Noname manuscript No.
(will be inserted by the editor)

Scheduled Approximation and Incremental Enhancement for Accuracy-aware Personalized PageRank

Fanwei Zhu · Yuan Fang · Kevin C. Chang · Jing Ying

the date of receipt and acceptance should be inserted later

Abstract As Personalized PageRank has been widely leveraged for ranking on a graph, the efficient computation of Personalized PageRank Vector (PPV) becomes a prominent issue. In this paper, we propose FastPPV, an approximate PPV computation algorithm that is *incremental* and *accuracy-aware*. Our approach hinges on a novel paradigm of *scheduled approximation*: the computation is partitioned and scheduled for processing in an "organized" way, such that we can gradually improve our PPV estimation in an incremental manner, and quantify the accuracy of our approximation at query time. Guided by this principle, we develop an efficient hub based realization, where we adopt the metric of *hub-length* to partition and schedule random walk tours so that the approximation error reduces exponentially over iterations. In addition, as tours are segmented by hubs, the shared substructures between different tours (around the same hub) can be reused to speed up query processing both within and across iterations. Given the key roles played by the hubs, we further investigate the problem of hub selection. In particular, we develop a conceptual model to select hubs based on the two desirable properties of hubs–sharing and discriminating, and present several different strategies to realize the conceptual model. Finally, we evaluate FastPPV over two real-world graphs, and show that it not only significantly outperforms two state-of-the-art baselines in both online and offline phrases, but also scale well on larger graphs. In particular, we are able to achieve near-constant time online query processing irrespective of graph size.

This material is based upon work partially supported by NSF Grant IIS 1018723, the Advanced Digital Sciences Center of the University of Illinois at Urbana-Champaign, and the Agency for Science, Technology and Research of Singapore. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the funding agencies.

F. Zhu · J. Ying
Zhejiang University City College, Hangzhou, China,
E-mail: {zhufw, yingj}@zucc.edu.cn

Y. Fang
Institute for Infocomm Research, Singapore
E-mail: yfang@i2r.a-star.edu.sg

K. C. Chang
University of Illinois at Urbana-Champaign, USA
Advanced Digital Sciences Center, Singapore
E-mail: kcchang@illinois.edu

1 Introduction

Graphs are ubiquitous in the real-world, such as the Web, social networks and entity-relationship graphs, calling for solutions to ranking on a graph. Formally, a graph $G = (V, E)$ is represented by a set of nodes V and edges E. As each edge embeds certain semantic relationship between the nodes, given a node $q \in V$ as the query, what are the nodes *relevant* to q through the edges in E? Here, the *input* is a query q, and the *output* is a ranked list of nodes in V. We motivate such ranking with two example scenarios.

Scenario 1: Bibliographic search. Consider a bibliographic network with interconnected nodes such as papers, venues and authors. Given a paper, who are the best matching experts to review it? In this case, the input query is a paper node, and the output is a ranking over the author nodes in the network.

Scenario 2: Social recommendation. Consider a social network with users as nodes, which are connected by their friendships. Given a user in the network, how can we recommend some potential friends to her? Taking the user node as the input query, a ranking over all the other user nodes can be leveraged for the recommendation task.

In the above scenarios, the rankings are specific to the dynamic queries, reflecting the "relevance" of nodes to the query node. As a well-studied graph ranking algorithm, *Personalized PageRank* [19,16] is effective in calculating such query-specific relevance based on the link structure of the graph. In this paper, we study the efficiency aspect of Personalized PageRank.

Background on Personalized PageRank. Personalized PageRank is an extension of the famous *PageRank* algorithm [19], both of which are based on a random surfer model.

To understand Personalized PageRank, we first review the original PageRank briefly. A random surfer starts at any node on the graph. At each step, with a probability of $1-\alpha$ the surfer moves to a neighboring node randomly, and with a probability of α she gets bored and teleports to a random node on the graph. This process is repeated until the random walk converges to a steady state. The stationary probability of the surfer at each node is taken as the PageRank score of the node. However, this form of score is purely based on the static link structure, indicating the overall "popularity" of each node on the graph, without tailoring to a specific query node.

In contrast, Personalized PageRank enables query-sensitive ranking, in the sense that we can specify a query node to obtain a "personalized" ranking accordingly. It is based on the same random surfer model of the original PageRank, except when the surfer teleports, she always prefers the query node q. Specifically, at each step, with probability α the surfer teleports to q instead of a random node, thus visiting the neighborhood of q more frequently. Thus, the stationary distribution, called a Personalized PageRank Vector (PPV), is biased towards q and its neighborhood, which can be interpreted as a popularity or relevance metric specific to q. We denote the PPV *w.r.t.* a query node q by $\mathbf{r}_q$, and $\mathbf{r}_q(p)$ refers to the entry corresponding to node p in $\mathbf{r}_q$, *i.e.*, p's score *w.r.t.* q.

More generally, a query q can comprise multiple nodes on the graph, such that in the teleportation the surfer can jump to any node in q. Fortunately, the computation for a multi-node query is no more difficult than for a single-node query due to the *Linearity Theorem* [16,12,10], as the PPV *w.r.t.* a multi-node query is a simple linear combination of the individual PPV *w.r.t.* each node in the query. Hence, our discussion only covers single-node queries.

Challenges in efficiency. Unfortunately, computing an *exact* PPV is, in general, infeasible even on a moderately large graph due to the prohibitive time or space cost [12,16]. To make exact computation manageable, early works [15,16] restrict personalization (*i.e.*, the query) to only some selected nodes. While such partial personalization is in some cases acceptable, most applications demand full personalization, which supports any arbitrary node as queries. Thus, some recent efforts [16,12,6,10,11] propose full personalization methods for *approximate* PPVs. They trade accuracy for faster query processing by reducing the computation in consideration online, as well as resorting to partial precomputation offline, which we will further elaborate in Sect. 2. However, in these schemes, once the offline precomputation is completed at a predetermined approximation level, the trade-off between efficiency and accuracy cannot be easily controlled dynamically.

Our proposal. In this paper, we present FastPPV, an approximate algorithm for computing fully personalized PPV. To highlight, it features *incremental* and *accuracy-aware* query processing, which means we can control the trade-off between efficiency and accuracy online. The key insight to achieve such a control hinges on the novel concept of *scheduled approximation*—we "organize" the random walk paths to be considered in some meaningful layers, such that the approximation can be incremented layer by layer, and more layers render better accuracy.

In particular, we develop this scheduled approximation upon an existing concept called *inverse P-distance* [16]. As shown previously [16], a node p's score in the PPV *w.r.t.* a query node q equals to the inverse P-distance from q to p, which is the *reachability* from q to p through all possible tours (*i.e.*, paths):

$$\mathbf{r}_q(p) \equiv \sum_{t \in \{q \rightsquigarrow p\}} R(t), \tag{1}$$

where a tour $t \in \{q \rightsquigarrow p\}$ is a sequence of edges from q to p that may contain cycles. $R(t)$, the *reachability* of t, is the probability of reaching p from q through tour t in a random walk. For a tour t of the form $v_0 \to v_1 \to \cdots \to v_{\mathcal{L}(t)}$ with length $\mathcal{L}(t)$,

$$R(t) \triangleq (1-\alpha)^{\mathcal{L}(t)} \cdot \alpha \cdot \prod_{i=0}^{\mathcal{L}(t)-1} \frac{1}{|\mathrm{Out}(v_i)|}, \tag{2}$$

where $\alpha \in (0,1)$ is the teleporting probability in the random surfer model, and $|\mathrm{Out}(v_i)|$ denotes the out-degree of node v_i.

We note that inverse P-distance was previously explored [16] to decompose the computation of a PPV. Specifically, they use inverse P-distance to compute some

PPV components for a restricted set of nodes, which are then assembled to obtain the final PPVs *w.r.t.* those restricted nodes. Thus, their goal is to compute exact PPVs, but only for a fixed subset of nodes, lacking full personalization. While identified as their future work [16], devising an approximate algorithm for full personalization, solely based on their original use of inverse P-distance without exploiting other properties, appears implausible. In this paper, we solve this problem by making a new observation on inverse P-distance—the tours in Eq. 1 are not equally important in contributing to the computation. This observation prompts us to investigate the novel principle of scheduled approximation by *partitioning and prioritizing* tours, which leverages inverse P-distance in a distinct way, as we discuss next.

Principle (Sect. 3). We *partition* the set T of all tours involved in inverse P-distance (Eq. 1) into disjoint subsets $T = T^0 \cup \cdots \cup T^\eta$ according to their contribution to the computation, and *prioritize* them to tackle a more "important" partition T^i earlier. While simple, this partition-and-prioritize principle has remained unexplored for PPV computation to this date, and possesses two ideal properties:

- *Incremental.* FastPPV processes tours partition by partition, starting from T^0. In iteration-i, it covers tours in T^i to compute an increment $\hat{\mathbf{r}}_q^i$, adding to the overall estimate $\hat{\mathbf{r}}_q = \hat{\mathbf{r}}_q^0 + \cdots + \hat{\mathbf{r}}_q^i$. As it covers more partitions, the error (in terms of L1 norm) monotonically decreases, and $\hat{\mathbf{r}}_q$ asymptotically approaches $\mathbf{r}_q$.
- *Accuracy-aware.* In each iteration, we show that the current L1 error is determinable using only the current estimate, even without knowing the exact PPV. Thus, this error can be utilized as a stopping condition to control the trade-off between efficiency and accuracy at query time.

Realization (Sect. 4). To realize the basic principle of partitioning tours, we propose a novel notion, *hub length.* Given some *hub nodes* H selected from V, we measure the hub length of a tour t, or $\mathcal{L}_h(t)$, as the number of hubs traversed by t. Then, we partition each T^i to contain tours of $\mathcal{L}_h(t) = i$. With a carefully selected set of hubs (which will be covered later), the goal is to achieve the two desirable properties in the following to enable efficient computation:

- *Discriminating.* By choosing nodes of high "decaying power" to reduce the reachability of t (Eq. 2), the fewer hubs t contains, the larger t's reachability becomes in general. Thus, tours in an earlier partition T^i (with i hubs) tend to contribute more than those in a later partition T^{k+m} (with more than k hubs), allowing us to efficiently focus on the first few partitions that are more important for an accurate estimation. We formally prove an error bound that decreases exponentially as more partitions are covered.
- *Sharing*: By choosing "popular" nodes on the graph as hubs such that they are more likely to be utilized by queries, different tours will share the same hubs. This sharing enables the reuse of common substructures to speed up computation. First, as the tour segments between hub nodes are shared, we can thus precompute and index their reachabilities, which we call *prime PPVs*, as *building blocks* to assemble an arbitrary tour at query time. Second, as we build partitions by hub length, it can be shown that tours in partition T^i simply extend those in T^{i-1} as prefixes, and thus successive iterations can reuse these prefixes.

Hub selection (Sect. 5). In the realization of FastPPV, the choice of hubs H are crucial to efficient computation. To select a useful set of hubs, we develop a conceptual model to integrate the two desirable properties in our realization—*sharing* and *discriminating*, by assessing the popularity and decaying power of the candidate hubs. We further explore several strategies to realize the conceptual model:

- *Naïve.* As a first attempt, we do not assume any particular query distribution, and select each hub candidate independently of others.
- *Query distribution-aware.* Given that queries often have a skewed distribution, we now consider the effect of query distribution on hub selection.
- *Community-based.* Lastly, we observe that discriminating power of the hub nodes are not independent of each other. In particular, the marginal (or additional) discriminating power that a candidate hub h can offer diminishes as more hubs lie in the "neighborhood" of h. We propose to approximate these neighborhoods by the communities in the graph [9], and diminish the marginal discriminating power of a candidate hub if many hubs already exist in its community.

Overall framework (Sect. 6). Upon the hub-based realization, we devise an overall framework for FastPPV, consisting of two stages:

- *Offline precomputation*: We identify a desirable set of nodes as hubs, and precompute their prime PPVs as building blocks for online processing.

- *Online query processing*: We start from the tours of hub length 0 in T^0, and further process tours of increasing hub length in an iterative manner. Within each iteration, precomputed building blocks can be reused; across iterations, prefixes can be shared.

Through our scheduled approximation, online processing is incremental (by the ability to handle tours partition by partition), accuracy-aware (by the ability to measure error), and fast (by the ability to reuse computation). Furthermore, as real-world graphs are often too large to entirely fit into the main memory, we also propose a disk-based implementation.

Empirical evaluation (Sect. 7). Finally, we conduct extensive experiments on two real-world datasets, the bibliographic network DBLP, and the social network LiveJournal. We compare FastPPV with two competitive baselines [11,12], and find out that FastPPV significantly outperforms them in both online and offline phases. More importantly, we are able to demonstrate the scalability of FastPPV on growing graphs. In particular, FastPPV can achieve a near-constant time query processing irrespective of graph size, through only a linear increase in the offline precomputation costs.

2 Related work

While Personalized PageRank [19,16] enables a personalized or query dependent view of PageRank, its computation can be prohibitively expensive in time or space, for not only online but also offline scenarios. Even on a moderately large graph, it is infeasible to compute PPVs online using the naïve iterative method. Alternatively, even with the Linearity Theorem [16], naïve precomputation of the exact PPV *w.r.t.* every node on the graph (*i.e.*, full personalization) is time consuming and requires at least $\Omega(|V|^2)$ bits to store, which is quadratic in $|V|$ or the number of nodes on the graph. It can be shown that the quadratic space complexity holds no matter how clever the compression scheme is [12].

Thus, designing efficient algorithms for personalized PageRank has become an important research area. While earlier work pursues exact computation by supporting partial personalization (*i.e.*, only a subset of nodes can be used in a query), our work aligns with more recent developments that aim at full personalization (*i.e.*, any node can be used in a query) at the cost of accuracy.

Exact, partial personalization. Haveliwala *et al.* first proposed topic-sensitive PageRank [15], which only precomputes 16 PPVs—each corresponds to a top level category in the Open Directory Project[1]. With the Linearity Theorem [16], finer-grained personalization can be supported, *e.g.*, in hub decomposition [16], intelligent surfer [23] and ObjectRank [5]. Despite this, full personalization is still infeasible on large graphs.

Approximate, full personalization. To achieve full personalization, most efforts resort to *approximate* computation instead. Intuitively and informally, the PPV *w.r.t.* a query q is a measure over random-walk paths starting from q. Thus, most of the existing approximation approaches can be perceived as a reduction in the total number of paths in their computation. First, only *hub-pivoted paths* that pass through some important "hub" nodes are considered, *e.g.*, Web Skeleton [16]. Second, only *sampled paths* using a Monte Carlo simulation are considered, *e.g.*, [12,4,3]. Third, only *neighborhood paths* that are within some "radius" around q are considered, *e.g.*, Bookmark Coloring [6] and the HubRank family [10,21,11].

For such approximation methods, the accuracy of estimate and how to control it is always one of the core issues. As for the *hub-pivoted paths* methods, the choice of hubs would directly affect the accuracy. Jeh & Widom [16] discuss how to choose a good hub set H such that the PPV computed only on tours passing through H would be a good approximation, but they do not provide a formal gurantee on the accuracy of approximation. The *sampled paths* methods use a precomputed database of sampled paths called *fingerprints* for estimation. Fogaras *et al.* [12] prove that the accuracy could be improved by indexing more fingerprints during offline stage and it would eventually converges to the exact PPV with sufficiently large index. Bahmani *et al.* [4,3] also leverage the notion of fingerprints for approximation, but opportunistically determine whether to utilize some precomputed fingerprints or take on-the-fly walks to answer a query. Therefore, the accuracy of estimation can also be controlled by specifying the number of online walks. In *neighborhood paths* methods, PPVs are approximated on a neighboring subgraph of the query, which is bordered by hubs. The precomputed PPV of border hubs are used to compute the PPV of the query in some iterative way. However, as the size of subgraphs are determined by the preselected hubs, it cannot control the accuracy dynamically.

In addition, some techniques leverage additional properties, such as the block structure of the web [17]. A few top-K methods have also been explored [14,13], which often rely on bounds to identify the top-K nodes without an actual estimate on node scores.

[1] http://www.dmoz.org/

Comparison to our work. Our work also build upon the idea of reducing the number of paths for approximation. However, instead of focusing on some neighborhood around the query node [16,6,10,21,11] or randomly sampled paths [12,4,3], we introduce a novel approach of *structured approximation* that systematically organizes the entire computation space (*i.e.*, all the tours) by importance, and then gradually process it in an orderly way to achieve an incrementally enhanced approximation.

- Organizing the tour space. We recognize that tours are of different importance in the computation, and thus partition them into different sets so that tours in the same partition sharing similar importance.
- Scheduling the processing of tours. In the partitioned tour space, tours can be scheduled into computation according to their importance, partition by partition, gradually covering the entire computation space.

Such structured approximation leads to two distinct properties of FastPPV, compared to existing approximation methods.

First, FastPPV is "important-first," prioritizing more important tours to generate a fast yet good initial estimate. As tours are organized by importance, we are able to schedule more important tours into computation earlier, thus ensuring a good estimation even if we stop after the initial few iterations. While many previous approaches can also achieve more accurate results by handling more tours, they are not necessarily important-first. For example, Monte Carlo methods [12, 4,3] achieve higher accuracy by simulating more random walk tours on the graph, but there is no guarantee that the most important tours will always be sampled first. Thus, their approximation tends to suffer more than FastPPV if computation stops earlier. Other methods [6,10,21,11] obtain higher accuracy by expanding an active subgraph around the query. Although a smaller subgraph around the query intuitively contain more important tours, the exact correspondence between the structure of the subgraph and the importance of the tours remains unclear.

Second, FastPPV is "accuracy-aware," knowing how accurate the current estimate is during query time (of course without the knowledge of the exact PPV). Our structured approximation, which gradually covers the entire tour space, means that its estimation is monotonic (see Sect. 3). Monotonic estimation implies that its accuracy can be easily established even though the exact PPV is unknown. Thus, FastPPV is always aware of the accuracy of its current estimate during query time, and can terminate when the desired accuracy is obtained. We note that the accuracy-aware property due to monotonic estimation has not been investigated in the literature. While other approaches also allow for dynamic trade-off between accuracy and query time online, in general they are not monotonic in nature and thus are unaware of their current accuracy during query processing. In particular, Monte Carlo methods [12,4, 3] do not generate monotonic estimates given randomly sampled tours.

Apart from the key principle of structured approximation, we also employ the concepts of hub and subgraph, although their purposes differ from what has been explored in existing studies such as BCA and the HubRank family [6,11].

On the one hand, FastPPV uses hubs to quantify the importance of tours, in contrast to previous approaches which precompute a PPV for every hub. In FastPPV, the importance of a tour is quantified by its *hub length* (see Sect. 4), which means our hubs serve a different purpose—to partition and prioritize computation, rather than to directly provide precomputed PPVs. As such, our offline precomputation is significantly cheaper, since we do not need to compute the PPVs for hubs over the entire graph.

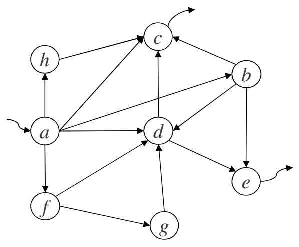
On the other hand, FastPPV uses subgraphs around the hubs as "building blocks" to extend tours for incremental enhancement, in contrast to previous methods which directly estimate the PPV on a subgraph around the query node. Instead, we assemble subgraphs online as a compact way of building up more tours during incremental enhancement. Thus, queries can be processed efficiently by precomputing and reusing popular building blocks.

3 Principle: Scheduled Approximation

In this section, we propose the general principle of a "scheduled" PPV approximation method, which enables *incremental* and *accuracy-aware* query processing.

Running example. As our motivating example, we introduce a toy graph $G = (V, E)$ in Fig. 1(a), where $V = \{a, b, \ldots, h\}$ and $E = \{(a, b), (a, d), \ldots\}$. To simplify discussion, the example graph is unweighted and contains no cycles, although our framework works for a general graph with cycles.

Suppose the query node is a. By Eq. 1, $\mathbf{r}_a$ the PPV *w.r.t.* a captures the *reachability* from a to each node in G. Consider the personalized PR score $\mathbf{r}_a(c)$ for a specific node c (*i.e.*, the reachability from a to c), which can be computed by summing up the reachability of 7 tours, as illustrated in Fig. 1(b). On a large graph, computing the reachability for all tours between each pair of nodes would cause serious efficiency issues. For-



(a) Graph G

Tours (a to c)	Reachability
t_1: $a \to c$	$R(t_1) = 0.0255$
t_2: $a \to h \to c$	$R(t_2) = 0.0216$
t_3: $a \to d \to c$	$R(t_3) = 0.0108$
t_4: $a \to b \to c$	$R(t_4) = 0.0072$
t_5: $a \to f \to d \to c$	$R(t_5) = 0.0046$
t_6: $a \to b \to d \to c$	$R(t_6) = 0.0046$
t_7: $a \to f \to g \to d \to c$	$R(t_7) = 0.0017$

$\Big\} \; \mathbf{r}_a(c)$

(b) $\mathbf{r}_a(c)$ computation (with teleporting prob. $\alpha = 0.15$)

Fig. 1: PPV computation example.

Partitioning tours		Prioritizing computation		
Tours	Sets	I. Computing over T^0	II. Computing over $T^0 \cup T^1$	III. Computing over $T^0 \cup T^1 \cup T^2$
t_9: $a \to b$	T^0 high reachability	$\hat{\mathbf{r}}_a^{T^0}$	$\hat{\mathbf{r}}_a^{T^0 \cup T^1}$	$\hat{\mathbf{r}}_a^{T^0 \cup T^1 \cup T^2}$
t_2: $a \to f$		c 0.0471	c 0.0579	c 0.0868
t_{10}: $a \to d$		d 0.0255	d 0.0444	d 0.0522
t_1: $a \to c$		b 0.0255	b 0.0255	b 0.0255
t_{11}: $a \to h$		f 0.0255	f 0.0255	f 0.0255
t_2: $a \to h \to c$		h 0.0255	h 0.0255	h 0.0255
t_{12}: $a \to f \to g$	T^1 medium reachability	▨	e 0.0108	e 0.0234
t_{13}: $a \to f \to d$		▨	g 0.0108	g 0.0108
t_{14}: $a \to f \to g \to d$		▨		$\Vert$
t_{15}: $a \to d \to e$		▨		exact PPV $\mathbf{r}_a$
t_3: $a \to d \to c$		▨		
t_{16}: $a \to b \to d$	T^2 low reachability	▨	▨	
t_4: $a \to b \to c$		▨	▨	
t_{17}: $a \to b \to e$		▨	▨	
t_{18}: $a \to b \to d \to e$		▨	▨	
t_6: $a \to b \to d \to c$		▨	▨	
t_{19}: $a \to f \to d \to e$		▨	▨	
t_{20}: $a \to f \to g \to d \to e$		▨	▨	
t_5: $a \to f \to d \to c$		▨	▨	
t_8: $a \to f \to g \to d \to c$		▨	▨	

Fig. 2: Scheduled approximation example.

tunately, we have observed two facts that motivate an efficient PPV computation approach.

- *Some tours are more important than others in PPV computation.* A tour with higher reachability (*e.g.*, t_1) will rank its destination node (*e.g.*, c) highly by contributing more to the computation of the final score (*e.g.*, $\mathbf{r}_a(c)$).
- *Covering more tours in the computation would improve the accuracy.* For instance, if we handle more tours from t_1 to t_7, the cumulative reachability would be closer to the exact $\mathbf{r}_a(c)$.

The above two observations lead to the key insights of a scheduled approximation approach, with two components in the following:

- *Partitioning tours.* Instead of treating all tours equally, we first partition them into different *tour sets* according to their importance *w.r.t.* the query node. A partition of a full set of tours T is a set of disjoint subsets $T^0, \ldots, T^\eta$, where $T = T^0 \cup \ldots \cup T^\eta$ and $T^i \cap T^j = \emptyset, \forall i \neq j$.
- *Prioritizing computations.* Given a partition of tours $T = T^0 \cup \ldots \cup T^\eta$, we exploit the varying contribution of different tour sets, and schedule them for a prioritized PPV approximation—the most important set T^0 is traversed first for a fast estimate $\hat{\mathbf{r}}_q^{T^0}$ (*i.e.*, the aggregated reachability of tours in T^0 only), while less important ones are handled later to improve the accuracy incrementally. Note that we use $\hat{\mathbf{r}}_q$ to denote an estimated PPV to distinguish it from the exact one $\mathbf{r}_q$.

To be concrete, consider the graph G in Fig. 1(a). Suppose a is the query node and, for the purpose of illustration, we magically have the reachability of each tour at hand. Then, as Fig. 2 shows, we can partition the tours starting from a, $T = \{t_1, \ldots, t_{20}\}$, into some (say three) disjoint tour sets with decreasing importance by their reachability range: T^0, T^1, T^2. Note that in real scenarios, we do not have the reachability of each tour beforehand; we will discuss a practical partitioning strategy in Sect. 4.

Now, to prioritize computation, we initially consider the most important set T^0 only for an estimated PPV $\hat{\mathbf{r}}_a^{T^0}$. According to $\hat{\mathbf{r}}_a^{T^0}$, the most relevant nodes to a (*i.e.*, c) have already been identified correctly. Subsequently, when T^1 is added, we obtain an enhanced estimate $\hat{\mathbf{r}}_a^{T^0 \cup T^1}$, which ranks the top five nodes c, d, b, f, h perfectly. Finally, when the tours in T^2 are also included, all the tours starting from a are covered, achieving the exact PPV $\mathbf{r}_a$.

This example illustrates a well-scheduled PPV computation process. First, T^0 is the top consideration since through T^0 most of the nodes in G, in particular the important ones, would be ranked. Next, T^1 and T^2 are successively included to gradually improve our estimation. Towards the concept of scheduled computation, we propose an incremental query processing, which computes PPVs in a progressive manner where more time will render higher accuracy.

Incremental query processing. We estimate the PPV through multiple iterations, with each iteration handling an additional tour set, enhancing the overall approximation iteration by iteration.

More formally, given a partition of tours $T = T^0 \cup \ldots \cup T^\eta$ with decreasing importance, in iteration-i, a *PPV increment* $\hat{\mathbf{r}}_q^{T^i}$ is computed over the i-th important tour set T^i. For brevity, we also denote the increment by $\hat{\mathbf{r}}_q^i$. The overall approximation is the summation of all PPV increments from all iteration so far.

That is, after iteration-k, we obtain an approximate PPV $\hat{\mathbf{r}}_q^{(k)}$:

$$\hat{\mathbf{r}}_q^{(k)} = \hat{\mathbf{r}}_q^{T^0 \cup T^1 \cup \ldots \cup T^k} = \sum_{i=0}^{k} \hat{\mathbf{r}}_q^i \tag{3}$$

Note that in $\hat{\mathbf{r}}_q^{(k)}$, the superscript (k) is enclosed in parentheses to mean that it is cumulative from T^0 to T^k, while the superscript in each PPV increment $\hat{\mathbf{r}}_q^i$ has no parentheses.

Such an incremental process enables flexible trade-off of efficiency and accuracy. As we process more iterations, the accuracy of approximation is gradually enhanced and if all tour sets are processed, an exact PPV is obtained. The reason is quite obvious. For a disjoint partition $T = T^0 \cup \ldots \cup T^\eta$, in iteration-$i$, tours in T^i are included in the computation. Thus more iterations will tackle more tour sets, and after iteration-η, each tour in T is covered for exactly once. This property is formalized by the following theorem.

Theorem 1 *Given a query node q, let T be all tours starting from q and $T^0, \ldots, T^\eta$ be a partition of T. The estimated PPV score of any node $p \in V$ monotonically enhances with more iterations and eventually equals the exact one after iteration-η:*

$$\hat{\mathbf{r}}_q^{(0)}(p) \le \hat{\mathbf{r}}_q^{(1)}(p) \le \ldots \le \hat{\mathbf{r}}_q^{(\eta)}(p) = \mathbf{r}_q(p) \tag{4}$$

Generally, the graph can be cyclic, which contains a countably infinite number of tours. Thus, the partitioning might result in an infinite number of tour sets (*i.e.*, $\eta \to \infty$) such that we need infinite iterations to achieve the exact PPV. However, we can expect an approximation which is arbitrarily accurate with sufficient iterations. In Sect. 4, given our specific partitioning and prioritizing methods, we would derive an error bound that is consistent with this expectation.

Accuracy-aware approximation. Due to the nature of our incremental processing, after each iteration, we can easily compute the L1 error of our estimation so far, even without knowing the exact PPV $\mathbf{r}_q$. The L1 error after iteration-k is defined as follows:

$$\varphi^{(k)} \triangleq \left\| \mathbf{r}_q - \hat{\mathbf{r}}_q^{(k)} \right\|_1 = \sum_{p \in V} \left| \mathbf{r}_q(p) - \hat{\mathbf{r}}_q^{(k)}(p) \right| \tag{5}$$

From Theorem 1, we know $\mathbf{r}_q(p) \ge \hat{\mathbf{r}}_q^{(k)}(p), \forall p, q \in V, \forall k \le \eta$. Together with the fact that $\sum_{p \in V} \mathbf{r}_q(p) = 1$ (since $\mathbf{r}_q$ is a probability distribution over V), Eq. 5 can be conveniently re-expressed as follows:

$$\varphi^{(k)} = \sum_{p \in V} \mathbf{r}_q(p) - \sum_{p \in V} \hat{\mathbf{r}}_q^{(k)}(p) = 1 - \sum_{p \in V} \hat{\mathbf{r}}_q^{(k)}(p) \tag{6}$$

The above equation provides a simple way to calculate $\varphi^{(k)}$ as the one's complement of the L1 norm of the current PPV estimate, even without the knowledge of the final exact PPV. Thus, during online query processing, we can measure the L1 error after each iteration to enable a user-controllable trade-off between accuracy and time, *e.g.*, by specifying an accuracy requirement in terms of the L1 error, or a time limit for query processing.

Furthermore, if we prioritize the tour sets appropriately, we can ensure that earlier iterations would bring in more improvement than later ones. Ideally, we should order the tour sets by their importance, or equivalently the sum of reachability of the constituent tours, *i.e.*, $\sum_{t \in T_0} R(t) \ge \ldots \ge \sum_{t \in T_\eta} R(t)$, which means that $\sum_{p \in V} \hat{\mathbf{r}}_q^0(p) \ge \ldots \ge \sum_{p \in V} \hat{\mathbf{r}}_q^\eta(p)$. By Eq. 6 and 3, this order will result in the largest reduction in L1 error after iteration-0, followed by iteration-1 and 2, and so on. Consequently, we can stop at an early iteration, yet still get the "most significant portion" out of the exact PPV. Formally, in Sect. 4, based on our actual partitioning and prioritizing strategy, we will prove an error bound that decreases exponentially as more iterations are processed.

However, while the principle is straightforward, the realization is challenging in two aspects:

- Challenge 1: *how can we partition and prioritize the tours?* In other words, how can we measure the importance of each tour? Naturally, we do not know the reachability of each tour beforehand—PPV computation is our ultimate goal, and thus it is impractical to partition the tours according to their reachability as we did in the example. We need a simple and unified metric, which can be efficiently applied to measure the importance of tours and is universally effective for different queries.
- Challenge 2: *how can we efficiently compute each PPV increment?* Computing each $\hat{\mathbf{r}}_q^i$ from scratch is not practical since it is expensive to naïvely sum up the reachability of all tours involved. In our previous example in Fig. 2, we observe large overlaps between tours in different sets. *E.g.*, t_{12}, t_{13}, t_{14} in T^1 share an edge $a \to f$, which is a tour t_2 in T^0. Thus, we can take advantage of these overlaps to efficiently calculate each $\hat{\mathbf{r}}_q^i(p)$.

4 Realization: FastPPV

As the next step, we tackle the two challenges in realizing the basic principle in Sect. 3. We seek an effective and simple partitioning-and-prioritizing metric, and an efficient algorithm for computing PPV increments.

To motivate both goals, let us take a deeper examination of our running example in Fig. 1. Observe that some nodes, like d, have two desirable properties for characterizing the importance of tours and enabling efficient PPV computations, respectively.

First, *Discriminating.* With many out-neighbors, d significantly decays the reachability of those tours passing through it, *i.e.*, it has a high "decaying power" due to Eq. 2. *E.g.*, for two resembling tours (with only one different node), $t_2 : a \to h \to c$ and $t_3 : a \to d \to c$, the reachability $R(t_3)$ is only 1/2 of $R(t_2)$ due to the high decaying power of node d on t_3.

Second, *Sharing.* As many tours pass through d, the segments (a sequence of edges) around d may be shared in different tours. *E.g.*, the segment $f \to g \to d$ is shared by three tours starting from a as shown in Fig. 2. We say that d is "popular."

We refer to nodes with such properties as *hub nodes*, because topologically they look like *hubs* in a network, at the center of different connections. While the notion of hubs has been explored previously [16,10,11], we stress that our hubs serve dual unique purposes—as a crucial response to the dual challenges raised earlier.

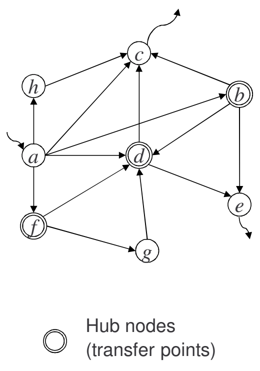
- *Tour set partitioning.* Hub nodes have high decaying power to discriminate tour importance and thus is a good criterion to partition tours (Challenge 1).
- *PPV increment computation.* Segments around hub nodes are shared by different tours and thus can be precomputed and reused to enable efficient computation (Challenge 2).

While hubs should possess the sharing and discriminating properties, we leave the concrete hub selection strategy to Sect. 5. For now, assuming a set of hubs are already selected, we will present an effective tour partition scheme (leveraging the discriminating property of hubs) in Sect. 4.1, and an efficient PPV computation algorithm (leveraging the sharing property of hubs) in Sect. 4.2.

4.1 Tour Partitioning and Prioritizing

As the first challenge of scheduled PPV approximation, we need a partition scheme which is effective in discriminating tour importance and can be efficiently applied on the fly.

To motivate, consider our example graph G, assuming $H = \{b, d, f\}$ is the hub set for G. For intuition, we make an analogy that G is a bus transportation network, in which each node is a city and each edge is a bus route connecting two cities. To facilitate long distance transportation, some particular cities (*i.e.*, hub nodes in our approach) where multiple bus lines pass through, marked in a double circle, are selected as *transfer points*, as Fig. 3(a) shows.



(a) Graph with hub nodes (Bus route network)

Tours	Sets	Priority
t_9: $a \to \underline{b}$	T^0 $\forall\, t \in T^0$: $\mathcal{L}_h(t) = 0$	Iteration-0
t_2: $a \to \underline{f}$		
t_{10}: $a \to \underline{d}$		
t_1: $a \to c$		
t_{11}: $a \to h$		
t_2: $a \to h \to c$		
t_{16}: $a \to \underline{b} \to \underline{d}$	T^1 $\forall\, t \in T^1$: $\mathcal{L}_h(t) = 1$	Iteration-1
t_4: $a \to \underline{b} \to c$		
t_{12}: $a \to \underline{f} \to g$		
t_{13}: $a \to \underline{f} \to \underline{d}$		
t_{14}: $a \to \underline{f} \to g \to \underline{d}$		
t_{15}: $a \to \underline{d} \to e$		
t_3: $a \to \underline{d} \to c$		
t_{18}: $a \to \underline{b} \to \underline{d} \to e$	T^2 $\forall\, t \in T^2$: $\mathcal{L}_h(t) = 2$	Iteration-2
t_6: $a \to \underline{b} \to \underline{d} \to c$		
t_{19}: $a \to \underline{f} \to \underline{d} \to e$		
t_{20}: $a \to \underline{f} \to g \to \underline{d} \to e$		
t_5: $a \to \underline{f} \to \underline{d} \to c$		
t_8: $a \to \underline{f} \to g \to \underline{d} \to c$		

(b) Partition & prioritize by hub length

Fig. 3: Hub length-based tour partitioning and prioritizing.

Now suppose a passenger is planning a trip from city a to c. Which route is most likely to be chosen? Apparently, taking a *direct bus route* (which does not pass through any transfer point) between a and c, *i.e.*, $a \to c$ or $a \to h \to c$ (here h is merely a "stop-over", not a transfer point) is most preferred, followed by making one transfer (*e.g.*, $a \to \underline{d} \to c$), and then two transfers (*e.g.*, $a \to \underline{f} \to \underline{d} \to c$). Note that, when writing a tour, we underscore a node to stress that it is a transfer point (or hub node).

Intuitively, just as people dislike routes that need many transfers, it is less likely to follow the tours containing more hub nodes in random walks, *i.e.*, these tours are less important in our prioritized PPV computation. More formally, each hub node would substantially decay the reachability of a tour passing through it due to its large out-degree. The more hubs a tour passing through, the less important the tour is.

Partitioning and prioritizing by hub length. We formally quantify the importance of a tour by the number of hubs it passes through, which we call *hub length.*

Definition 1 (Hub Length) Given a set of hub nodes H, for any tour t, the hub length of t, denoted by $\mathcal{L}_h(t)$, is the number of hub nodes in t, excluding the starting and ending nodes.

Given this hub length metric, partitioning tours is straightforward. Consider the example graph with $H = \{b, d, f\}$ in Fig. 3(a). The tours starting from a are partitioned into three sets: T^0, T^1 and T^2, with each containing tours of a distinct hub length—the hub length of every tour is 0 in T^0, 1 in T^1 and 2 inT^2, as shown in Fig. 3(b). These tour sets form a valid partition—they are pairwise disjoint and cover all tours starting from a. Furthermore, the importance of tours in different sets are decreasing from T^0 to T^2, which naturally shows a desired order for prioritized computation.

More generally, given a set of hub nodes, for a query node q, we can partition all the tours rooted at q into η sets $T = T^0 \cup T^1 \cup \ldots \cup T^\eta$ such that, for $i \in [0, \eta]$, $T^i = \{t \mid t \text{ starts at } q \wedge \mathcal{L}_h(t) = i\})$, where η is the maximal hub length of all tours in T. Given such a partition, for a prioritized incremental computation, to return better results earlier, as Sect. 3 explained, the sets with a shorter hub length are handled earlier, in the order T^0 to T^η. As Sect. 3 also explained, for a cyclic graph, η can be infinite. However, tours with large hub length contribute trivially and, thus, can be omitted for a good approximation.

Error bound. Based on our partitioning and prioritizing strategy, we further exploit the L1 error of our incremental approximation discussed in Sect. 3. In particular, we formally prove a bound for the L1 error in each iteration, which further implies two theoretically desirable properties. The proof of the theorem can be found in Appendix A.

Theorem 2 *After iteration-k, the L1 error $\varphi^{(k)}$ as defined in Eq. 5 satisfies the following bound:*

$$\varphi^{(k)} \leq (1-\alpha)^{k+2} \tag{7}$$

This bound exhibits two desirable properties. *First*, since $1-\alpha < 1$, $\lim_{k\to\infty} \varphi^{(k)} = 0$. *Second*, the rate of $\varphi^{(k)}$ approaching zero is exponential as k grows. In other words, an earlier iteration contributes exponentially more than a later one. Thus, we can stop early yet still obtain a good estimation.

As an example, for a typical $\alpha = 0.15$ [19], we have $\varphi^{(10)} \leq 0.143$, $\varphi^{(20)} \leq 0.028$ and $\varphi^{(30)} \leq 0.006$, which diminishes exponentially as k increases. We note that, as the proof of Theorem 2 builds upon, we bound the error of the overall reachability of $\cup_{i=0}^{k} T^i$ by that of $\cup_{i=0}^{k+1} S^i$. In practice, $\cup_{i=0}^{k} T^i$ contains many more tours than $\cup_{i=0}^{k+1} S^i$, which makes the error to converge even faster. Our experiments in Sect. 7 show that as few as three iterations yield a very accurate PPV.

4.2 Efficient PPV Increment Computation

We next tackle the second challenge—to efficiently compute the PPV increment $\hat{\mathbf{r}}_q^i$ in iteration-i, which aggregates the reachability of tours of hub length i. Towards its efficient realization, we analyze the structure of tours in aggregation and develop a tour assembly model– interestingly, since tours are built from segments, the aggregation of tours into PPV amount to the aggregation of PPVs of the constituent segments around hubs. Next, we propose the sharing and reusing of such "hub-segment PPVs" in aggregation and, the "overlapping" of aggregations in different rounds, to enable an efficient PPV-increment computation.

4.2.1 Structured Aggregation: Tour Assembly Model

To enable efficient computation of $\hat{\mathbf{r}}_q^i$, we develop a tour assembly model to aggregate the similar segments in different tours, so that the overall reachability can be aggregated by such structured components rather than each individual tour.

Recall the bus transportation analogy in Sect. 4.1 (Fig. 3(a)). Consider the possible itineraries with two transfers from a to c– $t_5 : a \to \underline{f} \to \underline{d} \to c$, $t_6 : a \to \underline{b} \to \underline{d} \to c$ and $t_7 : a \to \underline{f} \to g \to \underline{d} \to c$. We observe that all these itineraries can be constructed by three "direct bus routes": one from the source a and two from subsequent transfer points. *E.g.*, t_7 is built by $a \to \underline{f}$ (a direct route from a), and two direct routes $\underline{f} \to g \to \underline{d}$ and $\underline{d} \to c$ from transfer points f and d respectively.

Our $\hat{\mathbf{r}}_q^i$ computation shares the same insight with this analogy of building itineraries. For a query node q, by viewing each tour from q as an itinerary starting at q and going through hubs as making transfers, we can "assemble" the reachability of any tour by combining the reachability of a *direct segment* (*i.e.*, tours passing through no hubs) from q to its nearest hub node and then several direct segments from each hub on the tour. Specifically, let's examine the tours just mentioned (t_5, t_6, t_7). For each tour, we can calculate its reachability by assembling three direct segments as follows:

$$R(t_5) = \frac{1}{\alpha^2} \cdot R(a \to \underline{f}) \cdot R(\underline{f} \to \underline{d}) \cdot R(\underline{d} \to c)$$

$$R(t_6) = \frac{1}{\alpha^2} \cdot R(a \to \underline{b}) \cdot R(\underline{b} \to \underline{d}) \cdot R(\underline{d} \to c)$$

$$R(t_7) = \frac{1}{\alpha^2} \cdot R(a \to \underline{f}) \cdot R(\underline{f} \to g \to \underline{d}) \cdot R(\underline{d} \to c)$$

Note that by the reachability definition (Eq. 2), at the end of a segment, the random surfer would stop with a probability α. Thus, to continue the tour, we need to compensate a probability α at each "transfer," *i.e.*, the $\frac{1}{\alpha^2}$ term in our two-transfer example above.

With such an "assembly" of individual reachabilities, we now build a systematic understanding of assembling $\hat{\mathbf{r}}_q^i(p)$. As an example, we will consider the above tours from a to c, to assemble $\hat{\mathbf{r}}_a^2(c)$. The result can be derived, step by step, as Eq. 8 shows.

$$\begin{aligned}\hat{\mathbf{r}}_a^2(c) &\triangleq R(t_5) + R(t_6) + R(t_7)\\ &= \big(R(t_5) + R(t_7)\big) + R(t_6)\\ &=^1 \frac{1}{\alpha^2} \cdot \hat{\mathbf{r}}_a^0(f) \cdot \hat{\mathbf{r}}_f^0(d) \cdot \hat{\mathbf{r}}_d^0(c) + \frac{1}{\alpha^2} \cdot \hat{\mathbf{r}}_a^0(b) \cdot \hat{\mathbf{r}}_b^0(d) \cdot \hat{\mathbf{r}}_d^0(c)\\ &=^2 \frac{1}{\alpha^2} \cdot \sum_{h_2 \in \mathcal{H}'(h_1)} \sum_{h_1 \in \mathcal{H}'(a)} \hat{\mathbf{r}}_a^0(h_1) \cdot \hat{\mathbf{r}}_{h_1}^0(h_2) \cdot \hat{\mathbf{r}}_{h_2}^0(c) \quad (8)\end{aligned}$$

To begin with, as different tours may share the same hubs (recall the "sharing" property), we wonder if we can first aggregate such tours to factor out their common segments? Let's re-examine the tours t_5, t_6, and t_7. As Fig. 4 shows, t_5 and t_7 share the same hubs f, d. Thus, if we merge them at each hub node, we can aggregate the reachability of individual segments in different tours (*e.g.*, $R(f \rightsquigarrow d)$ in t_5, $R(f \rightsquigarrow g \rightsquigarrow d)$ in t_7) into an overall reachability between the ending nodes (*e.g.*, $\hat{\mathbf{r}}_f^0(d)$). We can transform t_6 similarly, since it is segmented by another set of hubs—all by itself. This assembling is illustrated in step-1 of Eq. 8.

More generally, as we observed, aggregating those tours with the same hubs is effectively aggregating direct segments between hubs. This will prove to be useful, since we have now abstracted the aggregation of tours in terms of the set of hubs they pass through (in this case, from a to c through $\{f, d\}$ or $\{b, d\}$).

Further, to aggregate these "hub-abstracted" tours, we wonder if they can be enumerated in a systematic order. From the result of step 1, we observe that even the tours segmented by different hubs can be generated by a same two-level expansion pattern: $a \rightsquigarrow h_1 \rightsquigarrow h_2 \rightsquigarrow c$ where h_1 is the first-level hub (*e.g.*, f and d) to be reached from a, and h_2 is any hub (*e.g.*, d) to be reached at the second level (from h_1). To emphasize this level-by-level property, we refer to the first-level hubs h_1 as the *neighboring hubs* of the starting node a, denoted $\mathcal{H}'(a)$, and similarly, the second-level hubs are referred as the neighboring hubs of h_1, *i.e.*, $\mathcal{H}'(h_1)$. Therefore, to aggregate every tour in the form $a \rightsquigarrow h_1 \rightsquigarrow h_2 \rightsquigarrow c$, we further merge the neighboring hubs $\mathcal{H}'(h_i)$ at each level i as step 2 of Eq. 8 shows.

Overall, with our tour assembly model, we can transform the aggregation of tours into the aggregation of intermediate segments between hubs at each level. Specifically, by merging the segments from a to the first-level hubs (*e.g.*, f and d), we gather the tours in the "neighborhood" of a (*i.e.*, tours in T^0) to form a *prime subgraph* which is rooted at a and bordered by h_1's, denoted $\mathcal{G}'(a)$. We call the aggregated reachability of the constituent tours in $\mathcal{G}'(a)$ the *prime PPV* of a, denoted $\hat{\mathbf{r}}_a^0$. Similarly, merging tour segments from h_1 to the second-level hubs, we form the prime subgraphs of h_1 (*e.g.*, $\mathcal{G}'(f)$ and $\mathcal{G}'(d)$).

Definition 2 (Prime Subgraph and Prime PPV) Given a graph G and a set of hub nodes H, for any node v,

- The *prime subgraph* $\mathcal{G}'(v)$ of v consists of all the nodes and edges in T^0, the tours starting at v with $\mathcal{L}_h(t) = 0$; and the neighboring hubs of v, $\mathcal{H}'(v)$, can also be referred as the *border hub nodes* of $\mathcal{G}'(v)$.
- The reachability from v to each node through all tours in $\mathcal{G}'(v)$ forms the *prime PPV* of v, *i.e.*, $\hat{\mathbf{r}}_v^0$.

In general, we can compute $\hat{\mathbf{r}}_q^i(p)$, the reachability between q and p over tour set T^i by assembling $\hat{\mathbf{r}}_q^0$ and the prime PPVs of up to i-th level hubs, as formalized in Theorem 3. The essence of this theorem boils down to the Chapman-Kolmogorov equation [20], which relates a joint probability distribution with the combination of a set of transition probabilities.

Theorem 3 *Let q be the query node, H be a set of hub nodes in graph G. For any node p, the personalized PR score estimated over tours of hub length i can be constructed as:*

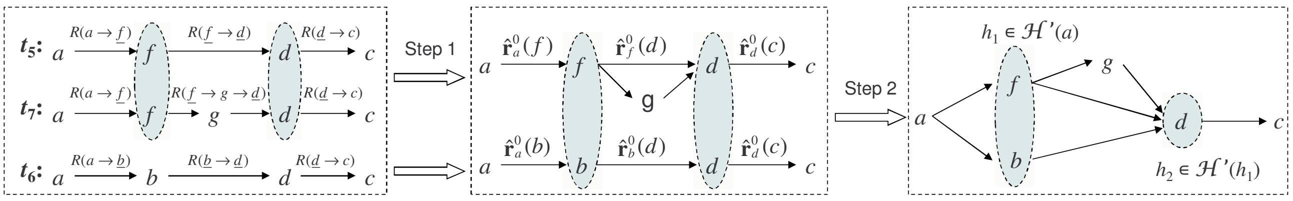
$$\hat{\mathbf{r}}_q^i(p) = \frac{1}{\alpha^i} \cdot \sum_{h_i \in \mathcal{H}'(h_{i-1})} \cdots \sum_{h_1 \in \mathcal{H}'(q)} \hat{\mathbf{r}}_q^0(h_1) \cdots \hat{\mathbf{r}}_{h_{i-1}}^0(h_i) \cdot \hat{\mathbf{r}}_{h_i}^0(p) \quad (9)$$

4.2.2 Computing with Prefixes and Building Blocks

By the tour assembly model discussed in Sect. 4.2.1, we are able to assemble a PPV increment by structured building blocks: the prime PPVs of hub nodes. We will now exploit the common substructures between PPV increments (calculated in successive iterations) for efficient computation.

To motivate, we rewrite Eq. 8 to connect $\hat{\mathbf{r}}_a^2(c)$ with the preceding PPV increments. To better illustrate this connection, we represent the border hubs in terms of their hub length, *e.g.*, $h_1 \in \mathcal{H}'(q)$ is explicitly represented as $\mathcal{L}_h(a \rightsquigarrow h_1) = 1$. Then, we rearrange and isolate the terms related to h_1 as an inner summation, which can be substituted with $\hat{\mathbf{r}}_a^1(h_2)$ (by Theorem 3), as follows:

$$\begin{aligned}&\hat{\mathbf{r}}_a^2(c)\\ =&\frac{1}{\alpha} \cdot \sum_{h_2 \in H, \mathcal{L}_h(a \rightsquigarrow h_2)=1} \left(\frac{1}{\alpha} \cdot \sum_{h_1 \in H, \mathcal{L}_h(a \rightsquigarrow h_1)=0} \hat{\mathbf{r}}_a^0(h_1) \cdot \hat{\mathbf{r}}_{h_1}^0(h_2) \right) \cdot \hat{\mathbf{r}}_{h_2}^0(c)\\ =&\frac{1}{\alpha} \cdot \sum_{h_2 \in H, \mathcal{L}_h(a \rightsquigarrow h_2)=1} \hat{\mathbf{r}}_a^1(h_2) \cdot \hat{\mathbf{r}}_{h_2}^0(c) \qquad (10)\end{aligned}$$



t_5: $a \xrightarrow{R(a\to f)} f \xrightarrow{R(f\to d)} d \xrightarrow{R(d\to c)} c$
t_7: $a \xrightarrow{R(a\to f)} f \xrightarrow{R(f\to g\to d)} g \to d \xrightarrow{R(d\to c)} c$
t_6: $a \xrightarrow{R(a\to b)} b \xrightarrow{R(b\to d)} d \xrightarrow{R(d\to c)} c$
Step 1
$\hat{r}^0_a(f)$
$\hat{r}^0_f(d)$
$\hat{r}^0_d(c)$
g
$\hat{r}^0_a(b)$
$\hat{r}^0_b(d)$
$\hat{r}^0_d(c)$
Step 2
$h_1 \in \mathcal{H}'(a)$
$h_2 \in \mathcal{H}'(h_1)$

Fig. 4: Tour assembly example, corresponding to the two steps in Eq. 8.

Basically, this transformation enables the efficient computation of PPV increments. As an example, to compute PPV increment-2 $\hat{\mathbf{r}}^2_a(c)$, we do not need to assemble the prime PPVs $\hat{\mathbf{r}}^0_a(h_1)$, $\hat{\mathbf{r}}^0_{h_1}(h_2)$ and $\hat{\mathbf{r}}^0_{h_2}(c)$ from scratch. Rather, it can be simply built from PPV increment-1 $\hat{\mathbf{r}}^1_a(h_2)$ and a specific PPV $\hat{\mathbf{r}}^0_{h_2}(c)$ involved in iteration-2. Likewise, $\hat{\mathbf{r}}^1_a(h_2)$ itself can be assembled by PPV increment-0 $\hat{\mathbf{r}}^0_a(h_1)$ and a specific PPV $\hat{\mathbf{r}}^0_{h_1}(h_2)$, as illustrated in the inner summation of the first line in Eq. 10.

Formally, in Theorem 4 we present a general result by recursively expanding Eq. 9 for each iteration i. For any $\hat{\mathbf{r}}^i_q$ $(i > 0)$, we can reuse $\hat{\mathbf{r}}^{i-1}_q$ computed over T^{i-1}, directly assembling it with the prime PPVs of the i-th level hub nodes in T^i, *i.e.*, $\hat{\mathbf{r}}^0_{h_i}(p)$. The proof mirrors the above derivation of Eq. 10.

Theorem 4 *Let q be the query node, H be a set of hub nodes in graph G. For any node p, the personalized PR score estimated over T^i can be computed as:*

$$\hat{\mathbf{r}}^i_q(p) = \frac{1}{\alpha} \cdot \sum_{h_i \in H, \mathcal{L}_h(q \rightsquigarrow h_i) = i-1} \hat{\mathbf{r}}^{i-1}_q(h_i) \cdot \hat{\mathbf{r}}^0_{h_i}(p) \qquad (11)$$

In summary, Theorem 4 exploits the shared substructures both within and across iterations, entailing two crucial aspects for speeding up computation:

- *Reusing Prefix Tours.* The PPV increment-i or $\hat{\mathbf{r}}^i_q(p)$, computed over tours in T^i, can be simply extended from its prefix $\hat{\mathbf{r}}^{i-1}_q(h_i)$, which is already computed in $\hat{\mathbf{r}}^{i-1}_q$, the PPV increment of the last iteration. Thus, the incremental PPV enhancement can be efficiently realized by recursively reusing the PPV increments in an earlier iteration to construct an enhanced estimation.
- *Precomputing Building Blocks.* The extension beyond the prefix is $\hat{\mathbf{r}}^0_{h_i}(p)$, the prime PPV of hub h_i, which is independent of the query q. Thus, to enable fast online computation, we can precompute these query-independent prime PPVs (*i.e.*, prime PPV of each hub) and use them as *building blocks* to construct any PPV increment on the fly.

5 Hub Nodes Selection

As discussed in Sect. 4, the realization of FastPPV relies on a set of hub nodes to partition tours and prioritize computation. Different hub nodes result in different building blocks, and thus ultimately impact the performance of FastPPV. While we previously assumed that the set of hubs H are given, we now tackle the issue of selecting the hubs.

To select a set of hubs, we need to decide which nodes shall be identified as hubs, and how many hubs shall be selected for a given graph. Since the number of hubs $|H|$ determines the trade-off between the offline precomputation and online processing, we leave its discussion to Sect. 6 when presenting the overall framework of FastPPV. In this section, we focus on the strategy of selecting hubs, given the number of hubs $|H|$ as input. In particular, we will first introduce a conceptual model for hub selection, followed by specific realizations of this model.

5.1 Conceptual Hub Selection Model

As FastPPV depends on a set of hubs to partition the tours by their hub length, and to accelerate computation by sharing and reusing hub segments, our goal here is to choose the most useful set of hubs. Formally, given a graph $G = (V, E)$, a query distribution Q which indicates the probability that each node in V would be queried, and the number of hubs $|H|$ as input, our goal is to select $|H|$ nodes from V such that H is most useful for FastPPV to answer the queries distributed according to Q. Mathematically, we select hubs as follows,

$$H^* = \arg\max_{H \subseteq V} U(H|Q), \text{ subject to a given } |H|, \qquad (12)$$

where $U(H|Q)$ denotes the *usefulness* of the hubs in H *w.r.t.* Q.

Greedy hub selection. However, to find the optimal solution of Eq. 12, the order of the search space is $\binom{|V|}{|H|}$, which is prohibitively high. Thus, we propose to approximate the optimization with a typical greedy approach—we iteratively construct the hub set, where

```
Algorithm 1: GreedyHubSelection
  Input: a graph $G = (V, E)$; query distribution $Q$,
         number of hubs $|H|$
  Output: a set of hubs $H$
1 $H_0 \leftarrow \emptyset$;
2 for $k \leftarrow 0$ to $|H| - 1$ do
3     $h_{k+1} \leftarrow \arg\max_{h \in V \setminus H_k} U(h|H_k, Q)$;
4     $H_{k+1} \leftarrow H_k \cup \{h_{k+1}\}$
5 end
6 return $H_{|H|}$.
```

in each step we choose a hub node that brings maximal immediate benefit. Specifically, given some already selected hubs H_k in $k \in \{0, \ldots, |H| - 1\}$ steps, in step $k + 1$ we pick the next hub node h_{k+1} such that

$$h_{k+1} = \arg\max_{h \in V \setminus H_k} U(h|H_k, Q), \tag{13}$$

where $U(h|H_k, Q)$ denotes the marginal (or additional) usefulness of h *w.r.t.* H_k:

$$U(h|H_k, Q) \triangleq U(H_k \cup \{h\}|Q) - U(H_k|Q). \tag{14}$$

The greedy hub selection procedure is sketched in Algorithm 1. It has been established that [18] the greedy algorithm gives a $1 - \frac{1}{e}$ approximation if the objective function $U(H|Q)$ is a monotone submodular function, *i.e.*, for any $h \notin H_j$ and $H_i \subseteq H_j$,

$$U(h|H_i, Q) \geq U(h|H_j, Q) \geq 0. \tag{15}$$

As we shall see later, our choice of $U(h|H_k, Q)$ would satisfy the above requirement.

Marginal usefulness. Next, we discuss the design principles of $U(h|H_k, Q)$, the marginal usefulness of h *w.r.t.* some existing hubs H_k. Let us first consider the two crucial roles played by hub nodes as discussed in Sect. 3: 1) decaying the importance of tours for discriminating tour partitions; and 2) segmenting tours for sharing computation during query processing. To facilitate these two roles, it is natural to integrate the following two desirable properties into $U(h|H_k, Q)$.

- *Sharing.* When processing different queries, different hubs would be utilized to segment tours for sharing. The more likely a hub can be utilized by queries, the better sharing it enables. Thus, given a query distribution Q, we capture the sharing property of a candidate h as the probability that it would be utilized to process queries in Q, which we denote $P(h|Q)$. Note that the sharing property depends on Q only, which is not affected by existing hubs H_k.
- *Discriminating.* When a candidate h is utilized to process some query, it has the ability to decay the importance of tours passing through itself (h), so that we can schedule the approximation according to the hub-length metric. However, since these tours may have already been decayed by existing hubs H_k, how discriminating h is will depend on H_k. Specifically, as the tours passing through h are decayed by more hubs in H_k, the marginal (or additional) discriminating power that h can offer diminishes. Thus, we capture the discriminating property of a candidate h in the context of already chosen hubs H_k, which we denote $D(h|H_k)$. Note that, given h has been utilized during query processing (which is determined by the sharing property), its discriminating property depends on H_k only, independent of the query distribution Q.

To integrate these two properties, given existing hubs H_k and a query distribution Q, we define the marginal usefulness of a candidate h as the expected marginal discriminating power utilized by Q:

$$U(h|H_k, Q) \triangleq P(h|Q) \cdot D(h|H_k) \tag{16}$$

Naturally, we should design $D(h|H_k)$ such that it is non-negative. Furthermore, due to the diminishing property of $D(h|H_k)$, it is easy to verify that $U(h|Q)$ is monotone submodular. Thus, our greedy algorithms is guaranteed to achieve a $1 - \frac{1}{e}$ approximation.

To realize the above conceptual model of marginal usefulness, we will explore several strategies to estimate $P(h|Q)$ and $D(h|H_k)$. First, Sect. 5.2 describes a naïve strategy that is oblivious of Q and H_k, *i.e.*, $P(h|Q) \approx P(h)$ and $D(h|H_k) \approx D(h)$. Second, Sect. 5.3 considers the effect of Q on the sharing property, thus estimating $P(h|Q)$ in a better way. Third, Sect. 5.4 accounts for the effect of H_k on the discriminating property, thus estimating $D(h|H_k, Q)$ in a better way.

5.2 Naïve Hub Selection

To realize the conceptual model of marginal usefulness (Eq. 16), we need to estimate $P(h|Q)$ and $D(h|H_k)$. We develop a naïve method by making a simplifying assumption—we do not know about Q and H_k.

First, suppose the query distribution Q is unknown. That is, query nodes are arbitrarily chosen from the graph $G = (V, E)$. Thus, the probability that a candidate hub h will be utilized by a query measures the "popularity" of h on the entire graph. It is common to quantify such popularity by the PageRank score, as applied in a number of previous works [19,16,10]:

$$P(h|Q) \approx P(h) \triangleq \text{PageRank}(h). \tag{17}$$

Note that to quantify the popularity of h, simpler alternatives exist, such as the in-degree of h. However, PageRank is more effective in capturing the "global" popularity of a node, while the in-degree of a node only reflects its "local" popularity attributed by its in-neighbors. In addition, the cost of PageRank computation is not a major concern, since we only need to run it once offline, and its computation time is actually dominated by the other precomputation steps in the offline stage (see Sect. 6).

Second, suppose the existing hubs H_k are unknown. In other words, we only assess the discriminating power based on the candidate h itself, ignoring the effect of existing hubs. Recall that based on the definition of P-inverse distance (Eq. 2), the reachability (importance) of a tour decreases to a factor of $\frac{1}{|\text{Out}(h)|}$ when passing through a node h. That is, given our hub-length based partitioning, a hub node with a higher out-degree can better discriminate the tour partitions. Thus, we use the out-degree of h to quantify its discriminating power:

$$D(h|H_k) \approx D(h) \triangleq |\text{Out}(h)|. \tag{18}$$

Incorporating both factors, we obtain the following naïve strategy to approximate the marginal usefulness of a candidate hub h:

$$U(h|H_k, Q) \approx \text{PageRank}(h) \cdot |\text{Out}(h)|. \tag{19}$$

5.3 Query Distribution-Aware Hub Selection

We now consider the effect of the query distribution on the sharing property, *i.e.*, $P(h|Q)$.

Recall that in the naïve strategy, we assumed Q is unknown and queries can be arbitrarily chosen from the graph. However, in real applications, queries are not arbitrary—some nodes are more likely to be chosen as queries than others. For example, previous statistics [25] shows that out of the 154 million queries collected from the query log of AltaVista, 13.6% of them occur more than 3 times, and the maximal query frequency is as high as 1.5 million. Thus, the sharing property of a candidate h, captured by the conditional probability $P(h|Q)$ in the conceptual model (Eq. 16), should account for the query distribution Q, simply because different queries will utilize different hubs. In the following, we assume that Q is known, which can be provided by domain experts, or estimated based on a history query log.

To compute $P(h|Q)$ based on a query distribution Q, we first rewrite it as follows.

$$\begin{aligned} P(h|Q) &= \sum_{q\in V} P(h,q|Q) \\ &= \sum_{q\in V} P(h|q,Q)P(q|Q) \\ &= \sum_{q\in V} P(h|q)P(q|Q) \end{aligned} \tag{20}$$

Now, $P(h|Q)$, the probability that h would be utilized by the queries, should be computed over $P(q|Q)$, the probability of q in Q, as well as $P(h|q)$, the popularity of h *w.r.t.* q. While $P(q|Q)$ is directly given by the query distribution, $P(h|q)$ can be quantified by $\mathbf{r}_q(h)$ (the personalized PageRank score of h *w.r.t.* q):

$$P(h|Q) \triangleq \sum_{q\in V} \mathbf{r}_q(h)P(q|Q) \tag{21}$$

Furthermore, according to the Linearity Theorem [16,12,10], given a particular query distribution Q, Eq. 21 can be computed offline using just one application of the power-iteration method, whose cost is insignificant as compared to other precomputation steps in the offline stage (see Sect. 6).

Overall, Eq. 21 implies that more frequent queries in Q would have a higher weight in determining the sharing property of the hubs, which is quite intuitive.

5.4 Community-based Hub Selection

Now we investigate the impact of existing hubs H_k on hub selection, which affects the discriminating property of a candidate h, *i.e.*, $D(h|H_k)$.

Discriminating property modeling. Recall that in the conceptual model, we define $D(h|H_k)$ as the additional power (beyond H_k) in decaying T_h, tours passing through h. Note that, as FastPPV only handles those important tours for approximation, when exploiting the decaying power of h, we also ignore those insignificant tours. Thus, in the following discussion, T_h is conceptually a set of important tours (with reachability greater than some abandon threshold) that pass through h.

Apart from the inherent decaying power of h (quantified as $|Out(h)|$ as in Sect. 5.2), there are two other factors affecting the actual discriminating ability of h.

First, the overall importance (or reachability) of the tours in T_h, denoted by $I(T_h)$. As $D(h|H_k)$ is defined in terms of the importance decayed by h, the more important T_h is, the larger amount of importance can be decayed by h. For example, consider two hub candidates h_1 and h_2 with the same out-degree, suppose $I(T_{h_1})$ is twice of $I(T_{h_2})$, then the importance of T_{h_1} decayed by h_1, *i.e.*, $I(T_{h_1})\cdot(1-\frac{1}{|\mathrm{Out}(h_1)|})$, is also twice of the importance of T_{h_2} decayed by h_2, *i.e.*, $I(T_{h_2})\cdot(1-\frac{1}{|\mathrm{Out}(h_2)|})$. Thus, $D(h|H_k)$ is proportional to $I(T_h)$.

Second, the number of already chosen hubs in T_h, denoted by $|H_k(T_h)|$. Since those hubs selected in the previous k steps have already decayed $I(T_h)$ to some extent, if there exist a larger number of hubs in H_k to decay T_h, then the additional importance further decayed by h would be a smaller amount. Thus, $D(h|H_k)$ is inversely proportional to $|H_k(T_h)|$. In other words, this factor captures the diminishing nature of the marginal discriminating power.

Integrating the above two factors, the marginal discriminating power of h can be modeled as follows:

$$D(h|H_k) \propto \frac{I(T_h)}{|H_k(T_h)|}. \tag{22}$$

We can explain $\frac{I(T_h)}{|H_k(T_h)|}$ as the average importance that can be decayed by a hub in T_h, *i.e.*, per-hub importance. Thus, intuitively, h would be more useful if such importance in T_h is higher. Such per tour importance can be further incorporated with h's inherent discriminating power (*i.e.*, $|Out(h)|$) to form a complete measure of $D(h|H_k)$:

$$D(h|H_k) = \frac{I(T_h)}{|H_k(T_h)|}|\mathrm{Out}(h)|. \tag{23}$$

Communities for efficient approximation. Now, we exploit the efficient computation of $\frac{I(T_h)}{|H_k(T_h)|}$. Basically, as $I(T_h)$ is defined as the overall reachability over all tours in T_h, *i.e.*, $I(T_h)=\sum_{t\in T_h} R(t)$, it is not trivial to compute $\frac{I(T_h)}{|H_k(T_h)|}$ for a specific h. Moreover, recall the greedy algorithm for hub selection, at each step we need to compute a $\frac{I(T_h)}{|H_k(T_h)|}$ for every hub candidate $h \in G$ (*i.e.*, every node in $G \setminus H_k$), which makes the problem even more challenging.

To tackle this challenge, we observe that there exists a significant overlap between the tours of closely-connected nodes. These tours "overlap" by sharing similar edges. For example, in our toy graph (Fig. 1), $f \to d \to c$ highly overlaps with $g \to d \to c$ as only the starting node is different in these two tours. Now consider two hub candidates, h_1 and h_2, whose tours overlap with each other. As two tours overlap when they pass through a similar set of nodes, we can claim that T_{h_1} shares a lot of nodes with T_{h_2}, thus $|H_k(T_{h_1})|$ is similar to $|H_k(T_{h_2})|$. Furthermore, $I(T_{h_1})$ is also similar to $I(T_{h_2})$ because of the high overlap between T_{h_1} and T_{h_2}. In conclusion, $\frac{I(T_h)}{|H_k(T_h)|}$ is similar for hubs that have overlapping tours.

This observation motivates an efficient algorithm to approximate $D(h|H_k)$ for all hub candidates. The principle is to aggregate a set of hub candidates S_h that have highly overlapped tour sets, and identify a shared tour set T_{S_h} to approximate their tour sets, so that for any $h \in S_h$, its marginal discriminating ability can be approximated over the shared tour set instead of each individual tour set T_h,*i.e.*, $D(h|H_k) \approx \frac{I(T_{S_h})}{|H_k(T_{S_h})|}$, .

Specifically, a reasonable aggregation and approximation should have two properties:

- There is a large overlap in the tours of the nodes in S_h, so that $\frac{I(T_h)}{|H_k(T_h)|}$ is similar for any $h \in S_h$.
- For each specific $h \in S_h$, the shared tour set T_{S_h} is a good approximation of T_h, thus it is reasonable to approximate $\frac{I(T_h)}{|H_k(T_h)|}$ as $\frac{I(T_{S_h})}{|H_k(T_{S_h})|}$.

Such desired properties can be observed in the *communities* that naturally exist on a graph. A community is a group of nodes that densely connects with each other within the community, while sparsely connects with nodes in other communities [9]. As nodes are densely intra-linked within the communities, the tours of nodes in the same community C_i are significantly overlapped (satisfying *property* 1). On the other hand, for an arbitrary node in C_i, as it is sparsely connected with nodes outside C_i, tours within C_i are a good approximate of its complete tour set. Thus, for each node in the community, its tour set can be reasonably approximated by the tours in C_i (satisfying *property* 2).

Therefore, we can leverage the community structure of a graph, which can be observed in most of the real graphs as discussed in [9], to obtain a good aggregation (or partition) of similar nodes (*i.e.*, nodes with large overlapped tours); furthermore, each community can serve as the common tour set of nodes in it.

Formally, given a set of communities $C = \{C_1, \ldots, C_n\}$ identified on G, the marginal discriminating ability of h in community C_i can be approximated on T_{C_i}, formalized as follows:

$$D(h|H_k) \approx \frac{I(T_{C_i})}{|H_k(T_{C_i})|}|\mathrm{Out}(h)| \tag{24}$$

Model simplification. We take a further step to simplify Eq. 24 so that it can be more efficiently computed.

First, we establish that for an arbitrary community C_i, its overall importance $I(T_{C_i})$ can be approximated as $|V_{C_i}|$, the number of nodes in C_i, as formalized in Theorem 5. The proof is presented in Appendix A.

Theorem 5 *Let $|V_{C_i}|$ be the number of nodes in community C_i, then $I(T_{C_i}) \approx |V_{C_i}|$.*

Then, the above measure in Eq. 24 can further be simplified as:

$$D(h|H_k) \approx \frac{|V_{C_h}|}{|H_k(T_{C_h})|}|\text{Out}(h)| \tag{25}$$

The simplified model provides an efficient way to compute $D(h|H_k)$ based on the number of existing hubs in each community. As a result, we can expect a more even distribution of hubs on the graph, among the communities, so that it is easy for queries in any region of the graph to find useful hubs to assemble their tours, accelerating the computation of FastPPV.

Early-termination algorithm for community based hub selection. Finally, let's consider our greedy hub selection framework (Algorithm 1). Now, the marginal usefulness $U(h|H_k, Q)$ of a hub candidate h is calculated as $U(h|H_k, Q) = \frac{|V_{C_h}|}{|H_k(T_{C_h})|} \cdot |\text{Out}(h)| \cdot \text{PageRank}(h)$. Since $|H_k(T_{C_h})|$ changes during the iterations, theoretically, we need to recalculate $U(h|H_k, Q)$ for every remaining node to find the one with maximal value, which is not efficient for large-scale graphs. Thus, we propose an *early-termination* algorithm to speed up the process. The main idea is that instead of calculating $U(h|H_k, Q)$ for every remaining node, we first rank them by roughly estimating their marginal usefulness, then starting from the most promising node, we only need to iterate through a few top candidates to find the best hub. Thus, the algorithm can terminate early without iterating through all the nodes. The algorithm is sketched in Algorithm 2.

First, all the nodes are sorted by the partial scores that are constant through iterations (Line 1-5), then at each selection, we compute the changing part $D(h|H_k)$ from the node with the highest value, and keep track of the maximal $D(h|H_k)$ we have encountered (Line 10-17). Once we see a node whose $|V_{C_h}||Out(h)|$ score divided by the minimal number of selected hubs in each community is smaller than the current maximal $D(h|H_k)$ tracked, we can terminate at this node, since subsequent nodes (with smaller constant scores) can never exceeds the current maximal score (Line 18-20). The node with the maximal $D(h|H_k)$ we have tracked so far will be selected as hub in this iteration.

Community detection methods. As the first step in FastPPV-C, we need to detect communities from a graph. Community detection or graph clustering is an orthogonal problem to our hub selection strategy, and there exists abundant literature [8] on it. Here we adopt the technique in [24] which are scalable to very large graphs, and briefly explain it below.

Algorithm 2: CommunityBasedHubSelection

```
Input: a graph $G =< V, E >$; $n$ communities
       $C = \{C_1, ...C_n\}$, number of hubs $|H|$
Output: a set of hubs $H$
1  foreach $v \in V$ do
2      Add $v$ to node list $N[]$;
3      Add $|V_{C_v}| \cdot PageRank(v) \cdot Out(v)$ to value list $V[]$;
4  end
5  Rank $N[]$ and $V[]$ in the ascending order of
   $|V_{C_v}| \cdot PageRank(v) \cdot Out(v)$;
6  $H_0 \leftarrow \emptyset$;
7  for $k \leftarrow 0$ to $|H| - 1$ do
8      $maxD \leftarrow 0$;
9      $minH \leftarrow Min(|H_{C_1}|, \ldots, |H_{C_n}|)$;
10     for $i \leftarrow 0$ to $N.length$ do
11         if $N[i] \in H_k$ then
12             Continue;
13         end
14         if $D(N[i]|H_k) \geq maxD$ then
15             $maxD \leftarrow D(N[i]|H_k)$;
16             $h_{k+1} \leftarrow N[i]$;
17         end
18         if $\frac{N[i]}{minH} \leq maxD$ then
19             break;
20         end
21     end
22     $H_{k+1} \leftarrow H_k \cup \{h_{k+1}\}$;
23 end
24 return $H_{|H|}$.
```

Specifically, given the number of communities $|C|$ as input, $|C|$ "anchor" nodes are chosen randomly from the graph. Every other node in the graph is assigned to its "nearest" anchor in terms of their personalized PageRank *w.r.t.* the anchor. It has been shown that personalized PageRank exhibits a good clustering quality [1]. Hence, we can obtain good communities even though the anchors are selected randomly, since every node in a community can become the anchor.

6 Overall Framework

To materialize the computation in Eq. 11, our overall framework consists of two phases: 1) *Offline precomputation* where we precompute the building blocks; 2) *Online query processing* where we reuse the building blocks and prefix tours to incrementally compute a gradually more accurate PPV for any query.

To develop the two phases, we first treat the graph as residing in the main memory, a typical assumption adopted in recent works such as [10,11]. Next, to handle graphs that are too large for the main memory, we propose a disk-based implementation for FastPPV.

6.1 Offline Precomputation

In the offline phase, we need to compute the building blocks, *i.e.*, prime PPVs for a set of hub nodes H over G. These building blocks are then stored in an index, which will be used in online query processing.

Given a graph G and number of hubs $|H|$ as input, we first select hubs according to their expected utility (see Sect. 4). Next, for each $h \in H$, we compute its prime PPV $\hat{\mathbf{r}}_h^0$ using the standard power-iteration algorithm over its corresponding prime subgraph, which is feasible as prime subgraphs are many orders smaller than the entire graph. The prime subgraph can be identified using a depth-first search starting from the query node. During the search, we backtrack when we hit a hub node (which are the border hub nodes for this prime subgraph), or a "faraway" node whose reachability to the query node is smaller than a threshold ϵ (say 10^{-8}).

The above steps are summarized in Algorithm 2. In particular, the precomputed prime PPVs or building blocks are stored in a PPV index on disk, which can be loaded into the main memory as needed during online query processing.

Time complexity. As hubs essentially form the borders of prime subgraphs, we can informally view a graph as being divided into prime subgraphs at the hub nodes. Intuitively, more hubs result in smaller prime subgraphs. As each hub node blocks an entire search subtree during the depth-first search for the prime subgraph, the size of a prime subgraph decreases exponentially in $|H|$. Hence, it is reasonable to assume that on average a prime subgraph is smaller than $O(1/|H|)$ of the entire graph, *i.e.*, contains fewer than $O(|V|/|H|)$ nodes and $O(|E|/|H|)$ edges. Thus, computing a prime PPV over such a prime subgraph using the standard power iteration costs less than $O\left(I(|V|+|E|)/|H|\right)$, where I is the number of iterations. Therefore, the total precomputation time for all hubs can be upper bounded by $O\left(|H| \cdot I(|V|+|E|)/|H|\right) = O\left(I(|V|+|E|)\right)$.

This result implies that our offline precomputation is scalable in the number of hubs, since the upper bound is independent of $|H|$. The ability to index a large number of hubs offline is crucial to speeding up online query processing as we will discuss in Sect. 6.2. Although our time complexity is obtained under a quite simplifying assumption, the experiments in Sect. 7.3 do demonstrate that offline precomputation is scalable in the number of hubs.

Space complexity. Under the same assumption that on average a prime subgraph is smaller than $O(1/|H|)$ of the entire graph, the space cost of the PPV index can be likewise upper bounded by $O\left(|H| \cdot |V|/|H|\right) = O(|V|)$, which is also independent of $|H|$.

```
Algorithm 3: OfflinePrecomputation
   Input: a graph $G$; number of hub nodes $|H|$
   Output: PPV index $\Phi$
1  $\Phi \leftarrow \emptyset$;
2  $H \leftarrow$ Select $|H|$ hubs on $G$;
3  foreach $h \in H$ do
4      Compute prime PPV $\hat{\mathbf{r}}_h^0$ for $h$ on $G$;
5      Store $\hat{\mathbf{r}}_h^0$ in PPV index $\Phi$;
6  end
7  return $\Phi$.
```

6.2 Online Query Processing

In the online phase, given a graph G, a precomputed PPV index Φ, a query node q and a stopping condition S, we incrementally compute an approximate PPV $\hat{\mathbf{r}}_q^{(\eta)}$ for a given query q according to Eq. 11, as sketched in Algorithm 3. The algorithm consists of two major steps: computing the initial iteration $i = 0$ (line 1–5) and subsequent iterations $i \geq 1$ (line 6–16).

In the initial iteration $i = 0$, we need to compute the prime PPV $\hat{\mathbf{r}}_q^0$ for the query node q, which is required in iteration $i = 1$ (see Eq. 11). If q happens to be a hub node, we can directly load $\hat{\mathbf{r}}_q^0$ from the precomputed index; otherwise we need to compute it on-the-fly.

To compute subsequent iterations $i \geq 1$ for Eq. 11, we will reuse the prefixes—PPV increment $\hat{\mathbf{r}}_q^{i-1}$ from iteration $i-1$, as well as the building blocks—precomputed prime PPVs of some hub nodes $h \in H_{\text{exp}}$ (line 11). In particular, these hub nodes $h \in H_{\text{exp}}$ are the border hub nodes of the hubs used in iteration $i-1$ (line 12).

It is worth noting that we also need to specify a stopping condition S as an input. The choice of S is flexible depending on the desired trade-off between accuracy and efficiency—we can stop the incremental iterations when an accuracy requirement (in terms of L_1 error) is achieved, or a time limit for query processing is up, or the maximum number of iterations η is reached.

For a practical implementation, we impose a threshold δ (say 0.005) on the border hub nodes, such that we include them only if $\hat{\mathbf{r}}_q^{i-1}(h) > \delta$ (line 9). This threshold prevents least contributing hubs, improving efficiency with minimal impact on accuracy.

Time complexity. Suppose that there is an average of $O(|\bar{H}|)$ border hub nodes in each prime PPV. Thus, in η iterations we need to handle $O(|\bar{H}|^{\eta})$ hub nodes. Typically $|\bar{H}| \ll |H|$ and $\eta \leq 5$. For instance, in our experiments, even when $\eta = 1$, an average precision of above 0.9 can already be achieved. Hence, this complexity is practically feasible. Additionally, if the query

```
Algorithm 4: OnlineQueryProcessing
  Input: a graph $G$; PPV index $\Phi$ over $H$; query node $q$;
         stopping condition $S$
  Output: estimated PPV $\hat{\mathbf{r}}_q^{(\eta)}$
1  if $q \notin H$ then
2  |  Compute prime PPV $\hat{\mathbf{r}}_q^0$ for $q$ on $G$;
3  else
4  |  Load $\hat{\mathbf{r}}_q^0$ from PPV index $\Phi$;
5  end
6  $\hat{\mathbf{r}}_q^{(\eta)} \leftarrow \hat{\mathbf{r}}_q^0$; $i \leftarrow 0$; $H_{\exp} \leftarrow \mathcal{H}'(q)$;
7  while the stopping condition $S$ not met do
8  |  $i \leftarrow i+1$; $\hat{\mathbf{r}}_q^i \leftarrow \mathbf{0}$; $H_{\text{nextToExp}} \leftarrow \emptyset$;
9  |  foreach $h \in H_{\exp}$ such that $\hat{r}_q^{i-1}(h) > \delta$ do
10 |  |  Load $\hat{\mathbf{r}}_h^0$ from PPV index $\Phi$;
11 |  |  $\hat{\mathbf{r}}_q^i \leftarrow \hat{\mathbf{r}}_q^i + \frac{1}{\alpha}\hat{\mathbf{r}}_q^{i-1}(h)\hat{\mathbf{r}}_h^0$;
12 |  |  $H_{\text{nextToExp}} \leftarrow H_{\text{nextToExp}} \cup \mathcal{H}'(h)$;
13 |  end
14 |  $\hat{\mathbf{r}}_q^{(\eta)} \leftarrow \hat{\mathbf{r}}_q^{(\eta)} + \hat{\mathbf{r}}_q^i$;
15 |  $H_{\exp} \leftarrow H_{\text{nextToExp}}$;
16 end
17 return $\hat{\mathbf{r}}_q^{(\eta)}$.
```

is not a hub node, an extra $O((|V| + |E|)/|H|)$ time is needed for computing its prime PPV, which decreases with a larger $|H|$.

6.3 Disk-based Implementation

Even with the availability of many graph compression techniques [22, 7], some real-world graphs are still too large to reside entirely in the main memory. To handle these graphs, we describe a disk-based approach for online query processing. Disk-based offline precomputation can be implemented using similar ideas.

First, we observe that in online processing, we need the entire graph in the main memory such that we can identify the prime subgraph for the query node. However, after we have obtained the prime subgraph, we no longer require the entire graph—only the prime subgraph is needed, which is generally many orders of magnitude smaller than the entire graph.

Hence, given a query node, the key to the disk-based online query processing is to identify its prime subgraph from a disk-resident graph. The basic idea is to segment the graph into a number of clusters, such that we can at least fit each single cluster into the main memory. Subsequently, we can assemble the prime subgraph by searching in each cluster separately.

Specifically, to identify the prime subgraph for the query node, we first load the cluster that contains the query node into the memory, and start the depth-first search in this cluster until we reach a node that is outside this cluster. We call this event a *cluster fault*, at which point we will swap the required cluster into the main memory to continue the depth-first search. As frequent cluster faults significantly slow down query processing, we may prematurely terminate the search once reaching a threshold on the number of cluster faults. This can considerably speed up query processing with a minimal loss in accuracy. In our experiments, we set the threshold to the total number of clusters, which is generally robust.

Finally, to segment a graph into clusters, we adopt the technique in [24], which has also been briefly explain in Sect. 5.4 for community detection. Note that clusters and communities are similar notions which we use interchangeably.

7 Experiments

In this section, we empirically evaluated FastPPV on two real-world graphs, and obtained promising results in terms of accuracy, efficiency, and scalability.

In the following, we first describe our experimental settings. Second, we show that FastPPV substantially outperforms previous state-of-the-art baselines in both the offline and online phases. Third, we also present the trade-off between accuracy and efficiency to gain more insights into FastPPV. Fourth, we showcase the performance of FastPPV on graphs of varying sizes as well as disk-resident graphs, and concluded that it is scalable. Last, we also compared the performance of different hub selection strategies to validate key role played by hub nodes.

7.1 Experimental Settings

Datasets. We used two public real-world graphs below. In particular, the first graph is undirected, and the second one is directed.

- DBLP[2]: A *bibliographic network* of authors, papers and venues, with *undirected* edges representing the author-paper and paper-venue relationships. The graph contains 2 million nodes and 8.8 million edges.
- LiveJournal[3]: A *social network* where users can declare their friends. The friendship relationship is not necessarily reciprocal, and hence a *directed* edge from node i to j means that user i declares j as a friend. We sampled a graph with 1.2 million nodes and 4.8 million edges. Larger samples will also be used to study the scalability of FastPPV in Sect. 7.4.

[2] http://www.informatik.uni-trier.de/~ley/db/
[3] http://snap.stanford.edu/data/

Test queries. We randomly sample 1000 nodes from each graph, where every chosen node is a test query.[4] We only focus on these single-node queries, since a multi-node query can be easily decomposed as multiple single-node queries using the Linearity Theorem [16, 12, 10]. For each experiment, we report the mean performance over all test queries.

Hub selection strategy. We assume the naïve strategy in all experiments unless otherwise stated. Nonetheless, in Sect. 7.5 we investigate the other two hub selection strategies proposed in Sect. 5. We label the three strategies as follows.

- FastPPV: the naïve strategy.
- FastPPV-Q: the query distribution-aware strategy.
- FastPPV-C: the community-based strategy.

Parameters. First, we must determine the *number of hubs* $|H|$, which influences online query processing speed. We will specifically study the performance of FastPPV under different $|H|$. However, as default, for other experiments we empirically set $|H| = 20$K for DBLP and $|H| = 120$K for LiveJournal unless otherwise stated, such that online query times are comparable on both datasets.

Next, we also need to specify the *number of iterations* η, which dynamically controls the trade-off between accuracy and query time in the online phase. By default we use $\eta = 2$ unless otherwise stated.

Finally, for the precomputed PPVs, we clip them at 10^{-4}, *i.e.*, discarding nodes with scores less than 10^{-4} for offline storage. It can drastically reduce offline space cost with minimal impact on accuracy [10, 11]. Moreover, we set $\alpha = 0.15$ (Eq. 2), which is a typical teleporting probability.

Accuracy metrics. Given a query, all the methods compute an approximate PPV. Thus, we need to evaluate their accuracy *w.r.t.* the exact PPV, in terms of ranking and score. Since users are usually more interested in higher ranked nodes, we focus on the top 10 nodes. Our accuracy objective is two-fold—we evaluate not only node rankings, but also node scores. In particular, we adopted four metrics from previous works [10–12], namely *Kendall's* τ and *precision* to measure the rankings, as well as *RAG* and *L1 error* to measure the scores. We refer readers to [10] for details.

For a consistent presentation, we report the complement of L1 error (1 − L1 error) instead, which we call *L1 similarity*. Now, all the metrics indicates a better accuracy with a larger value.

[4] Except the experiment on query distribution-aware hub selection, which will be discussed in Sect. 7.5.2.

Environment. We implement all methods in Java, and evaluate them on a Linux system with 2.67GHz CPU and 10GB RAM. The entire graph resides in the main memory except for the disk-based implementation in Sect. 7.4. In that case, the graph is disk-resident as we assume a reduced memory budget.

7.2 Comparison to baselines

In the following experiments, we compare the performance of FastPPV and previous state of the arts.

Baselines. We adopt three recent baselines, namely, HubRankP, MC1 and MC2. They represent two major lines of related work. Specifically, HubRankP is a deterministic algorithm based on decomposing and reusing computation, whereas MC1 and MC2 are Monte Carlo algorithms based on random walk simulations. In particular, two Monte Carlo methods are presented given that they have quite different offline strategies, as we will elaborate in the following.

First, we implemented HubRankP [11] using their proposed benefit-based hub selection model to optimize online query time. To realize their benefit model, we assume a uniformly distributed query log, which is fair as our test queries are also sampled uniformly. Note that HubRankP builds upon the Bookmark-coloring algorithm [6] with a better hub selection policy. In addition, as Chakrabarti *et al.* [11] show, HubRankP is also superior to HubRankD [10] (which is itself more efficient than Jeh and Widom's hub decomposition method [16]). Hence, among these various approaches [11, 10, 6, 16], we only present the state-of-the-art HubRankP as the baseline.

We implemented a second baseline using fingerprints [12], which we call MC1. Specifically, a *fingerprint* is a sample destination node for a random walk starting from the query node. Naturally, the more samples we obtain, the more accurate the approximation is. Although it was originally meant to sample fingerprints for each query node offline, we can process queries online by on-the-fly sampling. To reduce the online workload, we first sample fingerprints for a set of hub nodes offline, which can be reused online. To increase the chance of hitting a hub node, we select nodes with largest PageRank scores as hubs, which is a common strategy used in previous works [16, 6].

We also adopted another Monte Carlo method based on walk segments [4], which we call MC2. A *walk segment* is a sample random walk path of relatively short length, which can be concatenated with each other online to obtain a longer path. Starting from the query node, a random walk path of enough length can be used

	Dataset	FastPPV ($\|H\|$, η)	HubRankP ($\|H\|$, ϵ_{push})	MC1 ($\|H\|$, N)	MC2 (R, L)
I	DBLP	20K, 2	20K, 0.11	20K, 120K	10, 200K
II	DBLP	30K, 1	30K, 0.13	30K, 40K	5, 70K
III	LiveJournal	150K, 3	150K, 0.20	150K, 200K	20, 250K
IV	LiveJournal	200K, 1	200K, 0.29	200K, 10K	10, 20K

Fig. 5: Four accuracy-moderated configurations for FastPPV and the baselines (I, II, III, IV).

	Kendall				Precision				RAG				L1 similarity			
	FastPPV	HubRankP	MC1	MC2	FastPPV	HubRankP	MC1	MC2	FastPPV	HubRankP	MC1	MC2	FastPPV	HubRankP	MC1	MC2
I	.926	.921	.921	.916	.954	.952	.957	.951	.999	.999	.999	.999	.996	.985	.996	.996
II	.889	.894	.890	.878	.930	.935	.939	.929	.999	.999	.999	.999	.994	.978	.994	.994
III	.928	.927	.886	.882	.964	.963	.959	.958	.999	.998	.999	.999	.997	.977	.997	.997
IV	.823	.824	.774	.783	.918	.907	.919	.922	.997	.989	.999	.999	.990	.958	.988	.990

Fig. 6: FastPPV achieves an accuracy level similar to the baselines under each accuracy-moderated configuration.

to estimate the PPV. However, in the offline stage, unlike other methods which perform fairly extensive computation for a small set of hub nodes, MC2 performs light computation for every node. Specifically, for each node, MC2 only simulates and stores a few walk segments, which will be concatenated online to speed up query processing.

Lastly, let us look at the parameters for the baselines. We also need to set the number of hubs $|H|$ for HubRankP and MC1, whereas for MC2 we set R to control the number of walk segments per node. In addition, each of them also has a parameter to control the accuracy. In particular, HubRankP relies on a residual threshold ϵ_{push}, MC1 depends on the number of fingerprint samples per query N, and MC2 requires an online walk length of at least L. We will discuss the configuration of these parameters next.

Configurations. As all the methods only compute approximate PPVs, there is a trade-off between accuracy and query time. To demonstrate the edge of FastPPV over the baselines, we configure the parameters to moderate their accuracy, such that FastPPV *achieves an accuracy level similar to or better than the baselines.* Moderating their accuracy in this way enables us to fairly compare FastPPV with the baselines in terms of their online query time and offline aspects.

Four such accuracy-moderated configurations have been identified in Fig. 5. The "moderated" accuracy levels are presented in Fig. 6.

Results. We first examine the online query processing time in Fig. 7(a). Across the four accuracy-moderated configurations, FastPPV consistently achives the best performance. Specifically, it is 2.0–7.2 times faster than HubRankP, 2.4–5.2 times faster than MC1, and also 1.9–4.9 times faster than MC2.

Second, we examine the total space cost in the offline stage, as shown in Fig. 7(b). In particular, FastPPV requires substantially less space than every baseline in some or all of the configurations (HubRankP and MC1 in III–VI, and MC2 in I–VI), and comparable space in other configurations.

Third, we compare the total precomputation time in the offline stage, as illustrated in Fig. 7(c). The results reveal that FastPPV and MC2 take similar time, both of which are much faster than HubRankP and MC1 across different configurations.

In summary, FastPPV is superior to all of the three baselines, factoring in the performance during both online and offline phases.

7.3 Trade-off between Accuracy and Efficiency

There are two factors affecting the tradeoff between accuracy and efficiency: the number of hubs $|H|$ and iterations η. To study the effect of each one, we will fix one of the parameters with its default value, and vary the other.

7.3.1 Number of Hubs

We first illustrate the effect of varying the number of hubs $|H|$ on online query processing in Fig. 8. As expected, having more hub nodes drastically reduces the query time of FastPPV (see Sect. 6.2). Interestingly, even with a greatly reduced query time, all the accuracy metrics remain robust.

We further study the effect of $|H|$ on offline precomputation, as illustrated in Fig. 9. As $|H|$ grows, we observe that the total space cost increases sublinearly, whereas the total precomputation time actually decreases. Let us analyze such trends. As $|H|$ increases linearly, each prime subgraph becomes smaller exponentially (see Sect. 6.1). Hence, the total size of the prime subgraphs decreases, resulting in a decreasing total precomputation time. Likewise, we would also expect a decreasing total space cost, contrary to what we

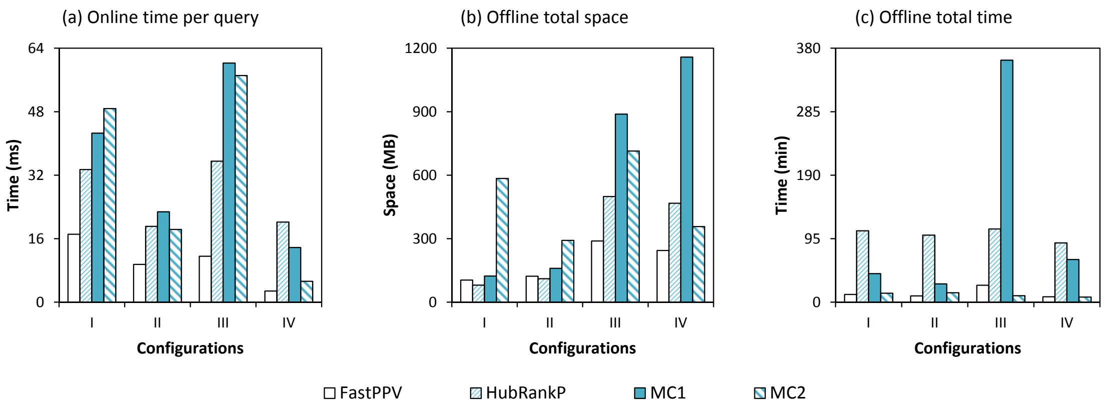


Fig. 7: Accuracy-moderated online and offline comparisons with baselines.

have observed. The reason is that we applied clipping on the prime PPVs, which is more effective on larger prime PPVs.

Hence, with a decreasing precomputation time and sublinearly increasing space cost, it is feasible to index more hubs offline, which also speeds up online query processing without compromising accuracy. Of course, if we index too many hubs (substantially more than what we are using now), the I/O overhead may eventually outweigh the benefit, since fetching the precomputed prime PPV of a hub node during online query processing requires one random access to the disk.

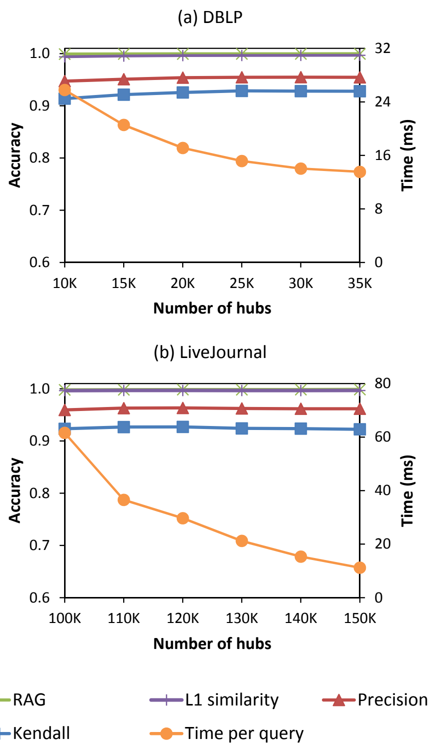


Fig. 8: Effect of $|H|$ on online processing. Left axis: accuracy (RAG, L1, Prec, Kendall). Right axis: time.

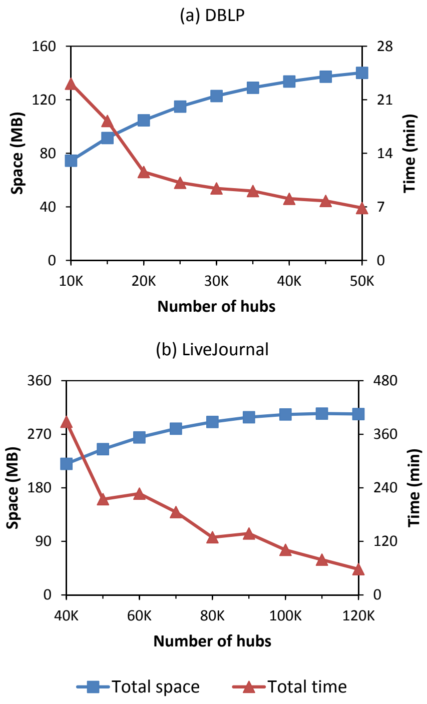


Fig. 9: Effect of $|H|$ on costs of offline precomputation. Left axis: space cost. Right axis: time cost.

7.3.2 Number of Iterations

We explore FastPPV's *incremental* query processing by varying the number of iterations η.

As depicted in Fig. 10, allowing more iterations results in better accuracy but takes longer to process. This verifies that our approximation indeed becomes more accurate in an incremental manner. In particular, the improvement in accuracy is more significant in earlier iterations, which is consistent with Theorem 2. Thus, good accuracy can be achieved with only a few iterations. For instance, in Fig. 10 all the accuracy metrics are above 0.9 with $\eta = 2$ only.

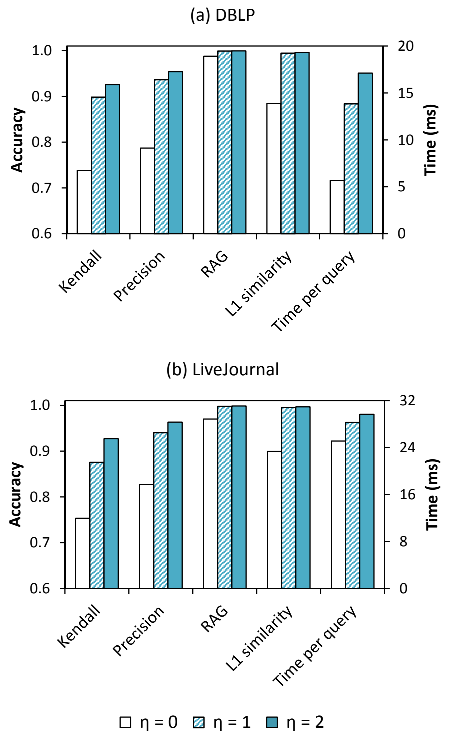


Fig. 10: Incremental online processing by varying η. Left axis: accuracy (RAG, L1, Prec, Kendall). Right axis: time.

It is worth noting that η only affects online query processing, which enables us to dynamically control the trade-off between accuracy and query time without re-executing the offline phase. In contrast, many previous works including our baselines lack such flexibility. To adjust the trade-off, their offline phases may need a re-execution.

7.4 Scalability

We investigate the scalability of FastPPV in terms of two aspects. First, how does FastPPV scale on larger graphs in the online and offline phases? Second, how does the disk-based implementation perform on a disk-resident graph given insufficient main memory?

7.4.1 Scaling to Larger Graphs

We first need to obtain graphs of varying sizes. On the one hand, each paper in DBLP has a timestamp. Thus, we take a snapshot of DBLP every four years from 1994 through 2010, as shown in Fig. 11(a). The snapshot graphs increase in size as time passes. On the other hand, we have no timestamp information in LiveJournal. Thus, we resort to sampling different numbers of edges from LiveJournal, as shown in Fig. 11(b). We order these sample graphs in increasing size, and label them S1 through S5.

(a) Snapshots from DBLP

Snapshot year	# Nodes	# Edges
1994	0.32M	1.11M
1998	0.54M	2.00M
2002	0.88M	3.48M
2006	1.51M	6.40M
2010	2.00M	8.79M

(b) Samples from LiveJournal

Sample ID	# Nodes	# Edges
S1	0.31M	0.76M
S2	0.83M	2.67M
S3	1.22M	4.81M
S4	1.53M	7.01M
S5	1.77M	9.30M

Fig. 11: Varying graph size for scalability study.

The key to scaling to larger graphs is to index more hubs offline. As shown in Fig. 12, even though the graph increases more than 5 folds on both datasets, by using a larger number of hubs $|H|$, we are able to achieve a *near constant online query time* without compromising accuracy. Hence, FastPPV can efficiently process queries online regardless of graph size, given sufficient number of hubs. In our study, we empirically determined the number of hubs required to achieve a constant query time over growing graphs. It is also interesting to predict the requirement analytically, which warrants further investigation in a future work.

Next, we examine any additional cost involved in the offline phase in order to achieve a constant online query time. In Fig. 13, we plot the total space and time needed by offline precomputation against graph size (*i.e.*, the total number of nodes and edges). The plots clearly show a linear relationship between the total space (or time) and graph size. We deem such linear growths in the offline phase acceptable for maintaining a constant online query time.

7.4.2 Disk-based Online Processing

Assuming that our graphs do not fit into the main memory, we use our disk-based online processing, where a graph is segmented into a number of clusters to mimic the limited memory budget (see Sect. 6.3). Recall that at any time, only one cluster needs to be in the main memory. Hence, the size of the largest cluster is the minimum working set, which is much smaller than the entire graph.

As reported in Fig. 14, the disk-based implementation is scalable in the number of clusters. First, when

(a) DBLP

Year	$\vert H\vert$	Kendall	Prec.	RAG	L1 sim.	Time per query
1994	1K	0.9304	0.9520	0.9995	0.9966	**15.7 ms**
1998	3K	0.9245	0.9508	0.9993	0.9968	**16.1 ms**
2002	8K	0.9309	0.9556	0.9995	0.9965	**15.1 ms**
2006	15K	0.9286	0.9527	0.9993	0.9962	**15.7 ms**
2010	25K	0.9285	0.9545	0.9994	0.9963	**15.2 ms**

(b) LiveJournal

ID	$\vert H\vert$	Kendall	Prec.	RAG	L1 sim.	Time per query
S1	14K	0.9274	0.9681	0.9984	0.9966	**28.5 ms**
S2	63K	0.9244	0.9637	0.9984	0.9970	**28.0 ms**
S3	120K	0.9269	0.9633	0.9985	0.9967	**29.7 ms**
S4	160K	0.9252	0.9645	0.9983	0.9965	**27.5 ms**
S5	200K	0.9210	0.9627	0.9986	0.9962	**29.9 ms**

Fig. 12: Scaling FastPPV in online query processing.

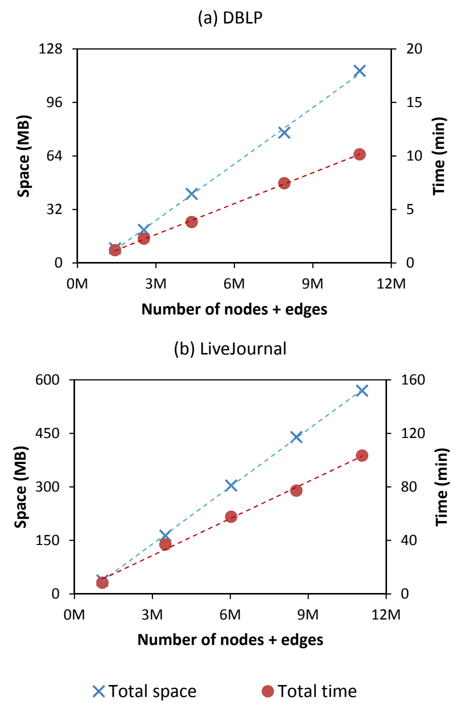


Fig. 13: The costs of offline precomputation in order to scale FastPPV online. Left axis: space cost. Right axis: time cost.

we have more clusters, query time remains stable. Although cluster faults become more frequent with more clusters, the clusters also become smaller which are faster to swap into the main memory, resulting in similar query times. Second, as the largest cluster also shrinks with more clusters, the memory requirement decreases as well.

7.5 Comparison of Hub Selection Strategies

Finally, we investigate and compare the different hub selection strategies proposed in Sect. 5.

7.5.1 Naïve Strategy

The naïve strategy uses the PageRank to capture the sharing property, and the out-degree to capture the discriminating property (Eq. 19). To show that both properties are desirable, we compare the naïve strategy with two simplifying approaches, each considering only one property. Specifically, we also select hubs by PageRank or out-degree alone. Additionally, we also evaluate a random selection strategy. However, its performance is substantially worse than the other strategies, and hence we omit it here.

As different strategies select different hub sets H, both offline precomputation for H and online processing using H are affected. To eliminate the effect of other parameters, we used the default number of hubs and iterations mentioned previously (Sect. 7.1).

We first present the impact of the three strategies (naïve with both PageRank and out-degree, PageRank only, out-degree only) on online query processing in Fig. 15. While the naïve strategy results in an accuracy level similar to or better than the others, it greatly speeds up online query processing—1.2× faster on DBLP and 2.4× faster on LiveJournal than the second best strategy. As DBLP is undirected, the three strategies are fairly correlated with smaller differences than they are on the directed LiveJournal. Hence, the speed-up is more significant on LiveJournal.

The naïve strategy also results in cheaper offline precomputation for the selected hubs. In Fig. 16, while the space cost of expected utility is similar to other strategies, precomputation is 1.3× faster on DBLP and 1.7× faster on LiveJournal than the second best strategy. Likewise, the improvement is larger on the directed LiveJournal. Note that the precomputation time here includes the time to compute the prime PPVs for every hub, but excludes the time to select these hubs—the latter is negligible compared to the former.

These results clearly demonstrate that both the sharing and discriminating properties must be accounted for in hub selection.

7.5.2 Query Distribution-Aware Strategy

To investigate the impact of the query distribution Q on the performance, we created a synthetic query log. As every node in the graph is a potential query, we assign a frequency to each node, such that the frequencies follow a power-law distribution as observed in many real-world query logs [2]. The query distribution Q is then obtained from the maximum likelihood estimation of the query frequencies. Finally, we sample the testing

	(a) DBLP			(b) LiveJournal		
# Clusters	# Faults per query	Time per query	Memory need †	# Faults per query	Time per query	Memory need †
10	7.3	1434 ms	15.2%	6.8	747 ms	19.8%
15	10.8	1376 ms	10.3%	10.2	783 ms	15.1%
25	17.8	1370 ms	7.6%	16.8	862 ms	11.7%
35	24.7	1316 ms	5.4%	23.4	833 ms	6.4%
50	35.0	1270 ms	3.5%	33.3	831 ms	5.3%

Fig. 14: Disk-based online query processing. († The size of the largest cluster as % of the entire graph.)

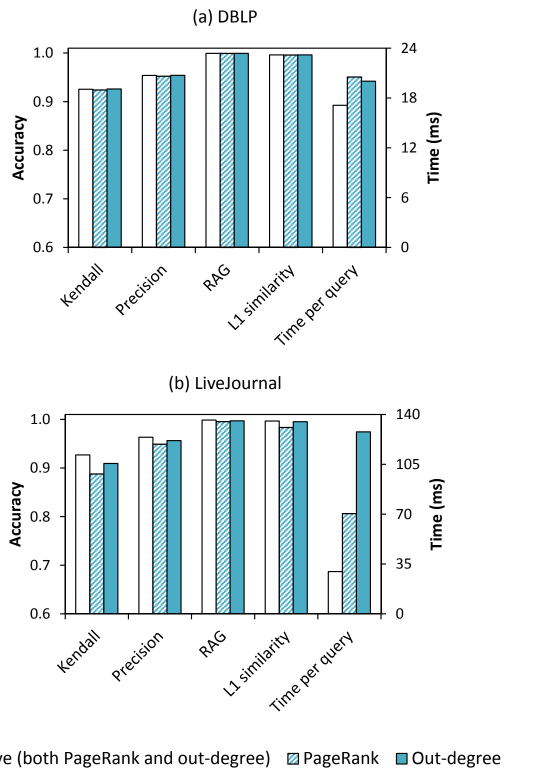


Fig. 15: Effect of hub selection with sharing (PageRank) and/or discriminating (out-degree) properties on online processing. Left axis: accuracy (Kendall, Prec, RAG, L1). Right axis: time.

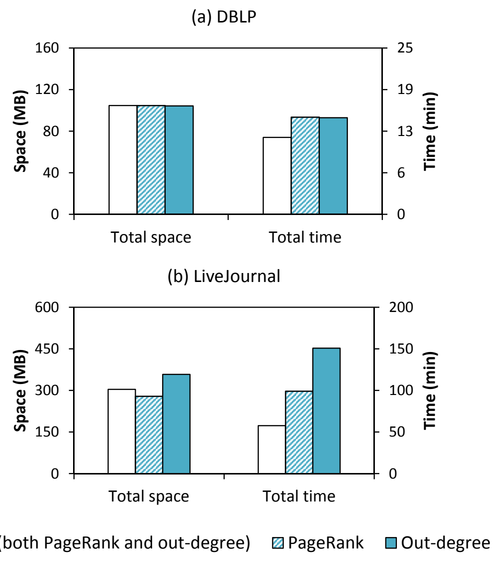


Fig. 16: Effect of hub selection with sharing (PageRank) and/or discriminating (out-degree) properties on offline precomputation. Left axis: space cost. Right axis: time cost.

Config.	Kendall	Precision	RAG	L1 Sim.	**Time**
I	−0.36%	−0.01%	−0.01%	−0.14%	**−24.83%**
II	+3.87%	+2.60%	+2.60%	−0.01%	**−34.48%**
III	+0.99%	−0.48%	−0.63%	−2.61%	**−11.86%**
IV	+2.63%	+0.11%	−0.66%	+2.79%	**−77.03%**

Fig. 17: Online processing of FastPPV-Q and FastPPV. The results are presented in terms of the relative performance of FastPPV-Q *w.r.t.* FastPPV, treating the latter as 100%.

queries from Q, assuming that more frequent queries in the past are more likely to be queried again.

We now compare FastPPV-Q and FastPPV (with the naïve strategy). As discussed in Sect. 5.3, FastPPV-Q selects hub nodes that are more likely to be utilized by the query nodes and hence enables better sharing. Thus, we expect that its online query processing time is better than FastPPV. Again, we adopt the accuracy-moderated configurations and compare their query time. As illustrated in Fig. 17, to achieve similar accuracy, FastPPV-Q is significantly faster than FastPPV, which validates our theory that the query distribution of Q would help select better hubs to accelerate PPV computation. Note that both FastPPV-Q and FastPPV perform similarly in offline computation.

7.5.3 Community-based Strategy

In the following experiments, we investigate the community-based hub selection strategy FastPPV-C. As the first step, we need to detect the communities in a graph. Recall that we adopted a simple clustering algorithm based on personalized PageRank [24], which is also discussed in Sect. 5.4.

We still use the same four configurations in Fig. 5 to compare FastPPV and FastPPV-C. As the latter also requires the number of communities $|C|$ as input, we append this parameter to the four configurations, which is presented in Fig. 18. Note that FastPPV (with the

	Dataset	both strategies: $\lvert H\rvert$	both strategies: η	FastPPV: $\lvert C\rvert$	FastPPV-C: $\lvert C\rvert$
I	DBLP	$20K$	2	1	20
II	DBLP	$30K$	1	1	20
III	LiveJournal	150K	3	1	3
IV	LiveJournal	200K	1	1	3

Fig. 18: Four configurations for FastPPV and FastPPV-C (I, II, III, VI).

naïve strategy) is equivalent to the scenario with only one community, *i.e.*, $|C| = 1$.

We compare the online processing of FastPPV-C and FastPPV in Fig. 19. The results illustrate that, to achieve similar accuracy, FastPPV-C is much faster than FastPPV. This validates our conjecture that the marginal usefulness of a candidate hub is affected by the existing hubs; thus when the hubs are more evenly allocated in communities, the overall hub set is more useful.

Config.	Kendall	Precision	RAG	L1 Sim.	**Time**
I	−0.07%	−0.02%	0.00%	+2.63%	**−6.21%**
II	+0.03%	+0.01%	0.00%	+0.15%	**−4.97%**
III	−0.04%	−0.02%	+0.56%	+0.56%	**−38.57%**
IV	+0.14%	−0.01%	−0.02%	+1.16%	**−40.59%**

Fig. 19: Online processing of FastPPV-C and FastPPV. The results are presented in terms of the relative performance of FastPPV-C *w.r.t.* FastPPV, treating the latter as 100%.

8 Conclusion and Future work

In this paper, we presented a scheduled approximation strategy to approximate PPVs. Specifically, we developed a hub length-based scheduling scheme for partitioning and prioritizing tours, as well as a structured aggregation model for assembling PPVs. As a result, our online processing is incremental and accuracy-aware, enabling a dynamic trade-off between efficiency and accuracy at query time. We also explore the issue of hub selection; we developed a conceptual model that integrates the sharing and discriminating properties to define the marginal usefulness of a hub, and propose several hub selection strategies aims at a hub set with maximal overall usefulness. Empirically, FastPPV is not only superior to two state-of-the-art baselines, but also scalable.

As future work, we identify three major directions to explore. First, *automatic configuration*: for example, automatically determine the optimal number of hubs by correlating with various graph properties like density and diameter. Second, *tackling dynamic graphs*: as a graph can evolve over time, a simple idea to process graph updates is to only re-compute the affected prime PPVs, without touching the unaffected ones. Third, *generalizing to other graph algorithms*: it is promising to apply the same principle of partitioning and prioritizing tours to other random walk-based algorithms, such as the hitting and commute time measures.

References

1. R. Andersen, F. Chung, and K. Lang. Local graph partitioning using pagerank vectors. In *FOCS*, pages 475–486, 2006.
2. R. Baeza-Yates and A. Tiberi. Extracting semantic relations from query logs. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 76–85, 2007.
3. B. Bahmani, K. Chakrabarti, and D. Xin. Fast personalized PageRank on MapReduce. In *SIGMOD*, pages 973–984, 2011.
4. B. Bahmani, A. Chowdhury, and A. Goel. Fast incremental and personalized PageRank. *VLDB*, pages 173–184, 2010.
5. A. Balmin, V. Hristidis, and Y. Papakonstantinou. ObjectRank: Authority-based keyword search in databases. In *VLDB*, pages 564–575, 2004.
6. P. Berkhin. Bookmark-coloring algorithm for personalized pagerank computing. *Internet Mathematics*, 3(1):41–62, 2006.
7. P. Boldi and S. Vigna. The webgraph framework I: compression techniques. In *WWW*, pages 595–602, 2004.
8. U. Brandes, M. Gaertler, and D. Wagner. Experiments on graph clustering algorithms. In *In 11th Europ. Symp. Algorithms*, pages 568–579. Springer-Verlag, 2003.
9. M. Brinkmeier, J. Werner, and S. Recknagel. Communities in graphs and hypergraphs. In *Proceedings of the Sixteenth ACM Conference on Conference on Information and Knowledge Management*, CIKM '07, pages 869–872, New York, NY, USA, 2007. ACM.
10. S. Chakrabarti. Dynamic personalized pagerank in entity-relation graphs. In *WWW*, pages 571–580, 2007.
11. S. Chakrabarti, A. Pathak, and M. Gupta. Index design and query processing for graph conductance search. *VLDBJ*, 20:445–470, 2010.
12. D. Fogaras, B. Rácz, K. Csalogány, and T. Sarlós. Towards scaling fully personalized pagerank: Algorithms, lower bounds, and experiments. *Internet Mathematics*, 2(3):333–358, 2005.
13. Y. Fujiwara, M. Nakatsuji, T. Yamamuro, H. Shiokawa, and M. Onizuka. Efficient personalized pagerank with accuracy assurance. In *SIGKDD*, pages 15–23, 2012.
14. M. Gupta, A. Pathak, and S. Chakrabarti. Fast algorithms for top-k personalized pagerank queries. In *WWW*, pages 1225–1226, 2008.
15. T. H. Haveliwala. Topic-Sensitive PageRank: a Context-Sensitive ranking algorithm for web search. *TKDE*, 15(4):784–796, 2003.

16. G. Jeh and J. Widom. Scaling personalized web search. In *WWW*, pages 271–279, 2003.
17. S. Kamvar, T. Haveliwala, C. Manning, and G. Golub. Exploiting the block structure of the web for computing PageRank. Technical report, Stanford University, 2003.
18. G. L. Nemhauser, L. A. Wolsey, and M. L. Fisher. An analysis of approximations for maximizing submodular set functionsi. *Mathematical Programming*, 14(1):265–294, 1978.
19. L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford University, 1999.
20. A. Papoulis, S. Pillai, and S. Unnikrishna. *Probability, random variables, and stochastic processes.* McGraw-hill New York, 1965.
21. A. Pathak, S. Chakrabarti, and M. Gupta. Index design for dynamic personalized pagerank. In *ICDE*, pages 1489–1491, 2008.
22. K. H. Randall, R. Stata, R. G. Wickremesinghe, and J. L. Wiener. The link database: Fast access to graphs of the web. In *DCC*, pages 122–131, 2002.
23. M. Richardson and P. Domingos. The intelligent surfer: Probabilistic combination of link and content information in pagerank. In *NIPS*, pages 1441–1448, 2002.
24. P. Sarkar and A. Moore. Fast nearest-neighbor search in disk-resident graphs. In *SIGKDD*, pages 513–522, 2010.
25. C. Silverstein, H. Marais, M. Henzinger, and M. Moricz. Analysis of a very large web search engine query log. *SIGIR Forum*, 33(1):6–12, Sept. 1999.

A Proof of Theorems

Theorem 2 After iteration-k, the L1 error $\varphi^{(k)}$ as defined in Eq. 5 satisfies the following bound:

$$\varphi^{(k)} \leq (1-\alpha)^{k+2}$$

Proof First, by Eq. 6 and 3, we have the following:

$$\begin{aligned}\varphi^{(k)} &= 1 - \sum_p \hat{\mathbf{r}}_q^{(k)}(p) \\ &= 1 - \sum_{i=0}^{k} \sum_{t \in T^i} R(t). \qquad (26)\end{aligned}$$

Second, by Def. 1, $\forall t \in T^k, \mathcal{L}_h(t) = k$, and $\forall t, \mathcal{L}_h(t) < \mathcal{L}(t)$ where $\mathcal{L}(t)$ is the natural length of t (*i.e.*, the number of edges in t). Thus, if $\mathcal{L}(t) \leq k+1$, then $\mathcal{L}_h(t) \leq k$, implying $\cup_{i=0}^{k} T^i \supseteq \cup_{i=0}^{k+1} S^i$ where $S^i \triangleq \{t : \mathcal{L}(t) = i\}$. Hence the following:

$$\sum_{i=0}^{k} \sum_{t \in T^i} R(t) \geq \sum_{i=0}^{k+1} \sum_{t \in S^i} R(t). \qquad (27)$$

Third, we claim that

$$\sum_{t \in S^i} R(t) = (1-\alpha)^i \alpha, \qquad (28)$$

which can be shown by induction. The base case $i = 0$ is clearly true. In the induction step, suppose it is true for $i = \ell$. All tours with length $\ell + 1$ must be extended from a tour of length ℓ by one step. Consider a particular tour t' of length ℓ. The total reachability of all tours of length $\ell+1$ that are extended from t' is $R(t')(1-\alpha)$ based on Eq. 2. Hence, $\sum_{t \in S^{\ell+1}} R(t) = \sum_{t' \in S^\ell} R(t')(1-\alpha) = (1-\alpha)^\ell \alpha (1-\alpha) = (1-\alpha)^{\ell+1}\alpha$, which proves the claim.

Finally, combining these results (Eq. 26, 27 and 28), we can derive that

$$\begin{aligned}\varphi^{(k)} &= 1 - \sum_{i=0}^{k} \sum_{t \in T^i} R(t) \\ &\leq 1 - \sum_{i=0}^{k+1} \sum_{t \in S^i} R(t) \\ &= 1 - \sum_{i=0}^{k+1} (1-\alpha)^i \alpha,\end{aligned}$$

which simplifies to $\varphi^{(k)} \leq (1-\alpha)^{k+2}$. □

Theorem 5 Let $|V_{C_i}|$ be the number of nodes in community C_i, then $I(T_{C_i}) \approx |V_{C_i}|$.

Proof We derive this computation of $I(T_{C_i})$ step by step as follows:

$$\begin{aligned}I(T_{C_i}) &=^1 \sum_{t \in T_{C_i}; \mathcal{L}(t) \leq k_i} R(t) \\ &=^2 \sum_{t \in T_{C_i}} \prod_{\mathcal{L}(t)=1}^{k_i} \frac{1}{d_i} \cdot \alpha \cdot (1-\alpha)^{\mathcal{L}(t)-1} \\ &=^3 \sum_{\mathcal{L}(t)=1}^{k_i} |V_{C_i}| \cdot d_i^{\mathcal{L}(t)} \cdot \frac{1}{d_i}^{\mathcal{L}(t)} \cdot \alpha \cdot (1-\alpha)^{\mathcal{L}(t)} \\ &=^4 |V_{C_i}| \cdot \alpha \cdot \sum_{\mathcal{L}(t)=1}^{k_i} (1-\alpha)^{\mathcal{L}(t)} \\ &\approx^5 |V_{C_i}|\end{aligned}$$

First, we define the importance of C_{T_i} as the overall importance of all tours with length no longer than k_i in *step 1*. Here, we apply a upper bound k_i on the length of tours to avoid those tours with infinite length in case C_{T_i} is cyclic; if C_{T_i} is acyclic, k_i simply equals to the length of longest tours in it. Next, in *step 2*, we group these tours by their length $\mathcal{L}(t)$ so that we can calculate the importance of tours of each length (from 1 to k_i) according to the P-inverse distance definition. Subsequently, we approximate the number of tours at each $\mathcal{L}(t)$ using the average outdegree d_i. Specifically, for each arbitrary node $q \in C_{T_i}$, there are d_i length-1 tours starting at q in C_{T_i}; for any of q's neighbors, it has d_i out-neighbors again, constituting d_i^2 length-2 tours from q. Generally, there are $d_i^{\mathcal{L}(t)}$ length-$\mathcal{L}(t)$ tours starting from an arbitrary node q, and thus in C_{T_i} which contains $|V_i|$ nodes, the total number of length-$\mathcal{L}(t)$ tours is $|V_{C_i}| \cdot d_i^{\mathcal{L}(t)}$. We thus reformulate the overall importance by the number and importance of each length-$\mathcal{L}(t)$ tours in *step 3*. In *step 4*, we eliminate the same factors in the formula and have $I(C_{T_i}) = |V_i| \cdot \alpha \cdot \sum_{\mathcal{L}(t)=1}^{k_i} (1-\alpha)^{\mathcal{L}(t)}$. Since $1-\alpha$ is smaller than 1, we can always find a x' such that for all $\mathcal{L}(t) > x'$, $(1-\alpha)^{x'} \approx 0$. Thus, we can finally have $I(C_{T_i}) \approx |V_{T_i}|$ in *step 5*. □