Accelerating Topic-Sensitive PageRank by Exploiting the Query History

Shufeng Gong, Zhixin Zhang, Jing Lu, Yanfeng Zhang(✉), Cong Fu, and Ge Yu

School of Computer Science and Engineering, Northeastern University, Shenyang, China
{gongsf, zhangyf, fuchong, yuge}@mail.neu.edu.cn
{yinshi, lujing}@stumail.neu.edu.cn

Abstract. Topic-Sensitive PageRank (TSPR) is a widely used algorithm in recommender systems and machine learning. However, the TSPR query is time-consuming as it requires multiple rounds of iterative computation. To accelerate the TSPR query, we propose an efficient TSPR query algorithm, FasTSPR, that accelerates the TSPR query by exploiting the previous TSPR query. Specifically, FasTSPR borrows the computation result of the previous TSPR query when performing the current TSPR query, avoiding redundant calculations and thus accelerating the TSPR query.

Keywords: Topic-Sensitive PageRanak · Query History · Acceleration.

1 Introduction

In a graph, it is essential to compute the relevance or importance score of each vertex with respect to a particular topic, which is often widely applied in recommender systems [5] and Graph Learning [6], where the vertex represents the webpage in the networking, the author in the citation network, the person in the social network, and so on. Topic-Sensitive PageRank (TSPR) [2] is a variant of the PageRank algorithm that is specifically designed to compute the relevance or importance score of each vertex with respect to a particular topic. The score of each vertex for a particular topic is computed by calculating the similarities of it to all vertices belonging to the topic. Different from PageRank, which results in only one score vector, TSPR generates a score vector for each topic, where the score S_{ij} in the score vector $\overrightarrow{S_i}$ is the score of j-th vertex for i-th topic. TSPR enables users to receive search results that are more aligned with their specific topic preferences, enhancing the personalization of search results.

To return the relevance score of a vertex to a specific topic efficiently, we can precompute the score vector of each topic. However, as the amount of data and data type diversity increase, the number of topics in graphs is also increasing. For example, in citation networks, topics include computer vision, machine translation, information retrieval, programming languages, graph computation, graph matching, graph learning, etc. And with the continuous development of new technologies, the number of topics is constantly increasing. It means that

a large number of score vectors need to be stored for enormous topics, and the dimension of each vector is equal to the number of vertices, which is obviously not feasible because it requires a large amount of storage space.

Based on the above discussion, it is necessary to compute the score vector online for each query topic. However, similar to PageRank, the computation of TSPR requires multiple rounds of iterative computation to update the score of each vertex, which is time-consuming. TSPR can be accelerated by returning an approximate result [4, 8, 7], but these methods can not be used in some high-precision situations. We can also accelerate TSPR by constructing some indexes in advance [7, 3], but the indexes will be invalid when the graph changes. Therefore, it is necessary to find a method that accelerates TSPR without using indexes and get high-precisions results.

To compute TSPR efficiently, we propose an efficient TSPR, FasTSPR, which is accelerated by leveraging the previous TSPR query. The design of FasTSPR is based on the observation that there is a large amount of overlap when computing score vectors for two different topics. When we have obtained the score vector with respect to one topic, we can use these overlap computations to accelerate the computation of the TSPR query for another topic. However, it is difficult to identify the overlapped computations, if the intermediate results are not memoized in each round of iterative computation. While the memoized intermediate results often bring the overhead of storing and updating, which reduces the efficiency of TSPR.

In order to eliminate non-duplicated calculations, we use the forward push method to perform TSPR. When a TSPR query is triggered, a message is sent from each vertex belonging to the topic, and then these messages are propagated on the graph, meanwhile, the value of the messages are decayed during the propagation. The value of the accumulated received messages for each vertex refers to the score of the vertex for the topic. Based on this computation method, when performing another TSPR query with a new topic, a message is sent from each vertex belonging to the topic, and a negative message is sent from the vertex belonging to the previous topic. The positive message and the negative message are propagated on the graph at the same time. For non-overlapped computations, negative messages will make them invalid for the current topic. For the overlapped computations, the positive and negative messages will be offset and will not continue to be propagated, thereby avoiding redundant computations.

The contributions of this paper can be summarized as follows,

- We propose an efficient TSPR algorithm, FasTSPR, that accelerates TSPR queries by leveraging the previous TSPR query.
- We provide a formal proof to prove the correctness of FasTSPR.
- We evaluate the efficiency FasTSPR on five real graphs with comprehensive experiments. The results show that FasTSPR can achieve up to 5.44× speed up over the traditional TSPR algorithm and up to 1.37× speed up over the state-of-the-art algorithm.

2 Preliminary

In this section, we will provide some fundamental definitions and background knowledge regarding TSPR.

2.1 Topic-Sensitive PageRank

Let $G = (V, E)$ be a directed graph with vertex set V and edge set E, $t \in T$ represent one specific topic of the topic set, where T is the topic set. There is a set of vertices $V_t \subseteq V$ belonging to the topic t. Note that, a vertex may belong to more than one topic. Given a decay factor $\alpha < 1$, the TSPR for t can be interpreted as a random walk on G that starts from the vertices V_t belonging to t and then iteratively jumps to a randomly chosen out-neighbor with probability $1 - \alpha$ or teleports to V_t with probability α. The score value of each vertex v_i is the probability that such random walk starts from vertices V_t belonging to t and stops at v_i. TSPR query returns a score vector $\overrightarrow{S}$ for topic t, where the i-th value S_i in $\overrightarrow{S}$ is the relevance score of j-th vertex with respect to the topic t.

Note that, since we are not able to store all the score vector $\overrightarrow{\mathcal{S}} = \overrightarrow{S_1}, \overrightarrow{S_2}, \cdots$, we only store on score vector $\overrightarrow{S}$ that is computed only when TSPR query is triggered. Therefore, in this paper, we use $\overrightarrow{S}$ to represent the score vector returned by the TSPR query of any topic t.

2.2 Power Iteration

We denote $\boldsymbol{A}$ as the adjacent matrix of G and D as the diagonal matrix, where $\boldsymbol{A}_{ij} = 1$ if there is an edge between i-th and j-th vertex, otherwise $\boldsymbol{A}_{ij} = 0$, and $\boldsymbol{D}_{ij} = 1$ if $i = j$, otherwise $\boldsymbol{D}_{ij} = 0$. Given the initial score vector $\overrightarrow{S}$ with a random value in each dimension, the power iteration is an iterative algorithm for solving the following Equation 1

$$\overrightarrow{S} = (1 - \alpha) \cdot \overrightarrow{S} \cdot \boldsymbol{P} + \alpha \cdot \overrightarrow{e}, \tag{1}$$

where $\boldsymbol{P} = \boldsymbol{A}^T \cdot \boldsymbol{D}^{-1}$ is the transition matrix of G, and $\overrightarrow{e}$ is the indicator topic vector, *i.e.*, $e_i = \frac{1}{|V_t|}$ if the i-th vertex belongs to topic t, otherwise $e_i = 0$. We denote the score vector for topic t_i as $\overrightarrow{S}_i^*$, the Equation 1 iteratively refines $\overrightarrow{S}$ closer to $\overrightarrow{S}^*$, *i.e.*, $||\overrightarrow{S}^* - \overrightarrow{S}||_1$ decreases with iterations, where $||\overrightarrow{\cdot}||_1$ is the l_1 normal of vector $\overrightarrow{\cdot}$. Since it is impossible to obtain $\overrightarrow{S_i}^*$ before the TSPR query, the iterative computation terminates at $||\overrightarrow{S}^{k-1} - \overrightarrow{S}^k||_1 < \epsilon$, where $\overrightarrow{S}^k$ is the score vector after k rounds of iterative computation, ϵ is an extremely small value.

Though *Power Iteration* can return TSPR query with high-precision results, it performs less efficiently. From Equation 1, it can be seen that each iteration involves the multiplication of a vector and a matrix, as well as the addition between vectors. However, some computations within the multiplication of vector and matrix make only minimal contributions to the convergence of the TSPR query.

2.3 Forward Push

Forward Push [1] is an efficient method to perform TSPR queries. In Forward Push, in addition to the score vector $\overrightarrow{S_i}$, there is another residual vector $\overrightarrow{r}$ to be kept, r_i means the sum of the probability of random walkers who are still alive and staying at the j-th vertex.

Initially, each element in $\overrightarrow{S}$ is set as "0", and $r_i = \frac{1}{|V_t|}$ if the i-th vertex belongs topic t, otherwise $r_i = 0$. During the iterative computation process, the α portion residual value of each vertex is added to its score S_i, *i.e.*, $S_i = S_i + \alpha \cdot r_i$, then the $1-\alpha$ portion of the residual is evenly pushed to its every outgoing neighbor $v_l \in OUT(v_i)$, *i.e.*, $r_l = r_l + \frac{(1-\alpha)\cdot r_i}{|OUT(v_i)|}$, where $|OUT(v_i)|$ is the outdegree of the i-th vertex. After that, the residual value is reset as "0". The iterative computation terminates at $||\overrightarrow{r_i}||_1 < \epsilon$. It has been proven that the *Forward Push* and the *Power Iteration* can return the same high-precision results [9] within the time complexity $O(m \cdot log\frac{1}{\epsilon})$, where m is the number of edges in G.

However, there are some inefficient computations during the iterations. For example, when the residual value r_{ij} of j-th vertex is very small, its impact on the score S is also very small. Both the addition and push operation contribute very little to the convergence of the iterative computation. To avoid inefficient computations, we skip vertices with very small residual values when performing the current iteration. The forward push operation is triggered when the residual value accumulates enough. However, inefficient computations occur during the iterations. For instance, when the residual value r_{ij} of the j-th vertex is exceedingly small, its influence on the score S_{ij} is minimal. Both addition and push operations contribute minimally to the convergence of the iterative computation. To avoid inefficient computations, we skip the vertices with very small residual values during the current iteration. In the next iterations, the forward push on v_j may be activated once its residual value accumulates sufficiently. The specific details of *Push Forward* are shown in Algorithm 1.

From Algorithm 1, it can be seen when r_{ij} is very small, the score of $_j$ is not been updated (line 6), so some inefficient computations are avoided to improve compute efficiency. The proposed FasTSPR algorithm in this paper is also built on top of the forward push calculation method.

3 FasTSPR

In this section, we will propose an efficient TSPR algorithm, FasTSPR, by exploiting the previous TSPR query. Before introducing it, we first provide the intuition behind the FasTSPR.

3.1 Inutition behind FasTSPR

From the process of Forward Push, it can be seen that Forward Push is an accumulative iterative algorithm, *i.e.*, the score value of each vertex is accumulated from the received residuals. When a new TSPR query is triggered for another

Algorithm 1 Forward Push

Input: Graph G, damping factor α, topic t, l_1 error ϵ;
Output: $\overrightarrow{S}$

for each vertex $v_i \in V_t$ **do**
    $r_i = \frac{1}{|V_t|}$;
    $S_i = 0$;
end for
while True **do**
    for each vertex v_i with $r_i > \frac{\epsilon}{|V|}$ **do**
        $S_i = S_i + r_i * \alpha$;
        for each $v_l \in OUT(v_i)$ **do**
            $r_l = r_l + \frac{r_i \cdot (1-\alpha)}{|OUT(v_i)|}$;
        end for
        $r_i = 0$;
    end for
    if $\sum_{i=1}^{|V|} r_i \leq \epsilon$ **then**
        break;
    end if
end while
return $\overrightarrow{S}$;

topic, we have to perform Forward Push from scratch by resetting the score vector to "$\overrightarrow{0}$" and residual vector $\overrightarrow{r}$.

We found that while the residuals originate from different vertices in the previous TSPR query and the current one, there exists a considerable of residuals, bearing the same values, propagating along the same edges, *i.e.*, lots of the computations between two TSPR queries are overlapped. If we borrow the computations from the previous TSPR query, the current TSPR query can reduce some unnecessary recomputation, then the current TSPR query will be accelerated significantly.

Example 1. Consider the following scenario, given a vertex v_i, and it receives two residuals with the same value in both TSPR queries, denotes r_i' and r_i'', and $r_i' = r_i''$. Since we have processed r_i' (including $S_i = S_i + r_i'$ and push r_i' to v_i's outgoing neighbors) in the previous TSPR query, it is unnecessary to reprocess r_i'' in the current TSPR query if we borrow the same process from the previous TSPR query. Because these two residuals have the same values and the same effect on the score.

During the iterative computation for two different TSPR queries, even if the residual values passing through the same edge are similar but not identical. It would be exciting if we can accelerate iterative computation by exploring their similarities.

However, it is difficult to determine which computations from the previous TSPR query can be utilized by the current one. This complexity arises from the fact that, in the previous TSPR query, all received residuals at each vertex have been aggregated into the score. Consequently, identifying which segment

of the score can be shared by two TSPR queries-specifically, the portion of the score computed through overlapping computations in the two queries-poses a significant difficulty.

To overcome this difficulty, we think about the problem from another angle, *i.e.*, from "identifying the overlapped computations" to "eliminating the non-overlapped computations". Continue to consider the vertex v_i in Example 1, in addition to the residual with the same values received in both two TSPR queries, v_i also received some distinct residuals from the previous query. To eliminate these distinct residuals, we can use the same residuals with "negative" values to eliminate their effect on the score. In this way, the score of each vertex from the previous TSPR query can be reused in the current one since we have eliminated the effect of non-overlapped computations on the score.

3.2 Details of FasTSPR

From the above discussion, we propose the FasTSPR. When a new TSPR query is triggered, we take the score computed by the previous TSPR query as the initial score vector of the current one. For the residual vector $\overrightarrow{r}$, we use the following two steps to set it,

- $r_i = r_i - \frac{1}{|V_{t_p}|}$ if $v_i \in V_{t_p}$ belongs to the previous topic t_1,
- $r_i = r_i + \frac{1}{|V_{t_c}|}$ if $v_i \in V_{t_c}$ belongs to the current topic t_c.

where t_p is the topic of the previous TSPR query, and t_c is the topic of the current one. The details of FasTSPR are described in Algorithm 2.

Note that compared with Algorithm 1, Algorithm 2 is not only different in the initialization of the score vector and residual vector but also different in filtering the residuals with small values. Since residuals may be positive or negative in FasTSPR, the absolute values of residuals are used to filter inefficient computations.

During the iterative computation of Algorithm 2, there are both positive and negative residuals propagating on G. When the positive and negative residuals meet, they will be canceled or cut significantly due to the aggregation operation (line 11), and the absolute values of the residuals will quickly become smaller after aggregation, thereby accelerating the convergence of the TSPR query.

3.3 Correctness of FasTSPR

It can be seen from the above analysis that the Algorithm 2 convergence is accelerated by the aggregation of positive and negative residuals. But is the final result correct? Next, we are going to answer this question.

From the Algorithm 1, it can be seen that the essential of residual propagation is the multiplication of the residual and the transfer matrix, *i.e.*, $(1-\alpha)\cdot\overrightarrow{r}\cdot\boldsymbol{P}$, and accumulating it into S. Then, without considering the inefficient computation

Algorithm 2 FasTSPR

Input: Graph G, damping factor α, topic t_c, l_1 error ϵ, score vector $\overrightarrow{S}$ and the residual vector $\overrightarrow{r}$ obtained from TSPR query for t_p.
Output: $\overrightarrow{S}$

for each vertex $v_i \in V_{t_p}$ **do**
  $r_i = r_i + \frac{1}{|V_{t_p}|}$;
end for
for each vertex $v_i \in V_{t_c}$ **do**
  $r_i = r_i + \frac{1}{|V_{t_c}|}$;
end for
while True **do**
  for each vertex v_i with $|r_i| > \frac{\epsilon}{|V|}$ **do**
    $S_i = S_i + r_i * \alpha$;
    for each $v_l \in OUT(v_i)$ **do**
      $r_l = r_l + \frac{r_i * (1-\alpha)}{|OUT(v_i)|}$;
    end for
    $r_i = 0$
  end for
  if $\sum_{i=1}^{|V|} r_i \leq \epsilon$ **then**
    break;
  end if
end while
return $\overrightarrow{S}$;

filterations, the Forward Push can be formalized into the following equation.

$$\overrightarrow{S}^k = \overrightarrow{S}^{k-1} + \alpha \cdot \overrightarrow{r}^k \tag{2}$$

$$\overrightarrow{r}^k = (1-\alpha)\overrightarrow{r}^{k-1} \cdot \boldsymbol{P}. \tag{3}$$

Combine the Equation 2 and the Equation 3, then we have

$$\begin{aligned}
\overrightarrow{S}^k =& \overrightarrow{S}^{k-1} + \alpha \cdot (1-\alpha) \cdot \overrightarrow{r}^{k-1} \cdot \boldsymbol{P} \\
=& \overrightarrow{S}^{k-2} + \alpha \cdot \overrightarrow{r}^{k-2} \cdot \boldsymbol{P} + \alpha \cdot (1-\alpha)^2 \cdot \overrightarrow{r}^{k-2} \cdot \boldsymbol{P}^2 \\
=& \overrightarrow{S}^{k-3} + \cdots \\
& \cdots \\
=& \overrightarrow{S}^0 + \alpha \cdot \overrightarrow{r} + \alpha \cdot (1-\alpha) \cdot \overrightarrow{r} \cdot \boldsymbol{P} + \cdots \\
& + \alpha \cdot (1-\alpha)^{k-1} \cdot \overrightarrow{r} \cdot \boldsymbol{P}^{k-1}
\end{aligned} \tag{4}$$

For the previous topic t_p, the TSPR returns score vector $\overrightarrow{S}_p^k$ after k round of iterative computations. For the current topic t_c, the initial score vector and residual vector are set as $\overrightarrow{S}_c^0 = \overrightarrow{S}_p^k$ and $\overrightarrow{r}_c^0 = \overrightarrow{r}_c^0 + \overrightarrow{r}_p^k - \overrightarrow{r}_p^0$ respectively according to Algorithm 2. After the first iteration of FasTSPR, we have

$$\overrightarrow{S}_c^1 = \overrightarrow{S}_p^k + \alpha \cdot \overrightarrow{r}_c \tag{5}$$

After the k-th iteration, according to the Equation 4, we have

$$\begin{aligned}\overrightarrow{S_c}^k =& \overrightarrow{S_p}^k + \alpha \cdot \overrightarrow{r_c} + \alpha \cdot (1-\alpha) \cdot \overrightarrow{r_c} \cdot \boldsymbol{P} + \cdots \\ &+ \alpha \cdot (1-\alpha)^{k-1} \cdot \overrightarrow{r_c} \cdot \boldsymbol{P}^{k-1}\end{aligned} \tag{6}$$

Unfold the $\overrightarrow{S_p}^k$ according to Equation 4, then we have

$$\begin{aligned}\overrightarrow{S_c}^k =& \overrightarrow{S}^0 + \alpha \cdot \overrightarrow{r_p} + \alpha \cdot (1-\alpha) \cdot \overrightarrow{r_p} \cdot \boldsymbol{P} + \cdots \\ &+ \alpha \cdot (1-\alpha)^{k-1} \cdot \overrightarrow{r_p} \cdot \boldsymbol{P}^{k-1} \\ &+ \alpha \cdot \overrightarrow{r_c} + \alpha \cdot (1-\alpha) \cdot \overrightarrow{r_c} \cdot \boldsymbol{P} + \cdots \\ &+ \alpha \cdot (1-\alpha)^{k-1} \cdot \overrightarrow{r_c} \cdot \boldsymbol{P}^{k-1}\end{aligned} \tag{7}$$

Replace $\overrightarrow{r_c}$ with $\overrightarrow{r_c} = \overrightarrow{r_c} + \overrightarrow{r_p}^k - \overrightarrow{r_p}$, then we have

$$\begin{aligned}\overrightarrow{S_c}^k =& \overrightarrow{S}^0 + \alpha \cdot \overrightarrow{r_p} + \alpha \cdot (1-\alpha) \cdot \overrightarrow{r_p} \cdot \boldsymbol{P} + \cdots \\ &+ \alpha \cdot (1-\alpha)^{k-1} \overrightarrow{r_p} \cdot \boldsymbol{P}^{k-1} \\ &+ \alpha \cdot \overrightarrow{r_c} + \alpha \cdot \overrightarrow{r_p}^k - \alpha \cdot \overrightarrow{r_p} \\ &+ \alpha \cdot (1-\alpha) \cdot \overrightarrow{r_c} \cdot \boldsymbol{P} + \alpha \cdot (1-\alpha) \cdot \overrightarrow{r_p}^k \cdot \boldsymbol{P} \\ &- \alpha \cdot (1-\alpha) \cdot \overrightarrow{r_p} \cdot \boldsymbol{P} + \cdots \\ &+ \alpha \cdot (1-\alpha)^{k-1} \cdot \overrightarrow{r_c} \cdot \boldsymbol{P}^{k-1} + \alpha \cdot (1-\alpha)^{k-1} \overrightarrow{r_p}^k \cdot \boldsymbol{P}^{k-1} \\ &- \alpha \cdot (1-\alpha)^{k-1} \overrightarrow{r_p} \cdot \boldsymbol{P}^{k-1} \\ =& \overrightarrow{S}^0 + \alpha \cdot \overrightarrow{r_c} + \alpha \cdot \overrightarrow{r_p}^k \\ &+ \alpha \cdot (1-\alpha) \cdot \overrightarrow{r_c} \cdot \boldsymbol{P} + \alpha \cdot (1-\alpha) \overrightarrow{r_p}^k \cdot \boldsymbol{P} + \cdots \\ &+ \alpha \cdot (1-\alpha)^{k-1} \cdot \overrightarrow{r_c} \cdot \boldsymbol{P}^{k-1} + \alpha \cdot (1-\alpha)^{k-1} \overrightarrow{r_p}^k \cdot \boldsymbol{P}^{k-1}\end{aligned} \tag{8}$$

Since the $\alpha \cdot \overrightarrow{r_p}^k$ is too small to be ignored, then we have

$$\begin{aligned}\overrightarrow{S_c}^k =& \overrightarrow{S}^0 + \alpha \cdot \overrightarrow{r_c} \\ &+ \alpha \cdot (1-\alpha) \cdot \overrightarrow{r_c} \cdot \boldsymbol{P} + \cdots \\ &+ \alpha \cdot (1-\alpha)^{k-1} \cdot \overrightarrow{r_c} \cdot \boldsymbol{P}^{k-1}\end{aligned} \tag{9}$$

Otherwise, the iterative computation will be performed continuously and generate $-\alpha \cdot \overrightarrow{r_p}^k$ to eliminate $\alpha \cdot \overrightarrow{r_p}^k$. It means that after k iterations, FasTSPR and Forward Push return the same result with high precision. Note that equation 9 is a high-precision approximation of equation 8. In practice, during the execution of the FasTSPR, $\alpha \cdot \overrightarrow{r_p}^l, (l \geq k)$ are not be removed and will be eliminated by $-\alpha \cdot \overrightarrow{r_p}^l, (l \geq k)$ generated in the subsequent iterative computations of subsequent iterations. Therefore, even if the topics are changed multiple times, we can still get high-precision results.

4 Experiment

4.1 Experimental Setup

In this section, we present the experimental evaluation of FasTSPR. By default, the experiments are performed on a commodity PC with 5.4GHz Intel Core i9 CPU, 32GB memory, and it runs on 64-bit Ubuntu 22.04 with compiler GCC 11.3.

Dataset Our experiments involve five real graphs, DBLP, Catster, Wiki-Talk, cit-Patents, and Google. Table 1 shows the detailed information of these graphs. Since the vertex belonging to the same topic is similar, thus the vertices in each topic come from the same cluster of the graph. In our experiments, each topic contains about 50 vertices by default.

Table 1: Datasets

Graph	# of vertex	# of edge
Catster (CT)	149684	10896394
DBLP (DL)	317080	2099732
Google (GL)	916428	12156500
Wiki-Talk (WT)	2394385	8505513
cit-Patents (CP)	3774768	16518948

Competitors We compare FasTSPR with traditional *Power Iteration* (PowItr), *Forward Push* (FwdPush), and a hybrid method *PowerPush* [9] that incorporated the strengths of both PowItr and FwdPush. In our experiment, we use the same l_1-error threshold $\epsilon = 10^{-9}$ for all competitors and FasTSPR.

4.2 Overall Performance

We first compare FasTSPR with competitors in terms of the average runtime on different graphs in Table 1. Figure 1 reports the normalized runtime of each algorithm, where FasTSPR is treated as the baseline. It can be seen that the FasTSPR outperforms other algorithms in all cases. Specifically, FasTSPR can achieve 2.42× speed up on average (up to 2.65×) over PowerIter, 3.22× speed up on average (up to 5.44×) over FwdPush, and 1.21× speed up on average (up to 1.37×) over PowerPush. It is notable that both PowerPush and FasTSPR outperform PowerItr and FwdPush because of some optimizations equipped in PowerPush, e.g., asynchronous pushes, hybrid global sequential scan and local random access, dynamic l_1-error threshold. Our FasTSPR outperforms PowerPush because of the utilization of query history, which reduces some redundant computations.

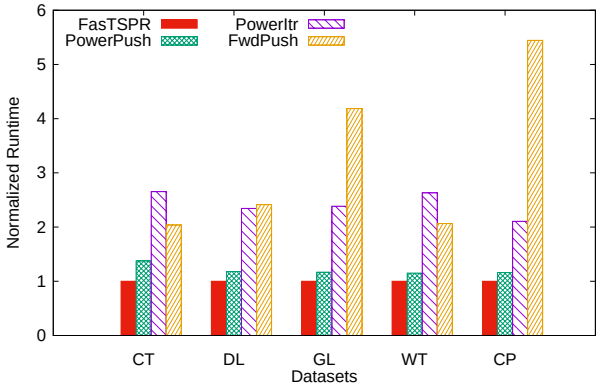
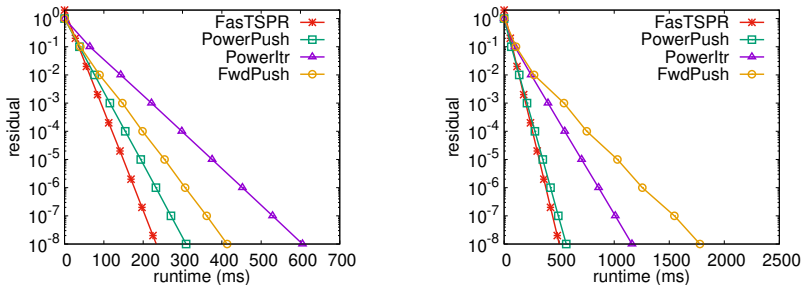


Fig. 1: The comparison of runtime

4.3 Convergence comparison

To evaluate the effectiveness of FasTSPR on accelerating the convergence of TSPR queries, we compare the convergence rates of FasTSPR and the competitors. We use the sum of residual values as the metric to measure the distance to convergence, i.e., $|\sum_{i=1}^{|V|} r_i$. Fig. 2 shows the trend of the sum of residual values over time. It can be seen that the residual in the FasTSPR algorithm decreases fastest and achieves to convergence state first. Note that the sum of the initial residual values of FasTSPR is 2 since there are positive and negative residuals in the current and previous topic vertices.

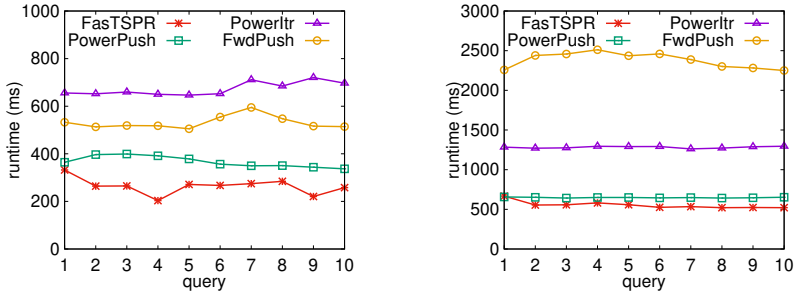


(a) CT (b) GL

Fig. 2: The comparison of convergence speed

4.4 The impact of different TSPR queries

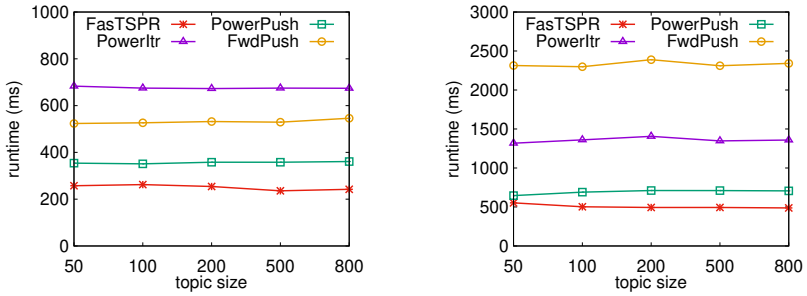
In order to test the impact of different TSPR queries on FasTSPR, we randomly performed 10 TSPR queries continuously and recorded the runtime of each query.



(a) CT (b) GL

Fig. 3: The runtime when varying the topics

Fig. 3 shows the runtime of FasTSPR and competitors when performing different TSPR queries on Catster (CT) and Google (GL) graphs. It can be seen that the FasTSPR outperforms other algorithms in all queries, except for the first query. This is because there is no previous query to be used for the first query, so its runtime is the same as that of PowerPush.



(a) CT (b) GL

Fig. 4: The runtime when varying the size of topics

4.5 The impact of topic size

To test the impact of the size of the topic, i.e., the number of vertices within the topic for TSPR, on the efficiency of FasTSPR, we performed TSPR queries on topics containing 50, 100, 200, 500 and 800 vertices and recorded their execution times. Fig. 4 shows the runtime of FasTSPR and others when varying the size of topics on Catster (CT) and Google (GL) graphs. It can be seen that the execution time of FasTSPR is smaller than others when varying the topic sizes, and the running time of FasTSPR slowly decreases with the increase of the number of vertices in the topic. This is because the more vertices in the topic, the higher

the possibility of the positive residuals and negative residuals meeting, which results in better acceleration.

5 Conclusion

In this paper, we propose FasTSPR, an efficient Topic-sensitive PageRank (TSPR) algorithm. FasTSPR accelerates the TSPR query by reusing the computation of the previous TSPR query, which can reduce some redundant computation since there are enormous overlapped computations between the current TSPR query and the previous one.

Acknowledgments. This paper is supported by the 111 Project (B16009), the National Natural Science Foundation of China (U2241212, 62072082, 62202088, 62137001, 62272093), Joint Funds of Natural Science Foundation of Liaoning Province (2023-MSBA-078), and Fundamental Research Funds for the Central Universities (N2416011).

References

1. Andersen, R., Chung, F., Lang, K.: Local graph partitioning using pagerank vectors. In: Proceedings of the 47th FOCS. pp. 475–486. IEEE (2006)
2. Haveliwala, T.H.: Topic-sensitive pagerank. In: Proceedings of the 11th WWW. pp. 517–526 (2002)
3. Jung, J., Park, N., Lee, S., Kang, U.: Bepi: Fast and memory-efficient method for billion-scale random walk with restart. In: Proceedings of the 2017 ACM International Conference on Management of Data. pp. 789–804 (2017)
4. Lofgren, P.A., Banerjee, S., Goel, A., Seshadhri, C.: Fast-ppr: Scaling personalized pagerank estimation for large graphs. In: Proceedings of the 20th SIGKDD. pp. 1436–1445 (2014)
5. Lü, L., Medo, M., Yeung, C.H., Zhang, Y.C., Zhang, Z.K., Zhou, T.: Recommender systems. Physics reports **519**(1), 1–49 (2012)
6. Wang, H., Yang, R., Huang, K., Xiao, X.: Efficient and effective edge-wise graph representation learning. In: Proceedings of the 29th SIGKDD. pp. 2326–2336 (2023)
7. Wang, S., Tang, Y., Xiao, X., Yang, Y., Li, Z.: Hubppr: Effective indexing for approximate personalized pagerank. Proceedings of the VLDB Endowment **10**(3) (2016)
8. Wang, S., Yang, R., Xiao, X., Wei, Z., Yang, Y.: Fora: simple and effective approximate single-source personalized pagerank. In: Proceedings of the 23rd SIGKDD. pp. 505–514 (2017)
9. Wu, H., Gan, J., Wei, Z., Zhang, R.: Unifying the global and local approaches: an efficient power iteration with forward push. In: Proceedings of the 2021 International Conference on Management of Data. pp. 1996–2008 (2021)