PowerWalk: Scalable Personalized PageRank via Random Walks with Vertex-Centric Decomposition

Qin Liu\textsuperscript{1}, Zhenguo Li\textsuperscript{2}, John C.S. Lui\textsuperscript{1}, Jiefeng Cheng\textsuperscript{1}
\textsuperscript{1}The Chinese University of Hong Kong
\textsuperscript{2}Huawei Noah’s Ark Lab
\{qliu, cslui\}@cse.cuhk.edu.hk
\{li.zhenguo, cheng.jiefeng\}@huawei.com

ABSTRACT
Most methods for Personalized PageRank (PPR) precompute and store all accurate PPR vectors, and at query time, return the ones of interest directly. However, the storage and computation of all accurate PPR vectors can be prohibitive for large graphs, especially in caching them in memory for real-time online querying. In this paper, we propose a distributed framework that strikes a better balance between offline indexing and online querying. The offline indexing attains a fingerprint of the PPR vector of each vertex by performing billions of “short” random walks in parallel across a cluster of machines. We prove that our indexing method has an exponential convergence, achieving the same precision with previous methods using a much smaller number of random walks. At query time, the new PPR vector is composed by a linear combination of related fingerprints, in a highly efficient vertex-centric decomposition manner. Interestingly, the resulting PPR vector is much more accurate than its offline counterpart because it actually uses more random walks in its estimation. More importantly, we show that such decomposition for a batch of queries can be very efficiently processed using a shared decomposition. Our implementation, PowerWalk, takes advantage of advanced distributed graph engines and it outperforms the state-of-the-art algorithms by orders of magnitude. Particularly, it responds to tens of thousands of queries on graphs with billions of edges in just a few seconds.

Keywords
Personalized PageRank; random walks; vertex-centric decomposition

1. INTRODUCTION
Nowadays graph data is ubiquitous, usually in very large scale. It is of great interest to analyze these big graphs to gain insights into their formation and intrinsic structures, which can benefit services such as information retrieval and recommendations. Consequently, large-scale graph analysis becomes popular recently, especially in domains like social networks, biological networks, and the Internet, where big graphs are prevalent. It is found that general-purpose dataflow systems like MapReduce and Spark are not suitable for graph processing. Instead, many graph computing engines such as Pregel \cite{GPP} and PowerGraph \cite{Powe} are being developed, which can be orders of magnitude more efficient than general-purpose dataflow systems \cite{GPP, Powe, Map}. A key feature of most graph engines is the adoption of a simple yet general vertex-centric programming model that can succinctly express a wide variety of graph algorithms \cite{GPP, Powe, Map}. Here, a graph algorithm is formulated into a user-defined vertex-program which can be executed on each vertex in parallel. Each vertex runs its own instance of the vertex-program which maintains only local data and affects the data of other vertices by sending messages to these vertices. To leverage the power of such a graph engine, it is thus important to formulate the algorithm into a vertex-centric program.

PageRank is a popular measure of importance of vertices in a graph \cite{Page}. This is particularly useful in large applications where data and information are crowded and noisy; a measure of their importance allows to process them by importance \cite{Page}. One prominent application is the web search engine of Google where the pages returned, in response to any query, are ordered by their importance measured by PageRank. The intuition behind PageRank is that a vertex is important if it is linked to by many important vertices. This “global” view of vertex importance does not reflect individual preferences, which is inadequate in modern search engines, online social networks, and e-commerce, where customizable services are in urgent need. Consequently, “personalized” PageRank (PPR) \cite{Page} attracts considerable attention lately that allows to assign more importance to a particular vertex in order to provide “personalized views” of a graph. PPR has been widely used in various tasks across different domains: personalization of web search \cite{Page}, recommendation in online social networks \cite{Rec} and mobile-app marketplaces \cite{Rec}, graph partitioning \cite{Rec}, and other applications \cite{Rec}. While the enabling of “customization” in PageRank greatly expands its utility, it also brings a huge challenge on its computation – the customization to each vertex leads to a workload $N$ times of the original PageRank, where $N$ is the number of vertices in the graph, and the customization on any combination of vertices exponentially increases the complexity.

Despite significant progress, existing approaches to PPR computation are still restricted in large-scale applications. While the power iteration method is used in most graph engines to calculate the global PageRank \cite{GPP, Powe, Map, Map2}, it is impractical for PPR computation, which would take $O(N(N + M))$ time for all PPR vectors ($N$ and $M$ are the numbers of vertices and edges respectively) \cite{GPP}. Furthermore, most algorithms for PPR computation are designed for single-machine in-memory systems which cannot deal with large-scale problems \cite{GPP, Powe, Map}. The Hub Decomposition algorithm \cite{GD} allows personalization only for a small set of vertices (hubs). To achieve full personalization (the computation of all $N$ PPR vectors), it requires $O(N^2)$ space. Fogaras et al. \cite{Fog} proposed a scalable Monte-Carlo solution to full personalization. This approach first simulates a large number of short random walks from each vertex, and store them in a database. Then it uses these random walks to approximate PPR vectors for online queries. As we will show in our evaluation (Section 4.3), to achieve high accuracy, the simulation of random walks is very time-consuming. Also, the precomputed database is usually too large to fit in the main memory for large graphs which makes this method undesirable in practice.
Bahmani et al. \cite{2} suggested to concatenate the short random walks in the precomputed database to answer online queries. This extension can reduce the size of database, but it still incurs random access to the graph data and precomputed database. If implemented on a distributed graph engine, it would require thousands of iterations to concatenate short random walks together to form a long walk which makes it inefficient.

One can see that, given the great effort in scaling up PPR, it remains open for a practically scalable solution, in that: (1) it enables full personalization for all vertices in the graph; (2) it is scalable for very large problems; (3) it supports online query efficiently; and (4) it executes a large number of queries in reasonable time, which is critical to modern online services where numerous requests are fed to a system simultaneously \cite{2}.

In this paper, we present a novel framework, PowerWalk, for scalable PPR computation. We implement PowerWalk on distributed graph engines, in order to harness the power of a computing cluster. Furthermore, to enable fast online services, we separate the computation into offline preprocessing and online query, in a flexible way that the computation can be shifted to the offline stage as much as the memory budget allows. The offline stage uses a variant of the Monte-Carlo methods \cite{13,19} to compute an approximation for each PPR vector. We can tune the precision of PPR vectors according to the memory budget, so that all approximate PPR vectors can be cached in distributed memory. At query time, the PPR vectors of the querying vertices are computed efficiently based on the precomputed PPR approximate vectors by utilizing the Decomposition theorem in \cite{19}. Our main contributions are:

- We propose a Monte-Carlo Full-Path (MCFP) algorithm for offline PPR computation, and prove that it converges exponentially fast. In practice, it requires only a fraction of random walks to achieve the same precision compared with the previous Monte-Carlo solution \cite{13}.

- We propose a Vertex-Centric Decomposition (VERD) algorithm which can provide results to online PPR queries in real-time and more accurate than their offline counterparts. More importantly, it is able to execute a large number of queries very efficiently with a shared decomposition.

- Combining the MCFP and VERD algorithms, we propose an efficient distributed framework, PowerWalk, for computing full personalization of PageRank. PowerWalk can adaptively trade off between offline preprocessing and online query, according to the memory budget.

- We evaluate PowerWalk on billion-scale real-world graphs and validate its state-of-the-art performance. On the Twitter graph with 1.5 billion edges, it responses to 10,000 queries in 3.64 seconds and responses to 100,000 queries in 17.96 seconds.

The rest of the paper is organized as follows. We elaborate our two algorithms for PPR computation in Section 2. Section 3 presents the implementation of PowerWalk. We extensively evaluate PowerWalk in Section 4. Section 5 reviews related work on PPR computation. Finally, Section 6 concludes the paper.

2. ALGORITHM

In this section, we propose two algorithms for computing Personalized PageRank (PPR) for web-scale problems. The first algorithm is called the Monte-Carlo Full-Path (MCFP) Algorithm, which is based on the Monte-Carlo simulation framework and can be used to compute the fully PPR (or all PPR vectors of all vertices in the graph) in an offline manner. Our second algorithm is called the Vertex-Centric Decomposition (VERD) Algorithm, which is capable of computing the PPR vector for any vertex in an online manner. As we will show in Section 3, the proper coupling of these two algorithms can lead to a practically scalable solution to the PPR computation which can response to a large number of queries very efficiently. Before we proceed to our algorithms, let us first provide some preliminaries.

2.1 Preliminaries

Let $G = (V, E)$ be a directed graph, where $V$ is the set of vertices and $E$ is the set of edges. An edge $(u, v) \in E$ is considered to be directed from $u$ to $v$. We also call $(u, v)$ as an out-edge of $u$ and an in-edge of $v$, and call $v$ as an out-neighbor of $u$ and $u$ as an in-neighbor of $v$. Let $N = |V|$ and $M = |E|$ denote the numbers of vertices and edges respectively, and $O(v)$ denote the set of out-neighbors of $v$.

Personalized PageRank (PPR) measures the “importance” of all vertices from a perspective of a particular vertex. Let $p_u$ be the stochastic row vector that represents the PPR vector of $u$ (i.e., $p_u$ is a non-negative row vector with entries summed up to 1), and $p_u(v)$ represents the Personalized PageRank of vertex $v$ from the perspective of the source vertex $u$. If $p_u(v) > p_w(v)$, this means that, from the perspective of vertex $u$, vertex $v$ is more important than vertex $w$. Finally, $p_u$ can be defined as the solution of the following equation \cite{13}:

$$p_u = (1 - c)p_u A + c e_u,$$  \hspace{1cm} (2.1)

where $A$ is a row-stochastic matrix and $A_{i,j}$ equals to $1/O(i)$ if $(i,j) \in E$, $e_u$ is a unit row vector with the $u$-th entry $e_u(u)$ being one and zero elsewhere, and $c \in (0, 1)$ is a given teleport probability. Typically, $c$ is set to 0.15, and empirical studies show that small changes in $c$ have little effect in practice \cite{35}. If $i$ is a dangling vertex (i.e., one without any out-edge), to make sure $A$ is row-stochastic, one typical adjustment is to add an artificial edge from $i$ to $u$ \cite{6}. This is equivalent to setting row $i$ of $A$ as $e_u$, and as a result, any visit to vertex $i$ will immediately follow by a visit to vertex $u$. This reflects the intrinsic of $p_u$, that is, it encodes the importance of all vertices from the perspective of vertex $u$.

The above equilibrium equation has the following random walk interpretation: a walk starts from vertex $u$; in each jump, it will move to one random out-neighbor with probability $1 - c$, or jump back (or teleport) to vertex $u$ with probability $c$. Particularly, if the current vertex is a dangling vertex, then the next jump will surely go to vertex $u$. In summary, the PPR vector $p_u$ is exactly the stationary distribution of such a random walk model, and $p_u(v)$ is the probability of the random walk visiting $v$ at its equilibrium.

2.2 The MCFP Algorithm

The fact that $p_u(v)$ is the probability of a random walk visiting $v$ at its equilibrium immediately suggests that one can approximate $p_u(v)$ by $x_u(v)/n$ for a large $n$, where $x_u(v)$ denotes the number of visits to $v$ in $n$ steps. However, simulating a long random walk (or a random walk with many jumps) on a large graphs is inefficient for the following reasons. As a large graph usually cannot be held in the main memory, it must be partitioned across multiple machines or to be stored on the disk. At each step, the random walk may jump to a vertex at another machine or access data on disk, incurring an expensive network or disk I/O and latency. Moreover, a long walk takes many iterations to reach the steady state solution, which can become a bottleneck for systems like MapReduce.

\footnote{1. e.g., $A$ is non-negative and the entries in each row sum up to 1.}
Finally, there is an inherent restriction of implementing a long random walk on any graph computation engines: the dependency between adjacent moves prevents executing the walks in parallel.

Figure 1: Breaking a long walk into three short ones

To overcome this inefficiency, we propose to transform the long random walk into many short walks by breaking it at every teleport move. This is illustrated in Figure 1. One crucial benefit of doing so is that these short random walks can be executed in parallel. In the next section, we will show how to simulate billions of short random walks on a distributed cluster. The above idea translates to our Monte-Carlo Full-Path (MCFP) algorithm for PPR computation as stated in Algorithm 1 and we show in Theorem 2.1 that it converges exponentially fast.

Algorithm 1: Monte-Carlo Full-Path (MCFP)

| Input: vertex $u$, the number of random walks $R$ |
| Output: an approximation of $p_u$ |

1. Simulate $R$ random walks starting from $u$;
2. At each step, each of the $R$ random walks terminates with probability $c$ and takes a further step according to the matrix $A$ with probability $1 - c$;
3. Approximate $p_u(v)$ by the fraction of $R$ random walks that terminate at vertex $v$, i.e., $p_u(v) \sim x_u(v)/n$, where $x_u(v)$ is the number of visits by all $R$ random walks to vertex $v$, and $n$ is the total moves in all $R$ random walks;

Theorem 2.1. Let $\hat{p}_u$ denote the estimator of $p_u$ obtained from Algorithm 1. The probability of over-estimating $p_u$ can be bounded as follows:

$$\Pr [\hat{p}_u(v) - p_u(v) \geq \gamma] \leq \frac{1}{\sqrt{c}} \left(1 + \frac{\gamma c}{10}\right) \exp \left(-\frac{\gamma^2 R}{20}\right).$$

The same bound also holds for the probability of under-estimation.

Proof. Please refer to Appendix A.

Prior Art. Let us compare with one current state-of-the-art random walk method in computing PPR [13]. Consider a random walk starting from vertex $u$. At each step, the random walk terminates with probability $c$, or it jumps to other states according to the matrix $A$ with probability $1 - c$. It has been proved in [22] that the last visited vertex of such a random walk has a distribution $p_u$ which suggests another Monte-Carlo algorithm in approximating $p_u$. Note that the distribution $p_u$ is achieved by simulating the random walk for many times, and each time, the random walk may end in a different vertex. So one needs to normalize all such ending vertex occurrences to have the probability distribution $p_u$. This state-of-the-art algorithm was proposed in [13] and it is illustrated in Algorithm 2.

Note that while Algorithm 2 takes only the ending vertex of each random walk into account, our proposed MCFP algorithm leverages the full trajectory of each random walk. Our motivation is that the intermediate vertices on each trajectory contain significant information regarding the distribution of the random walk and should be included in approximating. Our theoretical analysis stated in Theorem 2.1 guarantees not only the validity of our algorithm but also the exponential convergence rate. Our evaluation in Section 4 confirms that this does lead to a much better approximation. In other words, our MCFP is far more efficient than the state-of-the-art algorithm under the same precision. For example, to achieve the same precision of our MCFP algorithm with 1,000 random walks, Algorithm 2 needs to simulate 6,700 random walks.

2.3 The VERD Algorithm

With the MCFP algorithm, we can approximate the fully PPR, i.e., we can compute an approximate vector $p_u$ of $p_u$ for all $u \in V$, which can be carried out offline. Depending on our memory budget, we can adjust the precision of our approximations by varying $R$ used in the MCFP algorithm, so that all approximations of PPR vectors fit in the distributed memory. For web-scale problems and with limited memory budget, this can lead to low-resolution approximations. In this section, we propose a method to recover in real-time a high-resolution PPR vector from low-resolution approximations. Our method is inspired by the decomposition theorem [22] Theorem 3.

Theorem 2.2 (Decomposition). Suppose $p_u$ is the PPR vector of vertex $u$. We have

$$p_u = c \cdot e_u + \frac{1 - c}{|O(u)|} \sum_{v \in O(u)} p_v.$$ (2.2)

This theorem basically says that the PPR vector of a vertex can be recovered from the PPR vectors of its out-neighbors. Another more important observation is that such an approximation is more accurate than those of its out-neighbors, because it actually uses $R \cdot |O(u)|$ random walks when $p_v \in O(u)$, uses $R$ random walks, and the error of $p_u$ is reduced by a factor of $1 - c$. Furthermore, the decomposition theorem can be applied recursively to achieve higher precision. This suggests Algorithm 3 in approximating the PPR vector $p_u$ of vertex $u$ with $T$ iterations. When $T = 0$ (Lines 8 & 9), the algorithm simply returns the precomputed approximation obtained from our MCFP algorithm.

However, implementing such a recursive algorithm on distributed graph engines appears to be very challenging, because existing graph engines only provide vertex-centric programming interface which does not support recursion. On the other hand, there is a huge redundancy in Algorithm 3 because the out-neighbors of different vertices are likely to overlap. To illustrate our idea, consider the graph in Figure 2. To compute $p_u$, with two iterations of recursion by calling $\text{decomp}(v_1, 2), \text{decomp}(v_2, 0), \text{decomp}(v_3, 0)$, and $\text{decomp}(v_4, 0)$ will be called twice.

Algorithm 2: Monte-Carlo End-Point (MCEP) in [13]

| Input: vertex $u$, the number of random walks $R$ |
| Output: an approximation of $p_u$ |

1. Simulate $R$ random walks starting from $u$;
2. At each step, each of the $R$ random walks terminates with probability $c$ and takes a further step according to the matrix $A$ with probability $1 - c$;
3. Approximate $p_u(v)$ by the fraction of $R$ random walks that terminate at vertex $v$, i.e., $p_u(v) \sim y(v)/R$. Here, $y(v)$ denotes the number of the random walks terminate at $v$;
Algorithm 3: Recursive Decomposition

function $\text{decomp}(u, T)$

Input: a vertex $u$, the number of iterations $T$

Output: an approximation of $p_u$

if $T = 0$ then
    return the precomputed $p_u$
return $c \cdot e_u + \frac{1}{|\mathcal{O}(u)|} \sum_{v \in \mathcal{O}(u)} \text{decomp}(v, T - 1);$

Figure 2: A simple graph to illustrate the VERD algorithm

To remove such redundancy in $T$ levels of recursion, we suggest to *unfold* the $T$ levels of recursion by applying the decomposition in Equation 2.3 iteratively $T$ times:

$$p_u = s_u^{(T)} + \sum_{v \in \mathcal{V}} f_u^{(T)}(v)p_v;$$  

where $s_u^{(T)}$ and $f_u^{(T)}(v)$ are two row vectors given by

$$s_u^{(T)} = s_u^{(T-1)} + \sum_{v \in \mathcal{V}} c \cdot f_u^{(T-1)}(v)e_v$$

$$f_u^{(T)}(v) = \sum_{v \in \mathcal{V}} \frac{1-c}{|\mathcal{O}(w)|} \sum_{v \in \mathcal{O}(w)} f_u^{(T-1)}(v)e_v.$$

where

$$s_u^{(0)} = 0, f_u^{(0)} = e_u.$$

The benefits of the unfolded decomposition in Equation 2.3 are several-fold. First, besides the removal of redundancy, it enables vertex-centric programming, as we show in Algorithm 4. Consider Lines 6-9, the updates on the two vectors are carried entirely in a vertex-centric manner, which makes it easy to implement on existing graphs as shown in Section 3.4. For this reason, we call Algorithm 4 the Vertex-Centric Decomposition (VERD) algorithm. Second, since the VERD algorithm is iterative, it will not overload the stack like the recursive decomposition algorithm (Algorithm 3) when $T$ is large. Finally, as we show in Section 3.3, it allows to simultaneously compute multiple PPR vectors very efficiently by sharing network transfer, which is important to online services. Theorem 2.3 below shows that Algorithm 4 is equivalent to Algorithm 3.

**Theorem 2.3.** For $u \in \mathcal{V}$ and $T \geq 0$, $\text{decomp}(u, T) = \text{vc-decomp}(u, T)$.

**Proof.** Please refer to Appendix B.

To illustrate how the VERD algorithm removes redundant calls, consider again the graph in Figure 2. Calling $\text{vc-decomp}(v_1, 2)$, we have $s_{v_1}^{(2)}$ equals to

$$\left( c, \frac{c(1-c)}{2}, \frac{c(1-c)}{2}, 0, 0, 0, 0, 0 \right).$$

Algorithm 4: Vertex-centric Decomposition (VERD)

function $\text{vc-decomp}(u, T)$

Input: a vertex $u$, the number of iterations $T$

Output: an approximation of $p_u$

$s_u^{(0)} \leftarrow 0, f_u^{(0)} \leftarrow e_u;
for i = 1 \text{ to } T$ do
    $f_u^{(i)} = 0;$
    foreach $w \in \mathcal{V}$ do
        if $f_u^{(i-1)}(w) > 0$ then
            foreach $v \in \mathcal{O}(w)$ do
                $f_u^{(i)}(v) \leftarrow f_u^{(i)}(v) + \frac{1-c}{|\mathcal{O}(w)|} f_u^{(i-1)}(v);$ $s_u^{(i)}(w) \leftarrow s_u^{(i-1)}(w) + c \cdot f_u^{(i-1)}(v);$
    $s_u^{(T)}(v) \leftarrow \sum_{v \in \mathcal{V}} f_u^{(T)}(v)p_v;$

and $f_u^{(2)}(v_1)$ equals to

$$\left( 0, 0, 0, \frac{(1-c)^2}{8}, \frac{(1-c)^2}{4}, \frac{(1-c)^2}{4}, \frac{(1-c)^2}{4}, \frac{(1-c)^2}{8} \right).$$

Then the VERD algorithm simply returns a new approximation of $p_{v_1}$ by computing

$$s_{v_1}^{(2)} + \sum_{i=1}^8 f_{v_1}^{(2)}(i)p_{v_1}.$$

In this process, each involved precomputed approximation is loaded only once. In contrast, in the recursive decomposition algorithm, we have to load $p_{v_1}, p_{v_2},$ and $p_{v_3}$ twice.

The sizes of $s_{v_1}^{(T)}$ and $f_u^{(T)}$ grow with the number of iterations $T$, and in the worst case, they can be up to the size of $p_u$, which is the number of vertices reachable from $u$. In practice, when $T$ is small, $s_{v_1}^{(T)}$ and $f_u^{(T)}$ are usually highly sparse which can be stored in hash tables or sorted vectors. Also, we can discard values below a certain threshold $\epsilon$ in $s_{v_1}^{(T)}$ and $f_u^{(T)}$ to keep them sparse when $T$ becomes larger. On distributed graph engines, network communication is mainly caused by the synchronization at each iteration. As we will show in Section 4.2 by utilizing the precomputed approximations, our VERD algorithm needs just a few iterations to achieve reasonable accuracy, which makes it quite efficient for online query.

**Prior Art.** In Algorithm 4 when $T \to \infty$, we have $f_u^{(T)} \to 0$. So when $T$ is large enough, Algorithm 4 can be used to compute PPR vectors without any precomputed approximations by simply returning $s_{v_1}^{(T)}$ as an approximation of $p_{v_1}$. Similar approaches were also used in [11]. The key difference between our algorithm and previous approaches is that our VERD algorithm can utilize the precomputed approximations to accelerate the online query of PPR vectors. As shown in Section 4.3 with precomputed approximations, the query response time can be reduced by 86%.

3. IMPLEMENTATION

We now present our complete solution to PPR computation, including its implementation details. Our proposed framework is called PowerWalk, which consists of two phases: offline preprocessing and online query. In the offline preprocessing, our MCFP algorithm is used to approximate the fully PPR (all the vectors $p_u$ for all $u \in \mathcal{V}$). The size and computation time of an approximate PPR vector depend on the number of random walks $R$ in the

*Note: The text contains mathematical formulas and algorithmic descriptions that are typical in a research document, focusing on graph theory and algorithms for computing PPR vectors. The content is formatted using LaTeX and includes diagrams and pseudocode to illustrate the concepts. The document is structured with clear sections and subsections, with theorems, proofs, and algorithm descriptions. The figures and tables are not reproduced here but are integral to the understanding of the text.*
MCFP algorithm, which in turn depends on the available memory budget. In practice, $R$ should be chosen such that the approximate fully PPR can be computed in reasonable time and can be cached in the allocated memory. We call these approximate PPR vectors the “PPR index.” Recall that our MCFP algorithm simulates $R$ random walks for each vertex in $V$. Therefore, the major challenge in generating the PPR index is how to efficiently simulate all $N \cdot R$ random walks. We will discuss our solution implemented on top of DrunkardMob [25] in Section 3.1.

For the PPR query to a vertex $u$ (i.e., to compute $p_u$), we have two options: (1) we can return the approximate PPR vector $p_u$ in the PPR index directly; (2) we can use our proposed VERD algorithm to compute a new approximate PPR vector $p_u$ from the PPR index. In the first case, to achieve high accuracy, we need a very large $R$ in the offline preprocessing. As we will show in Section 4.3 when $R$ is large, the preprocessing is very time consuming. Also the resulting PPR index is too large to fit in the main memory which slows down the online query significantly. The key idea of PowerWalk is that, to achieve the same level of accuracy as the first case, the second case allows us to use a smaller $R$ in the preprocessing and migrate the computation cost to the online query. More importantly, the smaller $R$ allows us to cache the PPR index in the main memory which accelerates the online query greatly. We report in Section 4.3 that the second case is much more efficient in handling large batches of queries which is common in nowadays search engines [2]. We will discuss how to execute a large number of PPR queries efficiently on PowerGraph [17] in Section 3.3.

### 3.1 Preprocessing

In preprocessing, we use our MCFP algorithm to approximate $p_u$ for every vertex $u \in V$. These approximate PPR vectors will constitute the PPR index which accelerates online query. The MCFP algorithm simulates $R$ random walks per vertex, thus $N \cdot R$ random walks in total. Depending on the memory budget, there are two cases to consider. If the graph can be cached in the main memory of a single machine, we can use a simple loop to simulate one random walk for each source vertex, where the walk will be confined to one machine and no network or disk latency will be incurred. Repeating this procedure $N \cdot R$ times gives us $R$ random walks per vertex.

If the graph is too large to be cached in the main memory of a single machine, it has to be partitioned and distributed across multiple machines or to be accessed from disk. Then at each step, the random walk may incur a network transmission or disk access. As network or disk I/O is much slower than memory access, the random walk simulation will be inefficient. To address this challenge, Kyrola proposes an algorithm called DrunkardMob to simulate billions of random walks on massive graphs on a single PC [25]. Since it has been shown to be an efficient solution for computing random walks and is also used in production by Twitter [25], we implement our offline preprocessing based on DrunkardMob, with several important improvements, as discussed below.

DrunkardMob is built on top of disk-based graph computation systems like GraphChi [26]. To improve the performance, DrunkardMob simulates a large number of random walks simultaneously and assumes that the states of all walks can be held in memory. The states of walks can be seen as a mapping from vertices to walks: for each vertex, DrunkardMob knows the walks whose last hop is in that vertex. In a graph computation system, the edges are usually sorted by their source vertex, which means we can load a chunk of vertices and their out-neighbors using a sequential disk I/O. At each iteration, the graph computation system performs a sequential scan over the whole graph. For each incoming vertex and its out-neighbors, DrunkardMob first retrieves all walks on that vertex from memory and then moves these walks to the next hop.

To further accelerate the preprocessing, we reimplement DrunkardMob on our system, VENUS [30] (a disk-based graph computation system in C++) and extend it in a distributed environment using MPI [34]. PowerWalk distributes the simulation of random walks to a set of workers. One of the workers acts as the master to coordinate the remaining slave workers. The master divides the set of vertices of an input graph into disjoint intervals. When there is an idle slave, the master assigns an interval of vertices to that slave. Each slave works independently by using DrunkardMob to simulate $R$ random walks for each vertex in the assigned interval and computes an approximate PPR vector for each vertex. Finally, all approximate PPR vectors compose the PPR index which is used to accelerate the online query.

Remark. Simulating random walks on graphs is a staple of many other ranking and recommendation algorithms [14][23]. Our implementation of random walk simulation is highly competitive compared to previous solutions on general distribute dataflow systems. For example, to simulate 100 random walks from each vertex on the Twitter graph, DrunkardMob on VENUS takes 1728.2 seconds while an implementation on Spark takes 2967 seconds on the same cluster [29]. The detailed setup of our experiments is described in Section 4.

Next, we give a brief introduction to PowerGraph before we proceed to present our online query implementation.

### 3.2 PowerGraph

To achieve low query response time, it is essential to utilize the main memory. Hence we implement the query phase of PowerWalk on PowerGraph [17], which is a popular distributed in-memory graph engine. Note that it is also possible to implement PowerWalk on shared-memory graph processing systems like Galois [36] and Ligra [39]. However, we do not consider this case, since their maximum supported graph size is limited by the memory size of a single machine.

Many recent graph engines adopt a flexible vertex-centric programming model [33][18][17][12], which is quite expressive in encompassing most graph algorithms. A graph algorithm can be formulated into a vertex-program which can be executed on each vertex in a parallel fashion, and a vertex can communicate with neighboring vertices either synchronously or asynchronously. Each vertex runs its own instance of the vertex-program which maintains only local information of the graph and performs user-defined tasks. The vertex-program can affect the data of other vertices by sending messages to these vertices.

Compared with other graph engines [33][25], PowerGraph [17] is empowered by a more sophisticated vertex-centric GAS (Gather-Apply-Scatter) programming model. In the GAS model, a vertex-program is split into three conceptual phases: gather, apply, and scatter. In executing the vertex-program for a vertex, the gather phase assembles information from adjacent edges and vertices. The result is then used in the apply phase to update the vertex data. Finally, the scatter phase distributes the new vertex data to the adjacent vertices.

Next, we show how to formulate the query phase of our proposed framework, PowerWalk, on top of PowerGraph.

### 3.3 Batch Query

Most existing methods for fully PPR first compute and store the fully PPR in a database, and at query time, load the PPR vectors from the database directly [1][25]. This is not scalable as the computation and storage of fully PPR can be prohibitively costly, es-
especially for large graphs. For example, our evaluation shows that it takes more than 11 hours to compute the fully PPR for the uk-
union graph with 133.6 million vertices and 5.5 billion edges when
\( R = 2000 \), and it needs 1.1 TB to store the PPR index. The re-
quired time and space increase with the size of the graph and the
number of random walks starting from each vertex.

In practice, it is much desired to allow to allocate the computa-
tion and storage between the offline and online stages, according to
the available budget in memory, response time, and precision. Mo-
tivated by this, our framework PowerWalk proposed in this paper
computes an approximate of fully PPR offline, whose computa-
tion and storage cost can be decided based on the allowed time and
memory. These precomputed approximate PPR vectors are called
PPR index. At query time, we apply our VERD algorithm to effi-
ciently compute a more precise approximation of the PPR vectors
of interest based on the PPR index. Particularly, our VERD al-
gorithm can execute a large number of queries efficiently in par allel,
which is important to modern online services where multiple re-
quests can come at the same time. Our key observation is that the
computation of multiple PPR vectors shares the access to the graph
and the PPR index, so the small network packets used to access
the graph and the PPR index can be multiplexed and aggregated into
packets with large payload. This reduces the average computation
time of each PPR vector, because bulk network transfer workloads
are more efficient. For online query, PowerWalk buffers the incom-
ing PPR queries and computes a batch of PPR queries at a time. As
we will show in Section 4.3, our evaluation indicates that we can
compute thousands of PPR queries in several seconds.

We now elaborate further how we can perform batch PPR queries.
We use a vertex set \( S \) to represent a batch of PPR queries: for each
\( u \in S \), we would like to approximate \( p_u \). We can utilize our vertex-
program on PowerGraph to compute \( f_u \) and \( s_u \), for each \( u \in S \) as
shown in Algorithm 5. Each instance of the vertex-program main-
tains two maps: \( f \)-map and \( s \)-map. For the instance of vertex \( u \),
\( f \)-map and \( s \)-map represent \( f_u \) and \( s_u \) respectively. At the
beginning of each iteration, \( f \)-map is initialized with \( \text{prev}_f \)
from the previous iteration (Line 7), except that at the first
iteration we let \( f \)-map = \( e_u \) for \( u \in S \) (Line 8). In the method
\text{apply}(), for each vertex \( w \) that \( f_u \) is not zero, we update
\( s_u \) and \( f_u \) accordingly. The method \text{scatter}() will be
invoked on each out-edge of vertex \( u \), and \( f \)-map will be sent to
each out-neighbors of vertex \( u \). At the end of each iteration, the
PowerGraph engine will aggregate \( f \)-map by their target vertices.

In summary, the online query can be split into two steps. First,
we run the above vertex-program iteratively to compute \( f_u \) and \( s_u \)
for \( u \in S \). Suppose \( p_u \) is the approximate PPR vector of \( u \) stored
in the index. In the second step, we compute a refined approxima-
tion for each \( p_u \) as discussed in Section 4.4:

\[
p_u = s_u + \sum_{v \in V} f_u(v) \tilde{p}_v.
\]

### 3.4 Analysis

In this subsection, we analyze the complexity of our MCFP al-
gorithm and VERD algorithm. We also elaborate on some choices
we made in designing PowerWalk.

Let us start with the MCFP algorithm for a single vertex \( u \) de-
scribed in Algorithm 1. If the graph fits in the memory of a sin-
gle machine, the time complexity is \( O(R/c) \), since we simulate \( R \)
walks and the average length of these walks is \( 1/c \). More impor-
tantly, the size of the obtained approximation \( \tilde{p}_u \) is also \( O(R/c) \).
So the size of the PPR index obtained in the preprocessing phase
is bounded by \( N \cdot R/c \). On the other hand, consider the VERD
algorithm for a single vertex \( u \) described in Algorithm 2. When
the number of iterations \( T \) is large enough, the worst case time com-
plexity is \( O(N + M) \), since the algorithm works in the same way as
the breadth-first search. Similarly, in the worst case, the size of
\( s_u, f_u \), and the final approximation \( p_u \) is \( O(N) \).

To compute the PPR vectors for all vertices in a vertex set \( S \)
at query time, it is possible to use one of the two Monte-Carlo
methods described in Section 2.2 instead of the VERD algorithm
by starting a number of random walks from each vertex in \( S \). To
make sure that most random walks have terminated, we need to
run a Monte-Carlo method for at least 10 iterations. On the other
hand, if we use the VERD algorithm for online query, we only
need two iterations to achieve reasonable accuracy as shown in our
evaluation (Section 4.3). On distributed settings, the running time
is mainly decided by the bulk network transfers in each iteration
which makes the VERD algorithm a preferred approach. We com-
pare the running time of the MCFP algorithm and the VERD al-
gorithm in terms of online query in Section 4.3 and show that the
VERD algorithm is more efficient. For example, to execute 10,000
PPR queries on a Twitter social network with 41 million vertices,
the MCFP algorithm takes 179.38 seconds while the VERD algo-
rithm takes only 3.64 seconds, which has orders of magnitude im-
provement in computational efficiency.

### 4. Evaluation

In this section, we experimentally evaluate our framework, Pow-
nerWalk. In Section 4.2, we evaluate the accuracy of our MCFP
algorithm and our VERD algorithm with various parameters. In
Section 4.3, we evaluate the time and space costs of the offline pre-
processing and also the running time of the online batch query in
PowerWalk. Let us first introduce the setup of our experiments.

### 4.1 Experimental Setup

In the following experiments, unless we state otherwise, we set
the teleport probability to 0.15.

#### Datasets

To evaluate PowerWalk, we run our experiments on six
real-world graph datasets as shown in Table 1. Our datasets are of
different types: wiki-Vote is a Wikipedia who-votes-on-whom net-
work; twitter-2010 is an online social network; and web-BerkStan,
web-Google, uk-1m, and uk-union are web graphs.

#### Cluster

We perform all experiments on a cluster of eight ma-

we choose 100 vertices when selecting vertices, denoted by \( v \), from each vertex. Then we compute the average RAG for these vertices at different levels of \( R \) to achieve RAG larger than 0. Because the computation of the ground truth is very costly, we evaluate the accuracy on four small graphs: wiki-Vote, web-BerkStan, web-Google, and uk-1m.

In this subsection, we evaluate the accuracy of our MCFP algorithm and the MCEP algorithm. From Figure 3, we observe that our MCFP algorithm with only 1,000 random walks achieves the same level of accuracy as the existing MCEP algorithm with 6,700 random walks. This means that the dependent sample points from the MCFP algorithm and the independent sample points from the MCEP algorithm actually have almost the same effect in approximating PPR vectors. From Figure 4, we also observe that the RAG decreases as \( k \) increases. The reason is that when we have a larger value of \( k \), it is more difficult to approximate the top-\( k \) set, because lower ranked vertices has smaller difference in PPR scores.

**Effectiveness of the VERD algorithm.** As described in Section 2, our VERD algorithm computes a new PPR vector based on the PPR index whose precision is decided by the number of random walks \( R \) used in the preprocessing. Obviously, if the precision of the PPR index is low, we have to run the VERD algorithm for more iterations. If the precision of the PPR index is high enough for online query, then PowerWalk can return the PPR vector in the index directly. In this experiment, we first use the preprocessing procedure described in Section 4 to generate the PPR index with \( R = 10 \) or 100. Then, we use the batch query procedure based on our VERD algorithm described in Section 5.3 to execute the PPR queries. The results are shown in Figure 5. Here, when \( T = 0 \), it means that the batch query procedure returns the PPR vector stored in the PPR index directly. Also, when \( R = 0 \), it means we use the VERD algorithm directly without the precomputed PPR index. On all four graphs, to achieve RAG larger than 0.99, we need \( T = 7, 5, 2 \) iterations when \( R = 0, 10, 100 \) respectively. The results confirm with our hypothesis: with more random walks used in the preprocessing phase, we need less iterations during the batch query phase to achieve the same level of accuracy. However, more random walks also increase the overheads of the preprocessing phase. Let us examine the time and space costs of the preprocessing phase and also the running time of the batch query phase in the next subsection.

**4.3 Performance.**

In this subsection, we first evaluate the space and time costs of the preprocessing phase of PowerWalk on two large graphs twitter-2010 and uk-union. Then, we compare the online query of PowerWalk with other previous state-of-the-art methods. **Offline Preprocessing.** We show the preprocessing costs in Table 1.
The preprocessing time is sublinear to the number of random walks $R$ starting from each vertex, which is an attractive property. When we set $R = 100$, the preprocessing time of the uk-union graph is 53 mins and the index size is 1.6X compared to the graph size. As we will show later, we can achieve very fast online query when we set $R = 100$.

Fogaras et al. [13] propose to use the Monte-Carlo methods to first compute the fully PPR, and then return the PPR vectors directly as query results. This approach can be considered as a special case of our PowerWalk when $R$ is large. In the preprocessing, when $R$ is large enough, the precomputed approximate PPR vectors are already accurate enough and can be returned directly as online query results. If we want to achieve RAG larger than 0.99, this special case can be achieved by setting $R$ to 2000. However, as shown in Table 2 in this case, the preprocessing takes 11.6 hours for the uk-union graph to finish, which makes it undesirable for large graphs. More importantly, since the index size increases almost linearly with $R$, when $R = 2000$, the PPR index (1.48 TB on the disk for the twitter-2010 graph) cannot fit in the distributed memory (around 2.9 TB in our cluster) due to the additional storage overhead of the hash tables. This significantly slows down the online query as we will show later. This means that for very large graphs, computing and storing the fully PPR will be impractical. In contrast, PowerWalk only computes and caches a light-weight PPR index which allows fast online query.

**Online Query.** In Section 4.2, we observe that to achieve RAG larger than 0.99, when $R$ is set to 0, 10, and 100 in the preprocessing, the VERD algorithm should set the number of iterations $T$ to 7, 5, and 2 respectively. We compare the query response time by using different values of $R$ in the preprocessing, and these results are shown in Figure 4. We can observe that when $R$ increases, the query response time decreases because it needs less iterations to achieve the same accuracy. Note that $R = 0$ means that PowerWalk runs without the PPR index. In comparison, with the PPR index generated by setting $R = 100$, the query response time is reduced by 86% on the uk-union graph when the number of queries is 10,000 vertices. Moreover, when the number of queries increases, the query response time increases very slowly. This means our

**5. RELATED WORK**

There are extensive studies on Personalized PageRank computation. Here, we highlight some examples. The idea of Personalized PageRank is first proposed in [35]. The simplest method for computing PPR is by the power iteration method [35]. To compute $p_u$, we first compute the fully PPR, and then return the PPR vectors directly. However, in this case, the PPR index cannot fit in the main memory, so we store the PPR index in a constant key/value storage library called sparkey.

The results are summarized in Table 3. PowerWalk outperforms PI and MCFP significantly, especially when the number of queries is large. For example, as shown in Table 3 the MCFP algorithm takes 179.38 seconds to compute 10,000 PPR vectors on twitter-2010, while our VERD algorithm takes just 9.06 and 3.64 seconds when $R = 0$ and 100 in preprocessing respectively. Note that the power iteration method cannot even handle multiple queries due to its large space requirement. When the number of queries is one, FPPR has the fastest response time. The reason is that all queries are done in the same way and it only requires one disk I/O to retrieve the query result from sparkey. However, as the number of queries increases, the query response time increases linearly, because the throughput of sparkey is bounded by disk I/O when the PPR index is larger than the main memory.
small hub set composition algorithm which approximates the PPR vectors for a computation since then. Approximation techniques have been proposed to speed up the PPR computation. For example, DrunkardMob [25], is designed for single-machine disk-based graph engines to minimize the I/O efficiency [4]. The other system, called DrunkardMob, is designed for the MapReduce model, which aims to optimize computational power [37, 15, 31, 32, 38]. Fogaras et al. [13] proposed the Monte-Carlo End-Point algorithm which is the first signed for a single machine, and are therefore limited by its restricted computational power [37, 15, 31, 32, 38].

The seminal work by Jeh and Widom [22] proposed the Hub Decomposition algorithm for web graphs. To compute (Personalized) PageRank with the vertex-centric programming model, and 3) it may require an offline phase and an online phase. However, in the offline phase, their approach needs to permute the adjacent matrix and precompute the QR decomposition of the matrix. For large matrix, this can be very time consuming. For example, even for a graph with only 0.35 million vertices and 1.4 million edges, the precomputed random walks for each vertex, and then stitch short random walks to answer online (Personalized) PageRank queries [5]. Howev, er this algorithm is not suitable for distributed graph engines for several reasons: (1) it requires random accesses to the graph data and precomputed index, (2) its algorithmic logic is not compatible with the vertex-centric programming model, and 3) it may require a large number of iterations.

Avrachenkov et al. [3] proposed a Monte-Carlo complete path algorithm for computing the global PageRank which also utilizes the full trajectory of each random walk. Bahmani et al. [5] extended the Monte-Carlo method in [3] to incremental graphs and Personalized PageRank. Their approach stores a small number of precomputed random walks for each vertex, and then stitch short random walks to answer online (Personalized) PageRank queries [5]. However, this algorithm is not suitable for distributed graph engines for several reasons: (1) it requires random accesses to the graph data and precomputed index, (2) its algorithmic logic is not compatible with the vertex-centric programming model, and 3) it may require a large number of iterations.

Buehrer and Chellapilla [11] designed a graph compression algorithm for web graphs. To compute (Personalized) PageRank or other random walk based computations, they proposed to first compress the web graph, and then perform the computation on.
the compressed graph using the power iteration method. However, this approach has two disadvantages: (1) it cannot be extended to weighted graphs like our approach; (2) its efficiency can degrade significantly on social networks since compressing social graphs is usually more costly and the compression ratios on social networks are much worse compared to web graphs [8]. Also, to our knowledge, there is no available implementation of the algorithm in [11].

6. CONCLUSION
In this paper, we present our system, PowerWalk, which adopts a novel framework for online PPR computation on distributed graph engines. PowerWalk uses the MCFP algorithm to compute a light-weight PPR index, and then uses the VERD algorithm to compute PPR vectors by a linear combination of the PPR index in an online manner. Our evaluation shows that our MCFP algorithm provides a more accurate approximation compared to the existing Monte-Carlo End-Point algorithm [13] by simulating the same number of random walks. Extensive experiments on two large-scale real-world graphs show that PowerWalk is quite scalable in balancing offline reprocessing and online query, and is capable of computing tens of thousands of PPR vectors in an order of seconds. We believe PowerWalk can be readily extended for many large-scale random walk models [29,42].

7. REFERENCES

APPENDIX
A. PROOF OF THEOREM 2.1
To prove Theorem 2.1 we first need the following result on random walks [16, Theorem 2.1].
**Theorem A.1** (The Chernoff bound for Random Walks). Consider the random walk following the transition matrix $P$ on a graph $G = (V, E)$ with initial distribution $q$. Let $S \subset V$, and let $t_n$ be the number of visits to $S$ in $n$ steps. For any $\gamma \geq 0$,

$$\Pr \left[ t_n - n \sum_{v \in S} \pi(v) \geq \gamma \right] \leq \left( 1 + \frac{n\gamma\epsilon}{10n} \right) \sqrt{\frac{\sum_{w \in V} \left( \frac{q(w)}{\sqrt{\pi(w)}} \right)^2}{\sum_{w \in V} \frac{q(w)}{\pi(w)}}} \exp \left( -\frac{\epsilon \gamma^2}{20n} \right),$$

where $\pi$ is the stationary distribution and $\epsilon$ is the eigenvalue gap of stochastic matrix $P$.

**Proof of Theorem A.1** For a given vertex $u$, we define the matrix $P$ as

$$P = (1 - c)A + cI e_u,$$

where $I$ is a column vector whose elements are all 1’s. It is easy to see that $P$ is row-stochastic and Equation (2.1) can be rewritten as:

$$p_u = p_u P.$$

As discussed in Section 2, the $R$ random walks in Algorithm 1 can be seen as one long random walk from vertex $u$ following the transition matrix $P$. Since the average length of each walk is $1/c$, the total length of all $R$ random walks or the length of the long random walk is $n \approx R/c$. By using Theorem A.1 we have

$$\Pr [ \hat{p}_u(v) - p_u(v) \geq \gamma ] = \Pr [ x_u(v) - np_u(v) \geq n\gamma ] \leq \left( 1 + \frac{n\gamma\epsilon}{10n} \right) \sqrt{\frac{\sum_{w \in V} \left( \frac{q(w)}{\sqrt{\pi(w)}} \right)^2}{\sum_{w \in V} \frac{q(w)}{\pi(w)}}} \exp \left( -\frac{\epsilon \gamma^2}{20n} \right).$$

(A.1)

Since $P$ is a row-stochastic matrix, the largest eigenvalue $\lambda_1$ is 1, while the second largest eigenvalue, denoted as $\lambda_2$, is $1 - c$ [20]. So the eigenvalue gap $\epsilon$ is $\lambda_1 - \lambda_2 = c$. Since the merged long random walk always starts from $u$, the initial distribution is $e_u$.

From Equation (2.1), it is easy to see that $p_u(u) \geq c$. Hence, we have

$$\sqrt{\sum_{w \in V} \left( \frac{q(w)}{\sqrt{\pi(w)}} \right)^2} = \frac{1}{\sqrt{c}}$$

Now we can apply the above results into Equation (A.1), and we have

$$\Pr [ \hat{p}_u(v) - p_u(v) \geq \gamma ] \leq \left( 1 + \frac{n\gamma\epsilon}{10n} \right) \frac{1}{\sqrt{c}} \exp \left( -\frac{c(n\gamma)^2}{20n} \right) = \frac{1}{\sqrt{c}} \left( 1 + \frac{\gamma\epsilon}{10} \right) \exp \left( -\frac{\gamma^2 R}{20} \right),$$

which completes the bound of over-estimation.

**Applying Theorem A.1** to the set $V \setminus \{v\}$ gives the same bound for the probability of under-estimation

$$\Pr [ \hat{p}_u(v) - p_u(v) \leq -\gamma ] = \Pr [ x_u(v) - np_u(v) \leq -n\gamma ] = \Pr \left[ \sum_{w \in V \setminus \{v\}} x_u(w) - n \sum_{w \in V \setminus \{v\}} p_u(w) \geq n\gamma \right] \leq \frac{1}{\sqrt{c}} \left( 1 + \frac{\gamma\epsilon}{10} \right) \exp \left( -\frac{\gamma^2 R}{20} \right),$$

which completes the proof.

**B. PROOF OF THEOREM 2.3**

**Proof.** To prove Theorem 2.3 we claim that for $k = 0, 1, \ldots, T$

$$\text{vc-decomp}(u, T) = s_u^{(T-k)} + \sum_{v \in V} f_u^{(T-k)}(v) \text{decomp}(u, k).$$

The proof is by induction on $k$. The case for $k = 0$ is obvious. Suppose the claim is true for some $k \geq 0$. We show it holds for $k + 1$ as follows:

$$\text{vc-decomp}(u, T) = s_u^{(T-k)} + \sum_{v \in V} f_u^{(T-k)}(v) \text{decomp}(u, k)$$

$$= \left( s_u^{(T-k)} - k \right) + \sum_{v \in V} c \cdot f_u^{(T-k)}(v) (v') \text{decomp}(u, k)$$

$$+ \sum_{v \in V} \left( \sum_{w \in V} \frac{1 - c}{\text{deg}(v)} \sum_{v \in \text{deg}(w)} f_u^{(T-k)}(w) (v') \text{decomp}(w, k) \right)$$

$$= s_u^{(T-k-1)} + \sum_{v \in V} c \cdot f_u^{(T-k-1)}(w) (v') \text{decomp}(u, k)$$

$$+ \sum_{v \in V} \left( \sum_{w \in \text{deg}(w)} f_u^{(T-k-1)}(w) (v') \text{decomp}(w, k) \right)$$

$$= s_u^{(T-k-1)} + \sum_{v \in V} f_u^{(T-k-1)}(w) (v') \text{decomp}(u, k)$$

$$= s_u^{(T-k-1)} + \sum_{v \in V} f_u^{(T-k-1)}(w) (v') \text{decomp}(u, k)$$

When $k = T$, we have

$$\text{vc-decomp}(u, T) = s_u^{(T-k)} + \sum_{v \in V} f_u^{(T-k)}(v) \text{decomp}(v, T)$$

$$= \text{decomp}(u, T).$$

□