDD is partitioned

across the compute nodes of the cluster, so it can be operated on in parallel. There are two kinds of operations on RDD: transformations and actions. With transformations, we can create a new RDD from an existing one. Transformations are lazy, which means they are only computed when an action requires a result [28]. Actions return a value to the driver program or write data to external storage after running a computation on the RDD.

The driver program defines RDDs and invokes actions on them. Spark code on the driver also tracks the RDDs' lineage [27]. RDD lineage, which is also called RDD dependency graph or RDD operator graph, is a graph of all the parent RDDs of a RDD. It is built as the result of applying transformations to the RDD [29]. When a RDD partition is lost, it can be re-built based on the RDD's lineage. In this way, RDDs can automatically recover from node failures.

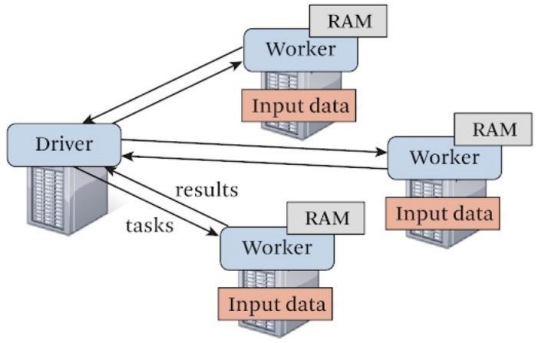

Figure 2.1. Spark runtime. The driver program launches multiple workers, which read data blocks from a distributed file system and persist computed RDD partitions in memory [27].

Figure 2.1 provides a work flow graph at Spark runtime [27]. The driver program coordinates multiple workers and the overall execution of tasks. When the driver loads input data, RDDs are created and distributed among workers. Based on the data placement, the driver can assign workers tasks to run transformations on these RDDs to generate new forms of RDDs and perform actions to collect or store results. Executors running on each worker node execute the assigned tasks by performing in-memory computation. During the process, RDDs can be cached in memory on the worker nodes to speed up data processing.

By default, when Spark runs a function in parallel as a set of tasks on different worker nodes, a copy of each variable used in the function is sent with each task. In this way, updating variables in the function on worker nodes will not change their value in the driver program. Sometimes, we may need to update a variable on each worker nodes and get the updated value in driver. Moreover, a variable might need to be shared across tasks, or between tasks and driver. To meet these needs, Spark provides two types of shared variables: broadcast variables and accumulators. Broadcast variables allow the programmer to keep a read-only variable cached on each node rather than ship a copy of it with tasks. Figure 2.2 shows the process of broadcasting a value to the executors. The driver creates a broadcast variable from variable *m*, and broadcasts it to all worker nodes. Worker nodes cache this broadcast variable in memory, which can be accessed by multiple executors running on the nodes by calling the *value* method.

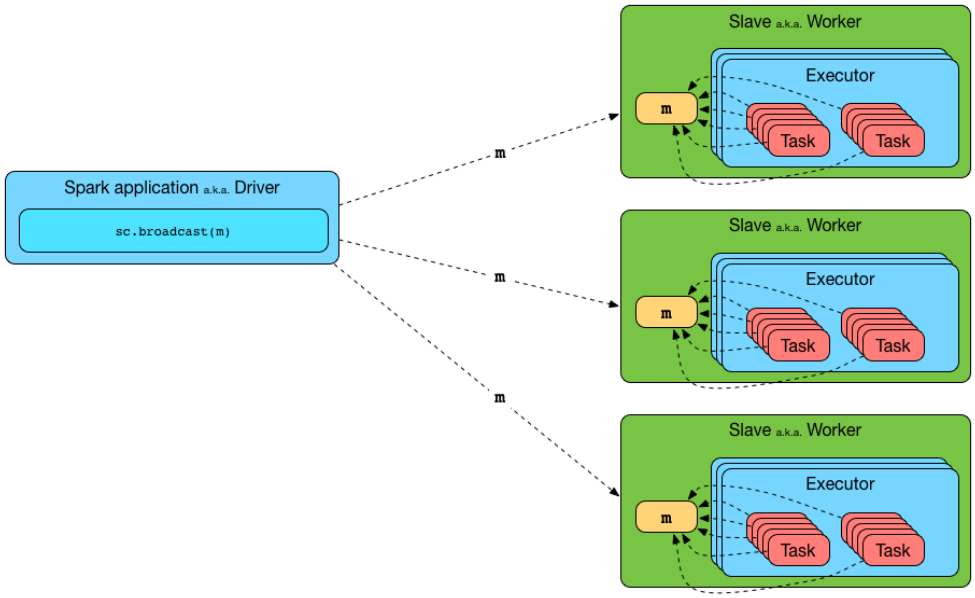

Figure 2.2. The driver program broadcasts a variable to workers. Multiple executors running on the same worker can access this variable [30]

Accumulators provide parallel implementations for sums and counters. While accumulators of numeric types are provided by Spark natively, users can define their own types of accumulators by extending the *AccumulatorV2* abstract class.

2.2.2 GraphX

GraphX library, provided in Apache Spark for graphs and graph-parallel computation, makes it easier to build applications that exploit graph analytics. GraphX introduces a graph abstraction called property graph, a directed graph with properties attached to each vertex and edge, and provides a set of fundamental operators for graph computation. Moreover, GraphX provides multiple graph algorithms and builders to facilitate graph analytics and computation.

Property graphs, which are implemented based on RDD abstraction, are immutable, distributed, and fault-tolerant. They can also be cached in memory to avoid recomputation. In property graphs, each vertex is identified by a *VertexID*, a unique 64-bit long identifier. Similarly, each edge is identified by corresponding source and destination *VertexIDs* [34]. The properties are stored as objects with each vertex and edge in the graph.

In graph-parallel computation, graphs need to be partitioned across machines. There are two main approaches for distributed graph partitioning: vertex-cut and edge-cut. GraphX adopts the vertex-cut approach, which assigns edges to distinct machines and allowing vertices to span multiple machines [31]. GraphX provides several vertex-cut partition strategies, such as EdgePartition1D, EdgeParition2D, CanonicalRandomVertexCut, and RandomVertexCut. EdgePartition1D strategy ensures that all edges with the same source are partitioned together. EadgePartition2D, CanonicalRandomVertexCut, and RandomVertexCut use both source vertices and destination vertices to calculate partitions. The difference is that the latter two uses *hashCode()* in partitioning. In RandomVertexCut, the edge that connects the same pair of nodes may be spread among two executors based on the direction of the edge. In contrast, CanonicalRandomVertexCut partitions the edges regardless to the direction, so the edges sharing both a source and a destination will be partitioned together.

As is shown in Figure 2.3, GraphX provides three views of the property graph: vertex view, edge view, and triplet view. With the vertex view, users can access an RDD which contains vertices of

the graph. Similarly, with edge view, the edges of the graph can be extracted. The triplet view provides a RDD by logically joining the vertex and edge properties. A triplet contains an edge with its property, as well as the properties of source vertex and destination vertex.

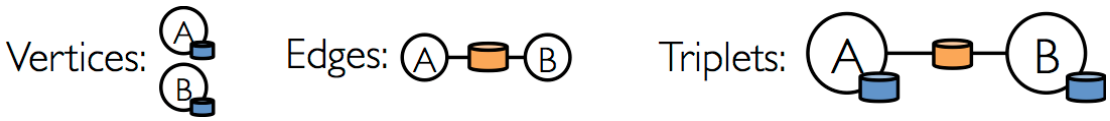

Figure 2.3. Three views of the property graph in GraphX

There are many successful practices for massive graph computation using Spark GraphX. For instance, the Tencent company utilizes Spark GraphX to compute common friends between each pair of its billions of users. With the benefits of Spark GraphX, the computation can be completed by a cluster within two and a half hours, which is a huge improvement compared to two days with a previous approach. In their experience, for massive graph computation, the size of executor memory should be large. One reason is that GraphX adopts vertex-cut partitioning method, i.e. putting vertex and its neighbors on one executor to reduce communication between executors. For executors with small memory size, suppose we need 10 executors to hold all neighbors of one certain vertex, this vertex should be replicated 10 times. For executors with large memory size, one executor can hold all its neighbors. It only needs to be replicated once theoretically, which would greatly reduce space cost. Another reason is that sometimes the vertices' sizes grow while merging and aggregating messages. In this case, if the memory of executors is not big enough, GraphX would do lots of shuffle and spill operations to accommodate all of data, thus making computation slower.

2.2.3 Apache ZooKeeper

Introduced in Section 2.2.1, Spark provides two types of shared variables: broadcast variables and accumulators. However, they both have limits. Broadcast variables are read-only, which means executors cannot update their values. Value of accumulators can be updated by executors, but

they can only be “added” to. Furthermore, tasks running on executors cannot read accumulators’ value; only the driver can read their value.

In PS-MWC, we need to maintain a global mutable variable, which contains the weight of the best solution found so far. It should be accessed by all executors, with the purpose of pruning the search space and be updated by executors when better solution is found. The two shared variables provided by Spark cannot meet these requirements, because broadcast variables are read-only and accumulators’ value cannot be accessed by executors.

Apache ZooKeeper, an open-source coordination service for distributed applications, is suitable for maintaining a global mutable variable which can be accessed by executors. It facilitates implementing synchronization, configuration maintenance, and naming of distributed applications [33].

With a shared hierarchal namespace, ZooKeeper enables distributed processes to coordinate with each other. The namespace consists of znodes, i.e. ZooKeeper data nodes, each of which can have data associated with it. Through the ZooKeeper API, distributed processes can connect to a ZooKeeper server, set watches on znodes, send requests, get responses, and get watch events. After setting a watch on a znode, a distributed process will receive a notification when the znode has been changed.

By keeping data in memory, ZooKeeper can achieve high throughput and low latency. It is especially fast in “read-dominant” workloads. The documentation of ZooKeeper [33] claims that it performs better where reads are more common than writes, at ratios of around 10:1. In PS-MWC, reads on the global variable are more common than writes, since writes only happen when a better solution is found. ZooKeeper is well suited for PS-MWC. In Section 5.2, experiments show that the global variable maintained using ZooKeeper greatly improves the efficiency.

2.3 Hotspot Discovery Framework

As the size of spatial data grows significantly, this trend calls for an efficient spatial knowledge discovery framework. Akdag and Eick [22, 23] propose a novel methodology for discovering interestingness hotspots in spatial datasets. The proposed framework discovers hotspots which maximize a user-defined interestingness function in five steps. At first, it identifies neighboring objects in the dataset and generates hotspot seeds based on them. Then, hotspots are grown from identified hotspot seeds. Afterwards, it finds an optimal set of hotspots by removing highly overlapping hotspots and finally computes the scope of result hotspots.

Removing highly overlapping hotspots can be formed as a MWCP, so PS-MWC can be used to find an optimal set of hotspots in this framework.

In this application, the inputs are a set of hotspots and an overlap threshold λ, where $0 \leq \lambda \leq 1$. The output is a set of high-quality hotspots with a low degree of overlap. We can calculate an optimal set of hotspots with the following steps:

- Calculate reward value for each hotspot as its weight;
- Create a weighted overlap graph of hotspots in which each vertex is a hotspot and there is an edge between two vertices if their degree of overlap is greater than the overlap threshold λ;
- Use PS-MWC to find the maximum weight clique (MWC) in the complemental graph of the weighted overlap graph;
- Output the union of all vertices in the maximum weight clique as the optimal set of hotspots.

CHAPTER 3

METHODOLOGY

In this chapter, we propose a distributed and scalable algorithm for solving maximum weight clique problem called PS-MWC. We introduce the necessary notations and definitions used in our algorithm. Then, we describe the entire process of PS-MWC and introduce the main steps in detail.

3.1 Notations and Definitions

Table 3.1 gives a list of commonly used symbols in this thesis.

Table 3.1. Table of Notations

Symbol	Meaning
$\boldsymbol{V}$	A set of vertices
$\boldsymbol{E}$	A set of edges
$\boldsymbol{G = (V, E)}$	An undirected graph
$\boldsymbol{\lvert V \rvert}$	The number of vertices in V
$\boldsymbol{\lvert E \rvert}$	The number of edges in E
$\boldsymbol{v}$	A vertex in V
$\boldsymbol{id(v)}$	The unique vertex id assigned on vertex $v \in V$
$\boldsymbol{e = (v, v')}$	An edge e in E
$\boldsymbol{Neighbors(v)}$	The set of neighbors of v: $\{v' \mid (v, v') \in E, v, v' \in V\}$
$\boldsymbol{d(v)}$	the degree of v, i.e. the number of neighbors of v
$\boldsymbol{w(v)}$	The weight of v in V
$\boldsymbol{G = (V, E, w)}$	An undirected weighted graph
$\boldsymbol{w(V')}$	The weight of a set of vertices V'
$\boldsymbol{v' >_d v}$	v' has higher-order than v
$\boldsymbol{H(v)}$	Higher-order neighbors of v
$\boldsymbol{G(S) = (S, E')}$	The subgraph induced by S from G, while $S \subseteq V$ and $E' = \{(v, v') \mid (v, v') \in E \wedge v \in S \wedge v' \in S\}$.
$\boldsymbol{C}$	A clique in the graph G
$\boldsymbol{C'}$	Maximum weight clique in the graph G

Let $G = (V, E)$ denotes an undirected graph, where V is the set of vertices and E is the set of edges of G.

Let $|V|$ and $|E|$ denotes the numbers of vertices and edges of G, respectively.

Each vertex $v_i \in V$ is assigned a unique vertex id i, $i \in \{1, 2, \dots, |V|\}$.

Each edge $e \in E$ is represented using two vertices (v, v'), which both belong to V.

v and v' are said to be neighbors, i.e. adjacent, if $v \in V, v' \in V$ and $(v, v') \in E$.

We define the set of adjacent vertices of v as: $Neighbors(v) = \{v' | (v, v') \in E, v, v' \in V\}$.

Let $d(v)$denotes the degree of v, i.e. the number of neighbors of v.

Let $G = (V, E, w)$ denote an undirected weighted graph, with each vertex assigned a weight value.

Let $w(v)$ denote the weight value of v in V.

The weight of a set of vertices V' is the sum of weights of all vertices in V': $w(V') = \sum_{v \in V'} w(v)$.

We assign an order to the vertices of G as such: $v' >_d v$ if and only if $d(v') > d(v)\ or\ d(v') = = d(v)\ and\ id(v') > id(v)$.

We further define higher-order neighbors of v as below:

$$H(v) = \{v' \mid v' \in Neighbors(v), v' >_d v\}$$

A graph $G' = (V', E')$ is considered as a subgraph of a graph $G = (V, E)$, if and only if $V' \subseteq V$ and $E' \subseteq E$.

A clique C in the graph $G = (V, E)$ is a subset of V such that all vertices in C are pairwise adjacent in G, i.e. $\forall v \in C, \forall v' \in C, (v, v') \in E\ holds$.

In our algorithm, a solution is represented by $(C, w(C))$.

Maximum weight clique C' in the graph $G = (V, E, w)$ is a clique with the maximum weight value among all cliques of G.

3.2 Overview of PS-MWC

In this section, we propose an algorithm called PS-MWC for solving maximal weight clique problems. We first describe the overall process of the algorithm, and then introduce its main steps in details.

The input of this algorithm is an undirected weighted graph G which is stored in two files: a vertices file and an edges file. Each line of vertices file is a pair of numbers which denote a vertex id and its weight. Each line of edges file is a pair of numbers which denote the source vertex id and destination vertex id of an edge in G.

The output of this algorithm is a tuple that consists of the maximum weight clique we find in the input graph and its weight: $(C', w(C'))$, where C' denotes the maximum weight clique.

The main idea of PS-MWC is to partition the graph into several sub-problems, solve each sub-problem in parallel, and get the result. The main steps of PS-MWC are shown in Figure 3.1.

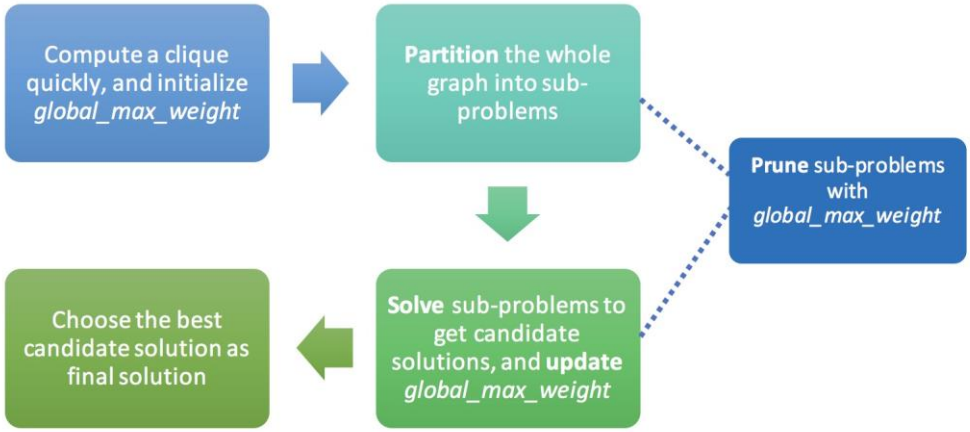

Figure 3.1. PS-MWC workflow diagram

During overall process, we maintain the weight of the best clique found so far in a variable $global\_max\_weight$, which could greatly prune sub-problems in the solution step.

At first, we try to compute a clique in a quick way, and initialize $global\_max\_weight$ with its weight.

For the purpose of balancing the sizes of sub-problems and reducing overlap, we assign an order to vertices in the graph and partition the whole graph into sub-problems in parallel.

In the following solving phase, we implement $FastWClq$ algorithm in [6] to solve sub-problems and get candidate solutions on each executor. Once a better solution is found, $global\_max\_weight$ is updated to be its weight value.

We prune sub-problems with $global\_max\_weight$ in both partitioning and solving phases, by discarding sub-problems which cannot beat the best clique found so far.

In the end, we choose the best candidate solution and output it as final solution.

In this section, we discuss the overall process of PS-MWC, and then provide a detailed explanation of its main steps in following sub-sections.

3.2.1 Overall Process of PS-MWC

PS-MWC constructs a graph from input files, quickly gets a clique, divides the graph and solves sub-problems on executors. Finally, PS-MWC outputs C' and $w(C')$ as results. C' denotes the maximum weight clique, while $w(C')$ denotes the weight of this clique.

Algorithm 1 gives the pseudo code of the overall process of our algorithm.

In step 2, PS-MWC constructs graph G from input files;

In step 3, PS-MWC calls $getQuickMWCL(G)$ function to obtain a clique of G, and initializes C' and $w(C')$ to be its vertices and weight;

In step 5, PS-MWC initializes the global best clique weight value $global\_max\_weight$ with value of $w(C')$. The variable can be read and updated by the driver and executors. Once an executor computes a higher clique weight, it updates $global\_max\_weight$ with the new value. This global value is a great help in pruning sub-problems.

In step 6, we generate independent sub-problems on graph G by calling *partition()* function.

For each sub-problem, we apply $FastWClq()$ function on it in step 8. $FastWClq()$ function, runs on executors, can solve each sub-problem, as well as update value of $global\_max\_weight$. Once the results of sub-problems are returned by executors, we select the best one among the candidate solutions and output the result.

ALGORITHM 1. Overall Process of PS-MWC

1: Procedure MWCP (vertices_file, edges_file)
   Input: vertices_file which contains the id and weight of each vertex, edges_file which contains the source vertex id and destination vertex id of each edge
   Output: $(C', w(C'))$
2: construct a vertex weighted graph $G = (V, E, w)$
3: set $C', w(C') = getQuickMWCL(G)$
4: set $results = \{(C', w(C'))\}$
5: set $global\_max\_weight = w(C')$
6: set $subProblems = partition(G, w(C'))$
7: foreach $subProblem\ in\ subProblems$ do:
8: set $results = results \cup (FastWClq(subProblem))$
9: end foreach
10: set $C', w(C')$ to be the best solution stored in $results$
11: return $(C', w(C'))$
12: end Procedure

3.2.2 Quickly Get a Clique

Algorithm 2 gives the pseudo code of the algorithm for getting a clique of $G = (V, E, w)$.

We use C to denote the clique we are constructing. In step 2, we initialize C to be an empty set.

We use $CandSet$ to denote the set of all vertices which can be added into C, i.e. each vertex in $CandSet$ is adjacent to all vertices in C. In step 3, we initialize $CandSet$ to be all vertices in the $workingGraph$.

To quickly get a clique for massive graph G, in step 3, we initialize $CandSet$ to be a subset of V with r vertices, where $r \ll |V|$, and then get a clique C in this subset. In our algorithm, we use r = 500. To be specific, we take a subset of V in which vertices' ids are smaller than 500 in G. In this way, the size of $CandSet$ is much less than V; then, we can get a clique quickly.

In step 5 to 10, the clique C is constructed by keep removing the vertex with highest weight and degree in $CandSet$ until $CandSet$ becomes empty.

ALGORITHM 2. Quickly Get a Clique

1: Procedure GetQuickMwcl (G)
   Input: a vertex weighted graph $G = (V, E, w)$
   Output: $(C, w(C))$
2: set $C = \emptyset, w(C) = 0$
3: set $CandSet$ = a subset of V

4: while $CandSet \neq \emptyset$ do:
5: set v = remove the vertex with highest weight and degree in $CandSet$
6: set $C = C \cup \{v\}$
7: set $w(C) = w(C) + w(v)$
8: set $CandSet = CandSet \cap Neighbors(v)$
9: end while
10: return $(C, w(C))$
11: end Procedure

Now, let us describe the method of construction with an example. Figure 3.2 gives a weighted undirected graph $G = (V, E, w)$. At first, we set C to be $\emptyset, w(C)$ to be 0. Since the number of vertices in V is few, we set $CandSet$ to be all vertices in V, for simplicity. Among the $CandSet = \{v_1, v_2, v_3, v_4, v_5, v_6\}$, v_6 has the maximum weight. Thus, we remove v_6 from $CandSet$, add v_6 into C, and add weight of v_6 to $w(C)$. Afterwards, C is $\{v_6\}$, and $w(C) = 50$. Then, we update $CandSet$ to be the intersection of previous $CandSet$ and $Neighbors(v_6)$, and the result $CandSet$ is $\{v_4\}$. Since $CandSet$ only has one vertex, we remove v_4 from $CandSet$, add v_4 into C, and add weight of v_4 to $w(C)$. Afterwards, C is $\{v_6, v_4\}$, and $w(C) = 50 + 9 = 59$. Afterwards, $CandSet$ becomes empty, so the construction stops.

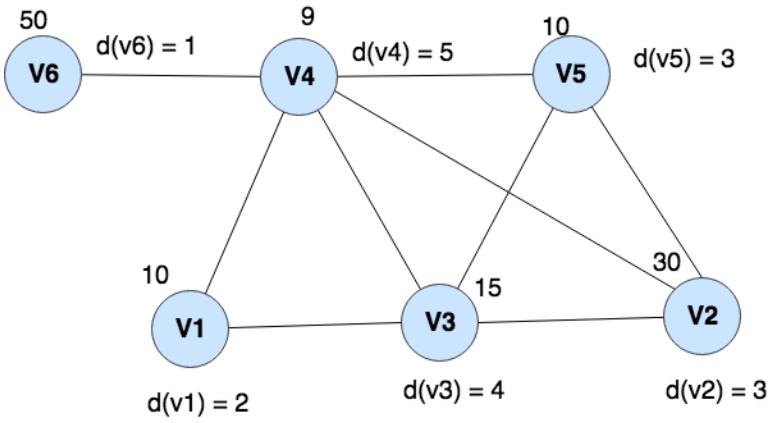

Figure 3.2. A weighted undirected graph G=(V,E,w)

3.2.3 Generate Sub-problems

In this step, we partition the input graph into sub-problems so that each executor can process a subset of sub-problems independently and in parallel. The maximum weight clique of input graph

should be contained in at least one sub-problem generated in this step. In this way, after solving sub-problems and getting their maximum weight cliques, which we call *candidate solutions*, we can get final solution by choosing the candidate solution with highest weight.

Algorithm 3 gives the pseudo code of the algorithm for generating sub-problems.

ALGORITHM 3. Generate Sub-Problems

1: Procedure Partitioning (G, $global\_max\_weight$)
   Input: a vertex weighted graph $G = (V, E, w)$, $global\_max\_weight$
   Output: $subProblems$
2: calculate degree for each vertex in $V(G)$
3: each vertex sends its id as message to its neighbors which have lower degree (or have the same degree and lower id) than it
4: set $subProblems = \emptyset$
5: foreach v in $V(G)$ do:
6: set $H(v) = \{v' \mid v' \in Neighbors(v), v' >_d v\}$
7: set $subProblem = (\{v\}, G(H(v)))$
8: set $upperBound(subProblem) = w(v) + w(H(v))$
9: if $upperBound(subProblem) > global\_max\_weight$ then:
10: set $subProblems = subProblems \cup subProblem$
11: end if
12: end foreach
13: return $subProblems$
14: end Procedure

At first, we assign an order to the vertices of G as such: $v' >_d v$ if and only if $d(v') > d(v)$ or $d(v') == d(v)$ and $id(v') > id(v)$. We further define $H(v)$ as the set of higher-order neighbors of v: $H(v) = \{v' \mid v' \in Neighbors(v), v' >_d v\}$. We use this ordering because it achieves a balanced workload which is interpreted in paper [21]. We also compare this ordering with other ordering in experiments, which show that this ordering has a better performance.

Instead of sorting all vertices before partitioning, which is adopted in [21], we get higher-order neighbors of each vertex in parallel. To be specific, in step 3, we let each vertex send its id as a message to its neighbors which have lower degree (or have the same degree and lower id) than it. In this way, when we merge messages received by each vertex, we can get a set of ids of its higher-order neighbors. Figure 3.3 shows the directions of messages sending on G.

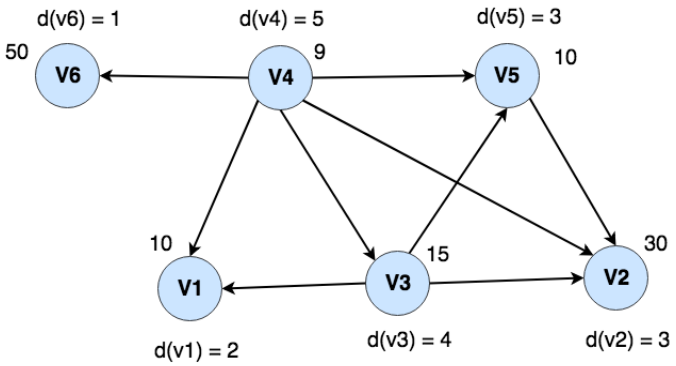

Figure 3.3. The directions of messages which are sent on the given graph G=(V, E, w) while getting higher-order neighbors

In Figure 3.3, because v_4 has the maximum degree among all vertices in V, v_4 sends its id to all neighbors. For v_5, its neighbors are $\{v_2, v_3, v_4\}$, among which v_3 and v_4 have higher degree than v_5, v_2 has the same degree but lower id than v_5; then, after merging the messages v_5 receives, we get $H(v_5)$: $\{v_3, v_4\}$.

In step 5 to 12, we generate sub-problems for each vertex by choosing this vertex as a partitioning path, and put the subgraph which is induced by higher-order neighbors of this vertex into a sub-problem. Each sub-problem contains two parts: a partitioning path, and an induced subgraph. Partitioning path indicates how the subgraph is obtained from the original graph [21]. For example, the sub-problem: $(\{v\}, subgraph = G(H(v)))$ is obtained by the partitioning algorithm via v. When we calculate a maximum weight clique C' for the $subgraph$, we know a maximum weight clique C of this sub-problem is: $(\{C' \cup v\}, w(C') + w(v))$.

After generating sub-problems for all vertices in G, we can compute candidate solutions, i.e. maximum weight cliques of all sub-problems, independently and in parallel. The solution of original graph is the clique with the largest weight among the candidate solutions.

We can prove that the maximum weight clique C' of input graph G is contained in at least one sub-problem generated in this way. Suppose v is the lowest-order vertex in C', such that for all

$v' \in C', v' \geq_d v$ holds. It is obvious that all vertices in C' except v, denoted by $C' - \{v\}$, must be a subset of $H(v)$. Thus, C' must be contained in the sub-problem: $(\{v\}, subgraph = G(H(v)))$. To compute efficiently, we use a branch and bound approach to avoid computing the maximum weight clique of a sub-problem if it can be shown that it cannot produce a better solution than the best solution found so far [6]. Specifically, we define $upperBound(subProblem) = w(v) + w(H(v))$, and discard sub-problems which has $upperBound$ less or equal to $global\_max\_weight$.

Figure 3.4 presents an example to illustrate the partitioning phase. In this figure, each circle denotes a vertex with its id, and the digit besides each circle denotes its weight. In this example, we use Algorithm 2 to get a clique $C = \{v_4, v_6\}, w(C) = 59$, and set $global\_max\_weight$ to be 59. Next, we generate sub-problems for each vertex. Take v_5 for example, its neighbors are $\{v_2, v_3, v_4\}$, in which v_3, v_4 have higher order than v_5, so the sub-problem of v_5 is $(\{v_5\}, G(\{v_3, v_4\}))$. Moreover, we compute $upperBound$ for each sub-problem and compare it with $global\_max\_weight$. All sub-problems are discarded except the sub-problem of v_2, which is the output of partitioning of G.

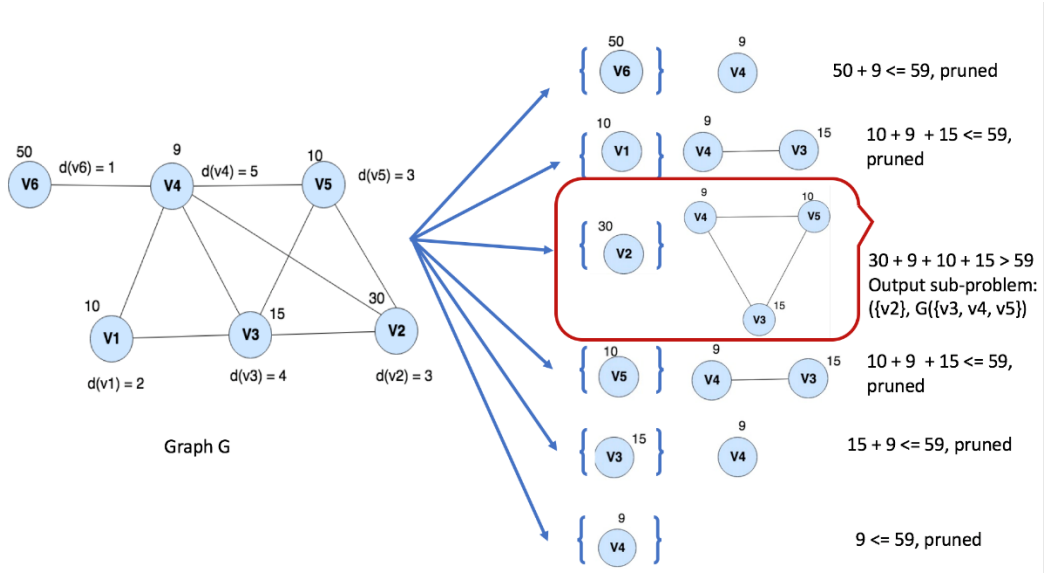

Figure 3.4. An example of partitioning algorithm

3.2.4 Solve Sub-problems on Each Executor

After generating sub-problems, we distribute them to executors which run single-node algorithm to compute candidate solutions in parallel. In this step, we can use any efficient sequential algorithms for MWCP. $FastWClq$ algorithm [6], which interleaves between clique construction and graph reduction, is an efficient algorithm which can find better solutions than state of the art algorithms while using less runtime. We use its main idea and implement a modified version which is efficient in distributed application.

In $FastWClq$ algorithm, a clique is constructed by extending from an empty set. In PS-MWC, we maintain the weight of maximum weight clique found so far in the global variable $global\_max\_weight$. Besides, the value of $global\_max\_weight$ is updated while executors are solving sub-problems in parallel. To avoid unnecessary construction procedures, we use a branch and bound strategy to stop construction procedures that are known not to get a higher clique weight than $global\_max\_weight$ [6]. In contrast to the original algorithm only prunes when a better clique is found, in our algorithm the pruning technique is applied at the beginning as well as when a clique is found for a sub-problem.

Before we describe $FastWClq$ algorithm, we introduce some notations. We use $path$ and $subG$ to denote the partitioning path and subgraph in the sub-problem we are solving. We use C to denote the clique we are constructing. $StartSet$ is the set containing all the vertices among which we want to find the maximum weight clique. As we defined in 3.1, $CandSet$ contains all vertices which can be added into C. We define $upperBound(v)$ as the upper bound on the weights of any clique extended from C by adding v and more vertices.

$$upperBound(v) = w(C) + w(v) + w(Neighbors(\mathrm{v}) \cap CandSet)$$

Algorithm 4 gives the pseudo code of the algorithm for solving sub-problems in each executor.

In step 3, we compare $upperBound$ of this sub-problem, which is defined in 3.2.3, with $global\_max\_weight$. If $upperBound$ of this sub-problem is less than or equal with $global\_max\_weight$, we can stop solving it as we discussed in 3.2.3. Note that we already use the same pruning method in the partitioning step; however, the value of $global\_max\_weight$ can become bigger and bigger in the process of solving sub-problems on executors; hence, we can discard a considerable number of sub-problems in the solving phase. Our experiment in Chapter 5 shows its efficiency by counting the number of pruned sub-problems in this phase. In step 5, if this sub-problem has an $upperBound$ greater than the $global\_max\_weight$, we get an $optimumWeight$ value by subtracting the weight of partitioning path from $global\_max\_weight$. In step 6, we reduce $subG$ with this $optimumWeight$ by iteratively removing vertices which have an $upperBound$ lower than or equal to the $optimumWeight$. Let G' denote the graph after reducing. In step 8, if $V(G')$ is empty, we stop solving this sub-problem, otherwise we start to construct a clique from G'.

In the clique construction procedure, we set $StartSet$ as the vertices of G'which we are working on in step 10, and pop a random vertex u from the $StartSet$ to serve as the starting vertex from which C will be extended in step 12. Next, we initialize C with u and set $CandSet$ to be $Neighbors(u)$. Then, in step 15 to 21, C is extended iteratively by adding a vertex from $CandSet$ until $CandSet$ becomes empty. In step 22 to 25, if the union of $path$ and C has a greater weight than $global\_max\_weight$, we update $global\_max\_weight$ to be $w(path) + w(C)$, and set C' which denotes the best clique we find so far to be the union of $path$ and C.

In step 27 to 32, if the $global\_max\_weight$ is updated, we reduce the working graph with the updated value and set $StartSet$ to be the vertices of reduced graph. We continue the procedure of constructing the clique as well as reducing the graph until $StartSet$ is empty.

ALGORITHM 4. Solve Sub-problems

1: Procedure FastWClq ($subProblem$)
Input: $subProblem$: $(path, subG)$
Output: $(C', w(C'))$
2: set $C = \emptyset, w(C) = 0, C' = \emptyset, w(C') = 0$
3: if $upperBound(subProblem) \leq global\_max\_weight$ then:
4: return $None$
5: set $optimumWeight = global\_max\_weight - w(path)$
6: set $G' = reduceGraph(subG, optimumWeight)$
7: set $old_global_$max$_weight = global_max_weight$
8: if $V(G') = \emptyset$ then:
9: return $None$
10: set $StartSet = V(G')$
11: while $StartSet \neq \emptyset$ do:
12: set $u = pop\ a\ random\ vertex\ from\ StartSet$
13: set $C = \{u\}, w(C) = w(u)$
14: set $CandSet = Neighbors(u)$
15: while $CandSet \neq \emptyset$ do:
16: set $addingV = ChooseAddVertex(CandSet)$
17: if $upperBound(addingV) + w(path) \leq global\_max\_weight$ then:
18: break
19: set $C = C \cup \{addingV\}, w(C) = w(C) + w(addingV)$
20: set $CandSet = CandSet \cup Neighbors(addingV)$
21: end while
22: if $w(path) + w(C) > global\_max\_weight$ then:
23: set $w(C') = w(path) + w(C)$
24: set $C' = path \cup C$)
25: set $global\_max\_weight = w(C')$
26: end if
27: if $old_global_$max$_weight < global_max_weight$ then:
28: set $optimumWeight = global\_max\_weight - w(path)$
29: set $G' = reduceGraph(subG, optimumWeight)$
30: set $StartSet = V(G')$
31: set $old_global_$max$_weight = global_max_weight$
32: end if
33: end while
34: return $(C', w(C'))$
35: end Procedure

In step 29 of Algorithm 4, we reduce the input graph with an $optimumWeight$ by iteratively removing vertices which have $upperBounds$ lower than or equal with the $optimumWeight$, which means they cannot be contained in a clique better than the best clique found so far. Algorithm 5 gives the pseudo code of the algorithm for reducing a graph.

We use $removeVs$ to denote the vertices to be removed from G.

In step 3 to 8, we put vertices which have $upperBound$ less or equal than $optimumWeight$ into $removeVs$. In step 9 to 18, we carry out a loop until $removeVs$ become empty. For each iteration of this loop, we pop a vertex from $removeVs$, and subtract its weight from its neighbors' $upperBound$. Once we find some neighbors' $upperBound$ less than or equal to $optimumWeight$, we put them into $removeVs$.

ALGORITHM 5. Reduce Graph

```
1:  Procedure ReduceGraph (G, optimumWeight)
        Input: a vertex weighted graph G = (V, E, w), optimumWeight
        Output: reduced Graph G'
2:      set removeVs = ∅
3:      foreach v in V(G) do:
4:          set upperBound(v) = w(v) + w(Neighbors(v))
5:          if upperBound(v) ≤ optimumWeight then:
6:              set removeVs = removeVs ∪ {v}
7:          end if
8:      end foreach
9:      while removeVs ≠ ∅ do:
10:         set u = pop a vertex from removeVs
11:         foreach v in Neighbors(u) do:
12:             set upperBound(v) = upperBound(v) − w(u)
13:             if upperBound(v) ≤ optimumWeight then:
14:                 set removeVs = removeVs ∪ {v}
15:             end if
16:         end foreach
17:         delete u and its incident edges from G
18:     end while
19:     return reduced Graph G'
20: end Procedure
```

In step 16 of Algorithm 4, during the construction procedure, we choose the adding vertex from $CandSet$ based on their estimated benefit values with a BMS (Best from Multiple Selection) heuristic [6], which is a probabilistic strategy that returns the best element from multiple samples. Let k denote the number of samples which we can control. It has been theoretically shown in [4] that BMS can approximate the best-picking strategy in $O(1)$ time.

ALGORITHM 6. Choose Adding Vertex from $CandSet$

1: Procedure ChooseAddVertex ($CandSet$)
Input: $CandSet$: Candidate set of vertices to be added in C
Output: $bestV$: a vertex with highest estimated benefit value in $CandSet$
2: if $|CandSet| < k$ then:
3: foreach v in $CandSet$ do:
4: set $estimated\_benefit(v) = w(v) + \frac{w(Neighbors(v) \cap CandSet)}{2}$
5: end foreach
6: set $bestV =$ the vertex with highest $estimated\_benefit$ value
7: else:
8: set $maxBenefit = 0$
9: for $i = 1$ to k do:
10: set $v =$ a random vertex in $CandSet$
11: set $estimated\_benefit(v) = w(v) + \frac{w(Neighbors(v) \cap CandSet)}{2}$
12: if $estimated\_benefit(v) > maxBenefit$ then:
13: set $maxBenefit = estimated\_benefit(v)$
14: set $bestV = v$
15: end if
16: end for
17: return $bestV$
18: end Procedure

CHAPTER 4

IMPLEMENTATION

Chapter 4 provides implementation details of main parts of PS-MWC. We implement PS-MWC using the GraphX library in Spark API, and use the Apache ZooKeeper service to maintain and update the value of shared variable $global\_max\_weight$ which can be accessed by each executor. Moreover, we improved the efficiency of our program according to the Spark Tuning Guide [24]. For example, we tuned the data serialization strategy, chose optimal data structures, and tuned the garbage collection strategy.

PS-MWC was implemented using Scala. We choose Scala because of its conciseness and efficiency from static typing. Furthermore, Scala runs on the JVM, so Scala and Java programs can be seamlessly integrated. The ZooKeeper API only has a Java and a C library, so we can write a Java class for utilizing ZooKeeper service, and call its function in our Scala code of PS-MWC.

We used Gradle to create a JAR package from the code. We installed Spark on a local machine with four cores, as well as on an AWS EMR cluster; thus, the implementation of PS-MWC can run locally with four worker threads, and on a YARN cluster as well.

4.1 Architecture

PS-MWC contains five main modules:

1. Main;
2. ZKHelper;
3. GetQuickMWCL;
4. Partition;
5. Solver.

The responsibilities of the Main module are:

1. It creates a *SparkContext* object, which communicates with the Cluster Manager.

2. It creates a property graph from the input file.
3. It calls the *GetQuickMWCL* module for getting a clique quickly.
4. It calls the *ZKHelper* module for connecting to ZooKeeper server and initializes the znode's value.
5. It collects *weightMap* and *neighborsMap* for vertices, and broadcasts these two Maps to workers in the cluster. The *weightMap* contains the key-value pairs of vertex id and vertex weight; while the *neighborsMap* contains the key-value pairs of vertex id and a set of its neighbors' ids. This ensures the graph information is available in memory for all workers, so that workers can independently solve sub-problems assigned to it.
6. It calls the *Partition* module that partitions the property graph into sub-problems. Specifically, we define a class called *Subproblem* which has three fields: *partition_path*, *upperBound*, and *vertexSet*. The *partition_path* is a list of vertex ids which indicates how the subgraph was obtained from the original graph. In our current implementation, each *partition_path* contains a vertex because we perform partitioning one time; however, it can contain more vertices if we perform partitioning multiple times. The *upperBound* is a long-type value which is the sum of weights of all vertices in this sub-problem and denotes the upper bound of weight of MWC in this sub-problem. The *vertexSet* contains a set of higher-order neighbors of vertices in *partition_path.*
7. It calls the *Solver* model for solving sub-problems and getting candidate solutions.
8. It finds and outputs the final solution.

4.2 ZKHelper

In PS-MWC, the global mutable variable $global\_max\_weight$ contains the weight of the best solution found and plays an important role in reducing the running time. This global variable should be accessed by all executors, as its value allows pruning of the search space. It can be updated by executors, when a better solution is found. The two shared variables provided by

Spark, broadcast variables and accumulators, cannot meet these requirements, because broadcast variables are read-only and accumulators' value cannot be accessed by executors.

We use Apache ZooKeeper service to maintain this global variable. With the ZooKeeper service, all executors register with ZooKeeper server to get updates when the variable's value is updated. Moreover, any executors can update the variable's value on the ZooKeeper server.

We set up a ZooKeeper server and implement a ZKHelper module to enable the driver program and executors to connect and communicate with the ZooKeeper server. The main functions in ZKHelper module are:

1. *startZK()* is called by the driver and executors to initiate a client connection to ZooKeeper server.
2. *initializeZNode()* is called by the driver to create the znode and initialize it with the input value.
3. *readWeightFromZookeeper()* is called by the driver and executors to read the data in the znode.
4. *updateZNode()* is called by executors to update the data in the znode.

4.3 GetQuickMWCL

This module takes a property graph as input and outputs a clique. We want to get an initial value for $global\_max\_weight$ in a short time; thus, we reduce the problem size by taking a sample with 500 vertices from the input property graph. Figure 4.1 shows the snippet code for getting a sub-graph with 500 vertices from original graph.

```
var workingGraph: Graph[Double, Int] =
graph.subgraph(vpred = (id, attr) => id <= 500).cache
```

Figure 4.1. Snippet code for getting a sub-graph with 500 vertices from original graph

Then, we generate a RDD which contains vertex ids, weights and degrees in this *workingGraph*, and define an ordering object in which vertices are ordered by weight and then by degree. Figure

4.2 provides the snippet code for the ordering object, in which we compare the weights of two vertices, and if the weights are equal compare their degrees.

```
val orderingByWeightAndDegree = new Ordering[Tuple2[VertexId
Tuple2[Double, Int]]]() {
  override def compare(x: (VertexId, (Double, Int)), y:
(VertexId, (Double, Int))): Int =
    if(x._2._1 > y._2._1 || x._2._1 < y._2._1) {
      Ordering[Double].compare(x._2._1, y._2._1)
    }else {
      Ordering[Int].compare(x._2._2, y._2._2)
    }
}
```

Figure 4.2. Snippet code for the ordering object

Based on this ordering, this module constructs a clique by keep choosing the vertex with the maximum weight and degree from candidate set which contains vertices can be added into the current clique.

4.4 Partition

The *Partition* module is in charge to partition the input graph into sub-problems. As is stated in Section 3.2.3, the most important task is to calculate higher-order neighbors $H(v)$ for each vertex. Based on the input graph, this module generates an *ubDegreeGraph* in which the edges RDD remains the same, and each vertex contains these attributes: its weight, upper bound and degree. The upper bound, i.e. the sum of weights of this vertex and all its neighbors, is used for initial pruning. If a vertex's upper bound is less or equal than $global\_max\_weight$, this vertex is not contained in final solution, thus, it doesn't need to send or receive any messages, avoiding unnecessary messages transfer.

Next, we use *degreeGraph.aggregateMessages()* to get $H(v)$ in parallel. Figure 4.3 shows the snippet code for calculating $H(v)$ for each vertex. Graph.*aggregateMessages()* is provided by the Spark GraphX library. This function transfers messages along the neighboring edges, and merges messages for each vertex. It has four parameters. The first parameter is the type of messages to be sent to each vertex. In our implementation, we set it to be a tuple that consists of a

Buffer[VertexId], in which we put each vertex's higher-order neighbor's id, and a *Double* value, which is the total weight of the higher-order neighbors. The second parameter is a *sendMsg()* function defined by the user, which runs on each edge and sends messages to neighboring vertices. In our implementation, only vertices which have an upper bound value greater than the $global\_max\_weight$ can send and receive messages, avoiding unnecessary costs. Messages are sent from higher-order vertices to its lower-order neighbors. The third parameter is a *mergeMsg()* function defined by the user, which combines messages that have been sent to the same vertex. In our implementation, we merged the higher-order neighbor's id into a buffer for each vertex, and added up their weights. The fourth parameter is *tripletFields*, by which users can specify which fields should be used in *sendMsg()* function. If *tripletFields* is not defined, all fields in triplets are used by default.

```
ubDegreeGraph.aggregateMessages[(Buffer[VertexId],
Double)](
    //sendMsg() function
    triplet => {
      //srcAttr and dstAttr: (weight, upperbound,
degree)
      //send message only if the upperbounds of src and
desc are both larger than global_max_weight
      if(triplet.srcAttr._2 > global_max_weight &&
triplet.dstAttr._2 > global_max_weight) {
        if (triplet.srcAttr._3 > triplet.dstAttr._3) {
          triplet.sendToDst((Buffer(triplet.srcId),
triplet.srcAttr._1))
        } else if (triplet.srcAttr._3 ==
triplet.dstAttr._3) {
          if (triplet.srcId > triplet.dstId)
triplet.sendToDst((Buffer(triplet.srcId),
triplet.srcAttr._1))
          else triplet.sendToSrc((Buffer(triplet.dstId),
triplet.dstAttr._1))
        } else {
          triplet.sendToSrc((Buffer(triplet.dstId),
triplet.dstAttr._1))
        }
      }
    },
    //mergeMsg() function
    //merge the higher-order neighbors'id and add up the
weights
    (n1, n2) => (n1._1 ++ n2._1, n1._2 + n2._2)
  ).leftJoin(weightedGraph.vertices)((vid, setAndSumHNb,
weight) =>
    (weight.getOrElse(0D) + setAndSumHNb._2,
setAndSumHNb._1.toSet)).
    filter(vertex => vertex._2._1 > global_max_weight)
```

Figure 4.3. Snippet code for calculating higher-order neighbors for each vertex

4.5 Improve Efficiency by Tuning

In this section, we describe several tuning activities to improve efficiency of PS-MWC in the light of Spark Tuning Guide [24]. We tuned the implementation in the following aspects: data serialization, data structure, and garbage collection.

4.5.1 Data Serialization

Spark provides two serialization libraries: Java serialization and Kryo serialization. The default serialization library is Java serialization, which can work with any class implements

java.io.Serializable interface. Kryo serialization is significantly faster and more compact than the Java serialization; however, it does not support all serializable types, so users need to register classes used in the program manually.

Figure 4.4 shows a snippet code for setting Spark configuration to use Kryo serialization. After the setting, we can register classes by calling *registerKryoClasses()* method of *SparkConf*. We set *spark.kryo.registrationRequired* to be *true*, so that exceptions will be thrown for unregistered classes; then, we can register them manually.

```
conf.set("spark.serializer",
        "org.apache.spark.serializer.KryoSerializer")
conf.set("spark.kryo.registrationRequired", "true")
```

Figure 4.4. Snippet code for setting Spark configuration to use Kryo serialization

By switching to Kryo serialization, the time cost was reduced by 10% when we test on a dataset called web-uk-2005 which contains 130,000 vertices and 12 million edges.

4.5.2 Data Structures

We tried to choose the best data structure for certain computation, especially for the most frequent operations. One important computation in PS-MWC was to calculate higher-order neighbors for vertices, which was implemented by calling *aggregateMessages(sendMsg, mergeMsg)* function. In this function, each vertex sends its id to its lower-order neighbors, and then merges the messages it receives into a set of ids of its higher-order neighbors. In this computation, merging received messages for each vertex is the most frequent operation; thus, we want to choose the best data structure for the messages, which can achieve the best performance on *concatenation* operation. According to [35], as is shown in Table 4.1, *mutable.Buffer* performs the best on *concatenation* operation among 9 collections in Scala.

Table 4.1. Runtimes of various collections on concatenation benchmark in seconds [35]

concat	0	1	4	16	64	256	1,024	4,096	16,192
Array++	89	83	85	91	144	330	970	4,100	17,000
arraycopy	23	18	20	27	48	280	1,000	4,000	16,000
List	7	81	162	434	1,490	5,790	23,200	92,500	370,000
Vector	5	48	188	327	940	3,240	12,700	52,000	210,000
Set	91	95	877	1,130	5,900	26,900	149,000	680,000	3,600,000
Map	54	53	967	1,480	6,900	31,500	166,000	760,000	4,100,000
m.Buffer	11	32	32	38	70	250	700	3,900	20,000
m.Set	58	81	142	1,080	4,200	16,000	69,000	263,000	1,160,000
m.Map	47	69	181	990	3,700	15,000	62,000	290,000	1,500,000

Our initial implementation used *Set* as the type of messages. Later, by using *mutable.Buffer* instead of *Set*, the time cost was reduced by 14% when tested on the dataset web-uk-2005.

4.5.3 Garbage Collection

We set the *spark.executor.extraJavaOptions* in Spark configuration to be *"-XX:+UseG1GC"*. With the benefit of the G1GC garbage collector, The GC time costs of our program have been greatly reduced. Table 4.2 and Table 4.3 shows the GC time costs before and after the setting.

Table 4.2. The GC time using default garbage collector

Summary metrics for 128 completed tasks

Metric ▲	Min	25th percentile	Median	75th percentile	Max
Duration	5.6 min	8.8 min	12 min	15 min	19 min
GC time	38 s	54 s	1.1 min	1.3 min	3.7 min
Input (size / records)	6.1 MiB / 1	8.3 MiB / 2	8.4 MiB / 2	8.4 MiB / 2	10.1 MiB / 2

Table 4.3. The GC time using G1GC garbage collector

Summary metrics for 128 completed tasks

Metric ▲	Min	25th percentile	Median	75th percentile	Max
Duration	4.0 min	7.3 min	11 min	13 min	16 min
GC time	3 s	6 s	8 s	10 s	13 s
Input (size / records)	8.3 MiB / 2	8.4 MiB / 2	8.4 MiB / 2	9.9 MiB / 2	10.2 MiB / 2

In Table 4.2, when we used default garbage collector, the min GC time on the tasks was 38 s, while the max GC time was 3.7 min.

In Table 4.3, for the same input data, when we used G1GC garbage collector, the min GC time on the tasks was reduced to 3 s, while the max GC time was reduced to 13 s.

CHAPTER 5

EXPERIMENTAL EVALUATION

Several experiments were carried out to evaluate the performances of PS-MWC on several real-world massive graphs. In these experiments, we examined its solution quality, benefits of the partitioning strategy, scalability, as well as the pruning efficiency.

Experiments showed that PS-MWC found the same solutions as FastWClq [6] and WLMC [5], two state-of-art MWC algorithms on a benchmark of MWCPs. Furthermore, experimental results showed that our strategy obtained significant performance improvement than other alternative strategies. The benefits of our partitioning strategy were also evaluated. After partitioning, the sizes of sub-problems were significantly smaller than the maximum degree of vertices in the original input graph. We also examined the relationship between the number of vertices of graph and the runtime of PS-MWC, as well as the pruning efficiency. The results showed that there was an almost linear relationship between the number of vertices in graphs and the corresponding time cost of PS-MWC. We also showed that our pruning strategy was highly efficient on test datasets, as 99.80% of sub-problems had been pruned, which meant only 0.2% of sub-problems were solved. Moreover, results showed that adding more cores achieved better time efficiency. For a graph with more than 3 million vertices and more than 117 million edges, the PS-MWC took 1349 and 575 seconds on an EMR cluster with 64 and 128 cores, respectively.

5.1 Datasets

The benchmarks we used in the experiments were downloaded from the Network Data Repository online [25], including graphs of social networks (soc-orkut-dir), biological networks (bio-dmela), genes (bio-human-gene1), web link networks (web-uk-2015), and collaboration networks (ca-dblp-2010, ca-dblp-2012, ca-hollywood-2009). We pre-processed the graphs by removing self-loops and multiple adjacencies. A self-loop is an edge both of whose endpoints are the same vertex [38]. Multiple adjacencies denote that multiple edges have the same endpoints.

After the pre-processing, the graphs were converted into simple graphs. Moreover, the original benchmarks were unweighted graphs, so we transformed them into weighted graphs with such a weight function: for the i^{th} vertex v_i, $w(v_i) = (i \ mod \ 200) + 1$. This weight function was also used in [4, 5, 6]. The maximum graph in the experiments had more than 3 million vertices and more than 117 million edges.

5.2 Platform

The experiments were carried out on a standalone machine and two Amazon EMR clusters. The standalone machine has a 2.8GHz quad-core Intel Core i7 processor and 16G memory.

Amazon EMR provides a managed Hadoop framework on which we can run Apache Spark and other distributed frameworks. It enables users to process data across clusters which consist of multiple Amazon EC2 instances [39].

In our experiments, one EMR cluster consists of one master node and seven worker nodes, while the other EMR cluster consists of one master node and 15 worker nodes. The master node is of m4.2xlarge type, which contains 16 cores and 32 GB memory. The worker nodes are of m3.xlarge type, each of which contains 8 cores and 15 GB memory. The nodes in the EMR clusters read input datasets from Amazon S3, an AWS data store, where we store the datasets.

5.3 Experimental Results

5.3.1 Comparison of Solution Quality

In the first experiment, we evaluated the solution quality of our algorithms on the standalone machine. By comparing our solution with two algorithms: FastWClq [6], and WLMC [5], Table 5.1 shows that PS-MWC found the same solutions with these two algorithms. In Table 5.1, the first four columns show the name and statistics information of each dataset. The last three columns show the solutions of those three algorithms for each graph. A solution of an algorithm for a graph denotes the weight value of the maximum weight clique found by this algorithm in this graph. In this table, K denotes a thousand, and M denotes a million.

Table 5.1. Solution Quality Comparison of FastWClq [6], WLMC [5], and PS-MWC

Graph	|V|	|E|	Density	Solution of FastWClq	Solution of WLMC	Solution of PS-MWC
bio-dmela	7.4K	25.6K	0.0009	805	805	805
ca-dblp-2010	226.4K	716.5K	2.8e-05	7575	7575	7575
ca-dblp-2012	317.1K	1M	2.09e-05	14108	14108	14108
web-uk-2015	129.6K	11.7M	0.0014	54850	54850	54850
ca-hollywood-2009	1.1M	56.3M	9.85e-05	222720	222720	222720

5.3.2 Runtime Comparison of Different Strategies

In the second experiment, we compared runtime difference between four different strategies on the standalone machine. The differences of the strategies lied in the partitioning methods and whether to use the ZooKeeper service. Except for those differences, other parts of implementations were the same.

In strategy_0, we partitioned the input graph basing on vertices' degree and then vertices' id, and used the ZooKeeper service to maintain a global best solution found for pruning the sub-problems. This strategy was adopted in PS-MWC.

In strategy_1, we didn't partition the input graph into sub-problems. In other words, we skipped the partitioning phase and directly solved the original graph in the solving phase.

In strategy_2, we partitioned the input graph basing on vertices' id, and used the ZooKeeper service.

In strategy_3, we partitioned the input graph basing on vertices' degree and then vertices' id, which was the same partitioning strategy as strategy_0, but did not use the ZooKeeper service for pruning. Instead, we used a broadcast variable which contained the weight value of the clique we get quickly. As is stated in Section 4.2, this broadcast variable is read-only, which means

executors can read its value but cannot update its value; thus, it can only prune the sub-problems in the partitioning phase, not in the solving phase.

Through comparison of runtime difference between those four strategies, we evaluated the benefits of using the ZooKeeper service to maintain a global best solution found so far, as well as the benefits of our partitioning strategy over other alternative strategies.

This experiment was conducted on the standalone machine with a 2.8 GHz quad-core Intel Core i7 processor and 16 G memory. The results given in Table 5.2 showed that except for the graph bio-dmela, strategy_0 obtained significant performance improvement over the other alternative strategies. The reason why strategy_3 was faster than strategy_0 on the graph bio-dmela might be that the size of this graph was relatively small, so the weight of the clique we got quickly could prune a large proportion of sub-problems in the partitioning step. Besides, updating the value of znode using the ZooKeeper service brought about additional time costs, which could be the reason why strategy_0 costed more time than strategy_3 on the graph bio-dmela. In this table, K denotes a thousand, and M denotes a million.

Table 5.2. Runtime Comparison of Four Strategies

Graph	**|V|**	**|E|**	**Time Cost of Strategy_0**	**Time Cost of Strategy_1**	**Time Cost of Strategy_2**	**Time Cost of Strategy_3**
bio-dmela	7.4K	25.6K	**2.07 s**	15.435 s	1.981 s	1.917 s
ca-dblp-2010	226.4K	716.5K	**7.638 s**	Longer than 5 min	10.4 s	15.480 s
ca-dblp-2012	317.1K	1M	**11.683 s**	Longer than 5 min	41.888 s	30.406 s
web-uk-2015	129.6K	11.7M	**40.225 s**	Longer than 20 min	273 s	Longer than 20 min

5.3.3 Partitioning Properties

In the third experiment, we ran PS-MWC for four graphs to evaluate the properties of our partitioning strategy. Table 5.3 shows the statistics of partitioning results of those graphs. The first five columns show the name and statistic information of each graph, in which the column

d_{max} denotes the maximum degree of vertices, and d_{avg} denotes the average degree of vertices. The 6th column denotes the number of generated sub-problems. The last two columns denote maximum number and average number of vertices in sub-problems. In this table, K denotes a thousand, and M denotes a million.

Table 5.3. Partitioning Results Statistics: number of vertices, number of edges, the maximum degree of vertices, average degree of vertices, number of generated sub-problems, maximum number of vertices in sub-problems, and average number of vertices in sub-problems

Graph	|V|	|E|	d_{max}	d_{avg}	Num of sub-problems	Max |V| in sub-problems	Average |V| in sub-problems
ca-hollywood-2009	1.1M	56.3M	11.5K	105	1069085	2208	52
soc-orkut-dir	3M	117M	33K	76	3072280	535	38
bio-human-gene1	21.9K	12.3M	7.9K	1.1K	21872	2338	563
web-uk-2005	129.6K	11.7M	850	181	129324	499	90

The results showed that after partitioning, the maximum size of sub-problems was much smaller than the maximum degree of vertices in the input graph. We may conclude that the partitioning strategy is robust to skewed degree distribution, in which quite few vertices have very large degree. Take the graph soc-orkut-dir as an example, the maximum degree of vertices was 33 thousand while the total number of vertices was 3 million, after partitioning, the maximum number of graphs in sub-problems were 535.

5.3.4 Scalability

In the fourth experiment, we used a massive data set, soc-orkut-dir, which we called it G5, to examine the relationship between the number of vertices and edges of graph and the runtime, as well as the pruning efficiency. Based on G5, we generated its subgraphs G1, G2, G3, and G4 using the following procedure. G1 was induced by 1/5 of vertices in G5; G2 was induced by 2/5

of vertices in G5, and so on. We conducted this experiment on an Amazon EMR cluster which consisted of one master node and seven worker nodes. The master node was of m4.2xlarge type, which contains 16 cores and 32 GB memory. The worker nodes are of m3.xlarge type, each of which contains 8 cores and 15 GB memory. For each graph, we ran PS-MWC on this cluster for five times, and reported the median time cost in seconds. Table 5.4 shows the number of vertices and edges in each graph, its density, and time cost of PS-MWC for each graph. In Table 5.4, we can see that from G1 to G4, the time cost increased almost linearly. For G4 and G5, the time costs are almost the same.

Table 5.4. Runtime Comparison of Five Graphs

Graph	|V|	|E|	Density	Time Cost(s)
G1	614,488	17,941,635	9.5e-05	147
G2	1,228,976	43,377,222	5.7e-05	483
G3	1,843,464	73,911,110	4.3e-05	981
G4	2,457,952	96,137,820	3.1e-05	1342
G5	3,072,441	117,185,083	2.5e-05	1349

We also examined the pruning efficiency by calculating the number of pruned sub-problems in both partitioning and solving steps. The numbers of pruned sub-problems in the two steps were obtained by creating two accumulator variables in the driver program, one variable for each step. Accumulators are variables that are used for aggregating information across the executors. Tasks running on executors in a cluster can add value to them. In our implementation, every time we discarded a sub-problem on each executor, we added 1 to the corresponding accumulator. Finally, the driver program read their value and output them.

Table 5.5. Pruning Efficiency

Graph	|V|	Time Cost (s)	Total Number of Pruned Sub-problems	Pruned percentage	Total Number of Solved Sub-problems
G1	614,488	**147**	613603	**99.86%**	**885**
G2	1,228,976	**483**	1226702	**99.80%**	**2274**
G3	1,843,464	**981**	1840880	**99.86%**	**2584**
G4	2,457,952	**1342**	2455226	**99.89%**	**2726**
G5	3,072,441	**1349**	3069429	**99.90%**	**3012**

Table 5.5 shows the number of vertices in each graph, time cost in seconds, total number of pruned sub-problems, pruned percentage, total number of solved sub-problems. The pruned percentage was calculated by dividing the number of vertices by the total number of pruned sub-problems. The results showed that, more than 99.80% of sub-problems were pruned, which means only 0.2% of sub-problems were solved.

Figure 5.1 manifests time cost and number of solved sub-problems when applying PS-MWC on the five graphs. The black line denotes the relationship between time cost and number of vertices in graphs, while the red line denotes the relationship between number of solved sub-problems and number of vertices in graphs.

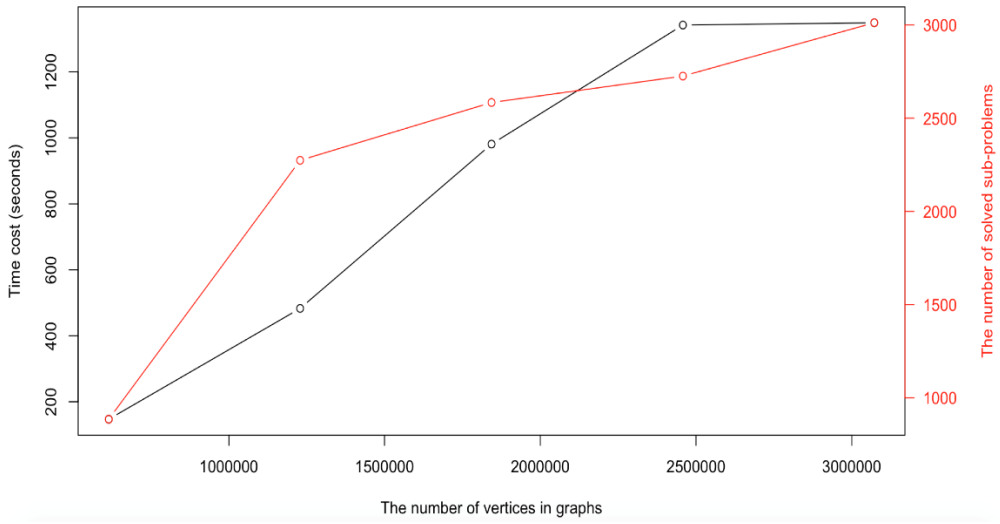

Figure 5.1. Time cost and solved sub-problems number when applying PS-MWC on graphs with different number of vertices.

In the fifth experiment, we ran PS-MWC for the same datasets on a cluster with different cores to evaluate its performance when the number of cores of executors increased. At first, we ran PS-MWC for G4 and G5 in the fourth experiment on an Amazon EMR cluster consisted of one master node and seven worker nodes, using 8 executors, each with 8 cores and 8 G memory. Afterwards, we ran PS-MWC for the same datasets on another Amazon EMR cluster consisting of one master node and 15 worker nodes, using 16 executors, each with 8 cores and 8 G memory.

We ran five times for each graph on each cluster, and reported the median time cost in seconds. Table 5.6 shows the number of vertices and edges in each graph, total number of cores of executors in EMR cluster, and the time cost of PS-MWC.

Table 5.6. Performance Evaluation When Increasing the Number of Cores

Graph	|V|	|E|	Number of cores	Time Cost (s)
G4	2,457,952	96,137,820	64	1342
			128	539
G5	3,072,441	117,185,083	64	1349
			128	575

As is indicated in Table 5.6, for the graph G5 with more than 3 million vertices and more than 117 million edges, PS-MWC took 1349 and 575 s on an EMR cluster with 64 and 128 cores, respectively. The results showed that when the cores of executors in EMR cluster doubled, the time cost decreased by more than two times. Based on the results, we are confident that PS-MWC can solve the MWCP for massive datasets which contain millions of vertices and hundreds of millions of edges. In order to process larger graphs, PS-MWC merely requires more memory and cores to efficiently solve the MWCP.

CHAPTER 6

CONCLUSION

In this thesis, we propose a scalable and distributed algorithm, PS-MWC, to solve the MWCP for massive graphs. As far as we know, PS-MWC is the only approach of solving the MWCP leveraging the Apache Spark distributed framework. PS-MWC harnesses the power of a distributed computing paradigm to partition a massive graph into independent sub-problems and then solve those sub-problems in parallel. In order to reduce the overlap between sub-problems as well as for load balancing, we define an order for vertices and generate one sub-problem for each vertex by inducing the subgraph which only contains its higher-order neighbors. In contrast to other partitioning methods [20, 21], which sort the vertices first and partition the input graph in sequential, we partition the input graph in parallel relying on a message passing approach, leveraging the capabilities of the GraphX framework in which each vertex sends messages to its neighbors and merges messages it received. Moreover, PS-MWC can utilize any existing sequential MWC algorithms for solving sub-problems on a single core. Meanwhile, we use the Apache ZooKeeper service to maintain and update the value of a global variable, which denotes the weight of the best solution found so far, in a data node that can be accessed by all executors. This global variable enables us to identify sub-problems which cannot beat current best solution in which case we prune those sub-problems, saving runtime.

The experimental results showed that PS-MWC found the same solutions as FastWClq [6] and WLMC [5], on a benchmark of MWCPs. The benefits of our partitioning strategy were also examined. Our experimental results showed that the partitioning strategy of PS-MWC obtained significant performance improvement than other alternative strategies. Moreover, after partitioning, the maximum size of the sub-problems was significantly smaller than the maximum degree of vertices in the input graph. In the scalability experiment, results showed that there was an almost linear relationship between the number of vertices in graphs and the corresponding time

cost of PS-MWC. Moreover, we demonstrated that our pruning strategy was highly efficient, as 99.80% of sub-problems were pruned, which means only 0.2% of sub-problems were solved. Furthermore, PS-MWC could solve the MWCP for massive datasets which contain millions of vertices and hundreds of millions of edges. For a graph with more than 3 million vertices and more than 117 million edges, PS-MWC took 1349 and 575 seconds on an EMR cluster with 64 and 128 cores, respectively.

We have shown that PS-MWC is efficient, distributed and scalable. It can solve MWCP on massive graphs which cannot be solved within reasonable time on a single processor. In order to process larger graphs, PS-MWC merely requires more memory and cores to efficiently solve the MWCP.

There is room for improvement for PS-MWC:

1. For now, we set up a ZooKeeper server in standalone mode. This is convenient for evaluation, development, and testing [36]. In the future, to avoid single point of failure, we can improve the implementation by running ZooKeeper in replicated mode, which means running a replicated group of servers which have copies of the same configuration file. In this way, even when a ZooKeeper server fails, the service will be available as long as a majority of the servers are up.
2. We utilize the GraphX library in the current implementation of PS-MWC. Another library GraphFrames has been proposed by [37] in 2016. GraphFrames, implemented over Spark SQL, provides a pattern library to generate and compose patterns such as cliques programmatically. It can "*combine relational processing, pattern matching and graph algorithms and optimize computations across them*" [37]. In the future, we will try to utilize the GraphFrame library, which supports all the operators available in GraphX, to see if it can achieve better efficiency.

3. We provide a straightforward method for calculating a clique from original graph quickly. It does get a clique in a very short time, but the quality of the obtained solution is quite low. In the future, we will explore an efficient alternative approach for getting a clique with the greater weight value the better, within a short time.
4. Finally, we will integrate PS-MWC with a spatial hotspot discovery framework [22, 23] for finding an optimal set of hotspots by removing overlapping hotspots.

REFERENCES

[1] Basagni, S. (2001). Finding a maximal weighted independent set in wireless networks. *Telecommunication Systems*, 18(1-3), 155-168.

[2] Sanghavi, S., Shah, D., & Willsky, A. S. (2009). Message passing for maximum weight independent set. *IEEE Transactions on Information Theory*, 55(11), 4822-4834.

[3] Flier, H., Mihalák, M., Schöbel, A., Widmayer, P., & Zych, A. (2010). Vertex disjoint paths for dispatching in railways. In *Proceedings of Algorithmic Approaches for Transportation Modeling, Optimization, and Systems* (pp. 61-73).

[4] Wang, Y., Cai, S., & Yin, M. (2016). Two Efficient Local Search Algorithms for Maximum Weight Clique Problem. In *AAAI Conference on Artificial Intelligence* (pp. 805-811).

[5] Jiang, H., Li, C. M., & Manya, F. (2017). An Exact Algorithm for the Maximum Weight Clique Problem in Large Graphs. In *AAAI Conference on Artificial Intelligence* (pp. 830-838).

[6] Cai, S., & Lin, J. (2016). Fast Solving Maximum Weight Clique Problem in Massive Graphs. In *International Joint Conferences on Artificial Intelligence* (pp. 568-574).

[7] Fang, Z., Li, C. M., & Xu, K. (2016). An exact algorithm based on maxsat reasoning for the maximum weight clique problem. *Journal of Artificial Intelligence Research*, 55, 799-833.

[8] Kumlander, D. (2004). A new exact algorithm for the maximum-weight clique problem based on a heuristic vertex-coloring and a backtrack search. In *Modelling, Computation and Optimization in Information Systems and Management Sciences* (pp. 202-208).

[9] Kumlander, D. (2008). On importance of a special sorting in the maximum-weight clique algorithm based on colour classes. In *Modelling, computation and optimization in information systems and management sciences* (pp. 165-174).

[10] Mascia, F., Cilia, E., Brunato, M., & Passerini, A. (2010). Predicting Structural and Functional Sites in Proteins by Searching for Maximum-weight Cliques. In *AAAI Conference on Artificial Intelligence* (pp. 1274-1279).

[11] Östergård, P. R. (2001). A new algorithm for the maximum-weight clique problem. *Nordic Journal of Computing*, 8(4), 424-436.

[12] Östergård, P. R. (2002). A fast algorithm for the maximum clique problem. *Discrete Applied Mathematics*, 120(1-3), 197-207.

[13] Yamaguchi, K., & Masuda, S. (2008). A new exact algorithm for the maximum weight clique problem. In *ITC-CSCC: International Technical Conference on Circuits Systems, Computers and Communications* (pp. 317-320).

[14] Shimizu, S., Yamaguchi, K., Saitoh, T., & Masuda, S. (2012). Some improvements on Kumlander's maximum weight clique extraction algorithm. In *Proceedings of World Academy of Science, Engineering and Technology* (pp. 307-311)

[15] Shimizu, S., Yamaguchi, K., Saitoh, T., & Masuda, S. (2013). Optimal table method for finding the maximum weight clique. In *Proceedings of the 13th International Conference on Applied Computer Science* (pp. 84-90).

[16] Kiziloz, H. E., & Dokeroglu, T. (2018). A robust and cooperative parallel tabu search algorithm for the maximum vertex weight clique problem. *Computers & Industrial Engineering*, 118, 54-66.

[17] Pardalos, P. M., Rappe, J., & Resende, M. G. (1998). An exact parallel algorithm for the maximum clique problem. In *High performance algorithms and software in nonlinear optimization* (pp. 279-300).

[18] Peng, L., Zebing, W., & Ming, G. (2010). A maximum clique algorithm based on MapReduce. In *Advanced Computer Theory and Engineering* (Vol. 6, pp. V6-87).

[19] Xiang, J., Guo, C., & Aboulnaga, A. (2013). Scalable maximum clique computation using MapReduce. In *2013 IEEE 29th International Conference on Data Engineering* (pp. 74-85).

[20] Tomita, E., & Kameda, T. (2007). An efficient branch-and-bound algorithm for finding a maximum clique with computational experiments. *Journal of Global optimization*, 37(1), 95-111.

[21] Elmasry, A., Khalafallah, A., & Meshry, M. (2016). A scalable maximum-clique algorithm using Apache Spark. In *Computer Systems and Applications* (pp. 1-8).

[22] Akdag, F., & Eick, C. F. (2016). Interestingness Hotspot Discovery in Spatial Datasets Using a Graph-Based Approach. In *Machine Learning and Data Mining in Pattern Recognition* (pp. 530-544).

[23] Akdag, F., Davis, J. U., & Eick, C. F. (2014). A computational framework for finding interestingness hotspots in large spatio-temporal grids. In *Proceedings of the 3rd ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data* (pp. 21-29).

[24] Spark Tuning Guide. Retrieved May 17, 2018, from https://spark.apache.org/docs/latest/tuning.html

[25] Network Data Repository. Retrieved from http://networkrepository.com/

[26] Prithviraj Bose. Spark Accumulators Explained: Apache Spark [Blog post]. Retrieved from https://www.edureka.co/blog/spark-accumulators-explained

[27] Zaharia, M. (2016). An architecture for fast and general data processing on large clusters. Retrieved from https://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-12.pdf

[28] Spark Programming Guide. Retrieved from https://spark.apache.org/docs/2.2.0/rdd-programming-guide.html

[29] Jacek Laskowski. Overview of Apache Spark. Retrieved January, 2018, from https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-overview.html

[30] Jacek Laskowski. Broadcast Variables. Retrieved January, 2018, from https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-broadcast.html

[31] GraphX Programming Guide. Retrieved November, 2017, from https://spark.apache.org/docs/latest/graphx-programming-guide.html

[32] Increasing the speed of calculating relationship chain for massive graph with hundreds of billions of edges by 20 times utilizing Spark GraphX [Blog post]. Retrieved April, 2018, from http://djt.qq.com/article/view/1487

[33] ZooKeeper: A Distributed Coordination Service for Distributed Applications. Retrieved February, 2018, from https://zookeeper.apache.org/doc/r3.5.0-alpha/zookeeperOver.html

[34] Graph Analytics with GraphX. Retrieved April, 2018, from http://ampcamp.berkeley.edu/big-data-mini-course/graph-analytics-with-graphx.html

[35] Haoyi, Li. (2016, September 29). Benchmarking Scala Collections [Blog post]. Retrieved from http://www.lihaoyi.com/post/BenchmarkingScalaCollections.html

[36] ZooKeeper Getting Started Guide. Retrieved February, 2018, from https://zookeeper.apache.org/doc/r3.4.12/zookeeperStarted.html

[37] Dave, A., Jindal, A., Li, L. E., Xin, R., Gonzalez, J., & Zaharia, M. (2016). GraphFrames: an integrated API for mixing graph and relational queries. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems* (pp. 2).

[38] Glossary of Graph Theory Terms. In Wikipedia. Retrieved May, 2018, from https://en.wikipedia.org/wiki/Glossary_of_graph_theory_terms

[39] Amazon EMR. Retrieved from https://aws.amazon.com/emr/