UNIVERSITY OF TARTU

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

Institute of Computer Science

Volodymyr Floreskul

Memory-Efficient Fast Shortest Path Distance Estimation in Large Graphs

Master's thesis (30 EAP)

Supervisor: Konstantin Tretyakov, MSc

Author: .... "20" May 2013

Supervisor: .... "20" May 2013

Approved for defence

Professor: .... "......" May 2013

TARTU 2013

Acknowledgements

The thesis is a part of a project of the social network analysis group at the Software Technology and Applications Competence Center.

I would like to thank my supervisor Konstantin Tretyakov for his encouragement, continuous guidance and insight on this study. I am extremely grateful to Prof. Marlon Dumas and the whole social network analysis team at STACC for their important feedback, advice and inspiration.

Special thanks to the Skype Labs and especially Ando Saabas for providing datasets and their valuable support during this project.

Contents

1 Introduction **4**

2 The Shortest Path Problem **6**
- 2.1 Graph theory 6
- 2.2 Exact shortest path algorithms 8
- 2.3 Landmark-based approximate algorithms 8
  - 2.3.1 Basic algorithm 8
  - 2.3.2 Landmark-LCA algorithm 9
  - 2.3.3 Landmark-BFS algorithm 10

3 Pruned Landmark-Based Algorithms **13**
- 3.1 Pruned landmark trees 13
- 3.2 Computation time and space complexity 14
- 3.3 Basic method 14
- 3.4 Interlandmark distance approximation 16
- 3.5 Cycle elimination 18
- 3.6 Restricted BFS method 18

4 Experimental Evaluation **23**
- 4.1 Datasets 23
- 4.2 Landmark selection 24
- 4.3 Experimental setup 24
- 4.4 Results 25
  - 4.4.1 Approximation error 25
  - 4.4.2 Query execution time 26
  - 4.4.3 Preprocessing time 30
  - 4.4.4 Memory usage 33
- 4.5 Approximation results for different distances 34

5 Conclusions **36**

Abstract (in Estonian) **38**

A Theorems and proofs **40**

References **43**

Chapter 1

Introduction

The shortest path problem is one of the core problems in graph theory. Effective algorithms have been developed and studied that work well on small and medium-size graphs. In recent years more and more interest is concentrated on large social networks (like Facebook, LinkedIn, Twitter), web and knowledge graphs. They have become important and attractive targets for analysis in both academic and industrial communities. The size of these large graphs makes even basic well-known algorithms hard to apply and graph analysis extremely challenging.

One version of the shortest path problem that is of particular interest in this thesis is a point-to-point (P2P) shortest path problem:

> Given a query consisting of a start node s and end node t, find the path with the minimal length that connects s with t in graph G.

This problem has numerous applications, such as route computation in transportation networks, protein interaction networks in biology, entity-relationship path finding in large-scale knowledge repositories, VLSI design in electronics. In relationship to social networks shortest path distance algorithms can be used for social-sensitive search, when a user is interested in finding other people or in finding content from people that are close to him in the social graph. In addition to that, point-to-point shortest paths can often be used as input to more complex graph analysis algorithms.

All classical exact P2P shortest path algorithms do not scale well on real-world datasets with hundreds of millions of nodes and billions of edges. A full breadth-first search of the Skype social graph from year 2011 with 539 M nodes and 2.2 B edges implemented in Java takes about 75 minutes, which makes this approach almost impossible to apply in many practical situations. Researchers have proposed several approximate methods that have different sets of characteristics in terms of running time, memory usage and average accuracy.

One family of techniques that is simple and scalable at the same time is based on upper bound distance approximation using a fixed set of selected nodes called *landmarks*. The approximation is done using the precomputed index which includes shortest path from each node in the graph to every landmark. Landmark-based methods provide good accuracy while keeping algorithm running time in

order of milliseconds, even on large graphs. Accuracy of these methods can be increased by using more landmarks, but this leads to linear increase of memory usage and preprocessing time with sublinear approximation error reduction.

In this thesis we describe an improvement to the landmark-based technique that can significantly reduce memory usage while keeping comparable accuracy and query running time. The idea of this modification is based on the fact that in majority of cases it is enough to keep only the shortest paths to the closest landmarks rather than the whole landmark set. The proposed improvement allows to use the number of landmarks that is higher by several orders of magnitude compared to previous methods.

In Chapter 2 we briefly describe the main definitions, concepts and classical methods of studying the shortest path problem. Chapter 3 is devoted to the description of the proposed shortest path query answering algorithm with different landmark selection techniques. In Chapter 4 we define evaluation metrics, experimental setup and provide the results of running experiments by using the described algorithms. Chapter 5 summarizes the obtained results and proposes possible future work.

Chapter 2

The Shortest Path Problem

The current work is based on methods and results from the fields of graph theory and social network analysis. While exact methods for finding shortest paths have been studied for decades, the scalable approximation algorithms continue to gain more and more attention from the research community in recent years [1, 7, 3, 13, 14].

2.1 Graph theory

Graph theory is the study of *graphs*, mathematical models of collections of objects and relations between them. For example, when a social network is represented as a graph, then users are mapped to *nodes* and connections between them form *edges*.

We use the following basic graph theory definitions in this work.

Definition 2.1 (Directed Graph). *A directed graph is an ordered pair $G = (V, E)$, where V is a set of nodes or vertices and $E \subseteq V \times V$ is a set of edges.*

Definition 2.2 (Undirected graph). *An undirected graph is a graph in which edges have no orientation. Each edge (a, b) is considered to be identical to (b, a).*

Definition 2.3 (Weighted graph). *A weighted graph is a graph where each edge (a, b) is associated with a real-valued weight $w(a, b)$. In the case when there are no weights assigned to edges of a graph it is referred to as unweighted.*

Definition 2.4 (Path in a graph). *Given two nodes $s, t \in V$, the path between them is defined as a sequence $\pi_{s,t} = (s, u_1, u_2, \ldots, u_{\ell-1}, t)$, where $\{u_1, u_2, \ldots, u_{\ell-1}\} \subseteq V$ and $\{(s, u_1), (u_1, u_2), \ldots, (u_{\ell-1}, t)\} \subseteq E$.*

Definition 2.5 (Cycle). *A cycle is a path for which the start and end nodes are the same.*

Definition 2.6 (Path length). *The length $\ell = |\pi_{s,t}|$ of a path $\pi_{s,t}$ is the number of edges in it (counting repeated edges multiple times).*

Definition 2.7 (Shortest path). *A shortest path between two given nodes s and t is a path in a graph whose length has the lowest value among all paths from s to t.*

There can be multiple shortest paths between two nodes in a graph, but all of them have the same length.

Definition 2.8 (Path concatenation). *The concatenation of two paths $\pi_{s,t} = (s, \dots, t)$ and $\pi_{t,v} = (t, \dots, v)$ is the combined path $\pi_{s,v} = \pi_{s,t} + \pi_{t,v} = (s, \dots, t, \dots, v)$.*

Definition 2.9 (Distance). *The* distance *(shortest path distance) $d(s,t)$ between vertices s and t is length of the shortest path from s to t.*

The distance in a graph satisfies the *triangle inequality*: for any $s, t, u \in V$

$$d(s,t) \leq d(s,u) + d(u,t)\,. \tag{2.1}$$

Inequality (2.1) turns into equality if there exists a shortest path between s and t, which passes through u.

Definition 2.10 (Diameter). *The diameter D of a graph is the maximal length of a shortest path in the graph.*

From social network analysis theory it is known that diameters of social graphs tend to be small [16].

Definition 2.11 (Subgraph). *A subgraph $G' = (V', E')$ of a graph $G = (V, E)$ is a graph where $V' \subseteq V$ and $E' \subseteq E$.*

Definition 2.12 (Induced subgraph). *A subgraph H of a graph G is called induced if for any pair of nodes u and v from S, where S is the set of vertices in H, (u,v) is an edge of H if and only if (u,v) is an edge of G. H can be written as $G[S]$.*

Definition 2.13 (Connected component). *In an undirected graph a connected component is a maximal subgraph in which any two nodes are connected through a path.*

A graph is called **connected** if all its nodes are in the same connected component.

Definition 2.14 (Tree). *A tree is an undirected graph that is connected and has no cycles.*

Definition 2.15 (Shortest path tree). *A shortest path tree (SPT) T rooted at node u in graph G is a tree that connects u to all nodes in the graph $v \in V$ and the path distance from root u to any v is the shortest path distance from u to v in G.*

2.2 Exact shortest path algorithms

The most basic algorithm for finding shortest path distances from a single node (source) to other nodes in unweighted graph is *breadth-first search* (BFS), see Algorithm 1. It begins with a given start node and inspects (visits) its neighborhood. Then for each of those nodes in turn it checks their neighboring nodes that have not yet been visited. The algorithm stops when the destination node is found or when all the reachable nodes have been inspected. In the worst case BFS works in $O(|V| + |E|)$ time and uses $O(|V|)$ space.

In weighted graphs BFS does not guarantee an optimal solution. In this case the problem can be solved by applying the Dijkstra's algorithm [4]. The original version based on binary heap works in $O((|V| + |E|) \log |V|)$ time, but the version with Fibonacci heap has complexity $O(|E| + |V| \log |V|)$.

Other important single-source shortest path methods include the A* algorithm [12], which is a generalization of the technique proposed by Dijkstra, that uses heuristics to speed up the search and the Bellman–Ford algorithm [2] that works with negative edge weights. The Floyd–Warshall algorithm [5] can be used to solve the *all pairs shortest path problem*.

2.3 Landmark-based approximate algorithms

In many applications it is much more important to keep query execution time as short as possible even if it requires resorting to approximations. Most of the approximate methods are based on graph indexing and using this index to compute approximate results.

One of the most popular families of techniques that is simple and scalable at the same time, is based on distance approximation using a fixed set of selected nodes called *landmarks*. The true shortest path distances are calculated between every node in the graph and each landmark $u \in U$. Then these distances can be used to get *lower* and *upper* bound approximations of the shortest path distance between any two nodes s and t by applying triangle inequalities:

$$d(s,t) \leq d(s,u) + d(u,t) =: d_U \tag{2.2}$$

$$d(s,t) \geq |d(s,u) - d(u,t)| =: d_L \tag{2.3}$$

The true distance lies in the range $[d_L, d_U]$. The approximated distance can be taken as equal to any of these two values or computed as some value in between e.g. arithmetic mean. The previous work [13] indicates that upper-bound estimates give much better results than most other types.

2.3.1 Basic algorithm

The idea of the landmark-based shortest path approximation is to use a set U of landmark nodes. In this case the final approximation can be computed as the

minimum of upper bound approximations for each landmark:

$$d^U_{\text{approx}}(s,t) = \min_{u \in U}(d(s,u) + d(u,t)) \tag{2.4}$$

The Algorithm 2 based on equation (2.4) is a formal description of the basic landmark-based shortest distance approximation method as given in [14].

The described basic method returns only distances, but not the shortest paths themselves. This limitation can be addressed by storing instead of the distance value the pointer to the previous node in the shortest path tree for each node and landmark.

For example, in the situation depicted in Figure 2.1 we want to get the shortest path between v_1 and v_3. The basic algorithm will return the path

$$\pi_{v_1,v_3} = \pi_{v_1,u} + \pi_{u,v_3} = (v_1, v_4, u, v_5, v_3) \tag{2.5}$$

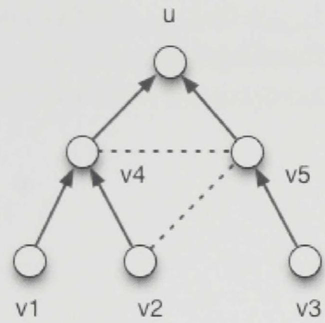


Figure 2.1: Shortest path tree for landmark u. Black arrows denote parent links, dotted lines are edges not in the landmark tree.

2.3.2 Landmark-LCA algorithm

The idea behind the Landmark-Basic algorithm is for each query (s,t) and landmark u to return the length of the path $|\pi_{s,u} + \pi_{u,t}|$. However, there may be the cases when $\pi_{s,u}$ and $\pi_{u,t}$ contain the same nodes (besides u).

For example, in Figure 2.1 for nodes v_1 and v_2 the Landmark-Basic method returns the path (v_1, v_4, u, v_4, v_2) of distance 4, but we can see that v_4 is traversed two times and therefore the result can be optimized by ignoring edges (v_4, u) and (u, v_4). In this case we get the path (v_1, v_4, v_2) of distance 2.

The node v_4 is a *common ancestor* of nodes v_1 and v_2 in the shortest path tree of u. There may be multiple such nodes, but the one that is encountered in both shortest paths from start and end nodes to a landmark first is called the *lowest common ancestor (LCA)*.

The Landmark-LCA algorithm (see Algorithm 3 from [14]) uses the presented idea to get more accurate results comparing to the Landmark-Basic method.

2.3.3 Landmark-BFS algorithm

Both the Landmark-Basic and Landmark-LCA algorithms use only the precomputed shortest path tree index and do not refer to the graph itself. The index includes only a limited subset of edges from the graph and therefore in practice there may be cases when the returned path can be further optimized by taking into account the information from the graph.

The idea of Landmark-BFS (see Algorithm 4 from [14]) is to run BFS algorithm on a limited subset of the nodes in the graph. This set is unique for each query (s, t) and consists of all the nodes in the shortest path trees between all landmarks and nodes s and t. This algorithm uses all edges from the original graph corresponding to the selected nodes and therefore can find shortcuts, which Landmark-LCA would not be able to identify.

Consider the situation from Figure 2.1 and shortest path query (v_1, v_3). In this case the Landmark-LCA algorithm cannot improve the result of the Landmark-Basic one. Landmark-BFS algorithm will run the BFS over the set of nodes v_1, v_3, v_4, v_5, u (nodes from the paths $\pi_{v_1,u}$ and $\pi_{v_3,u}$). There exists a shortcut edge (v_4, v_5), that is not included in the pruned landmark tree, but both nodes of it are in the selected set of nodes and therefore the returned path would be (v_1, v_4, v_5, v_3).

Algorithm 1 BREADTH-FIRST SEARCH

Require: Graph $G = (V, E)$

```
1: function RESTORE-PATH(x, p)
2:     Result ← ()                                   ▷ Empty path
3:     while x ≠ p[x] do
4:         Append x to Result
5:         x ← p[x]
6:     end while
7:     Append x to Result
8:     return Result
9: end function

10: function BFS(G, s, t)
11:     for v ∈ G.nodes() do                 ▷ Initialize previous node array
12:         p[v] ← nil
13:     end for
14:     Create a queue Q.
15:     Q.enqueue(s)
16:     p[s] ← s
17:     while Q is not empty do
18:         x ← Q.dequeue()
19:         if x = t then
20:             return RESTORE-PATH(t, p)
21:         end if
22:         for v ∈ G.adjacentNodes(x) do
23:             if p[v] = nil then                     ▷ Node is not visited
24:                 p[v] ← x                           ▷ Save previous node
25:                 Q.enqueue(v)
26:             end if
27:         end for
28:     end while
29:     return ()                                      ▷ No path exists
30: end function
```

Algorithm 2 LANDMARKS-BASIC, FROM [14]

Require: Graph $G = (V, E)$, number of landmarks k, d_u - array of the length $|V|$ of distances from each node to each $u \in U$.

```
1: function LANDMARKS-BASIC(s, t)
2:     d_approx ← min_{u∈U}(d_u[s] + d_u[t])
3:     return d_approx
4: end function
```

Algorithm 3 Landmark-LCA, from [14]

Require: Graph $G = (V, E)$, a landmark $u \in V$, a parent link $p_u[v]$ precomputed for each $v \in V$.

```
 1: function PATH-TO_u(s,π)
        Returns the path in the SPT p_u from the vertex s
        to the closest vertex q belonging to the path π
 2:     Result ← (s)                         ▷ Sequence of 1 element.
 3:     while s ∉ π do
 4:         s ← p_u[s]
 5:         Append s to Result
 6:     end while
 7:     return Result                        ▷ (s, p_u[s], p_u[p_u[s]], ..., q), q ∈ π
 8: end function
 9: function DISTANCE-LCA_u(s,t)
10:     π^(1) ← PATH-TO_u(s, (u) )
11:     π^(2) ← PATH-TO_u(t, π^(1))
12:     LCA ← Last element of π^(2)
13:     π^(3) ← PATH-TO_u(s, (LCA) )
14:     return |π^(2)| + |π^(3)|
15: end function
```

Algorithm 4 Landmarks-BFS, from [14]

Require: Graph $G = (V, E)$, a set of landmarks $U \subset V$, an SPT parent link $p_u[v]$ precomputed for each $u \in U, v \in V$.

```
 1: function LANDMARKS-BFS(s,t)
 2:     S ← ∅
 3:     for u ∈ U do
 4:         S ← S ∪ PATH-TO_u(s, (u))        ▷ (see Algorithm 3)
 5:         S ← S ∪ PATH-TO_u(t, (u))
 6:     end for
 7:     Let G[S] be the subgraph of G induced by S.
 8:     Apply BFS on G[S] to find a path π from s to t.
 9:     return |π|
10: end function
```

Chapter 3

Pruned Landmark-Based Algorithms

Traditional landmark-based methods rely on shortest path trees, where paths are stored from each landmark to all nodes in the graph. The improvements over well-known techniques, proposed in this work, are based on the idea that for each node it is enough to store paths to a limited set of the closest landmarks in order to get the approximation accuracy comparable to regular methods, with much lower memory requirements.

3.1 Pruned landmark trees

Define a *pruned landmark tree* (PLT) as a shortest path tree on a subset of nodes $V' \subset V$ in the graph G with a landmark node as the root.

There may be multiple pruning strategies. The method proposed by Vieira *et al.* [15] limits trees based on a *depth*, i.e. it ignores all nodes having the distance from the landmark that is larger than some fixed value. The drawbacks of this strategy are that nodes are inequally covered by landmarks and there may even exist nodes not connected to any landmarks which makes impossible to approximate distances between them and any other vertices.

We propose to apply restrictions to nodes instead of landmarks for pruned landmark trees computation. For each node x we limit the size of the associated landmarks set $L(x)$ with some fixed value of *number of landmarks per node* r: $\forall x \in V$, $|L(x)| \leq r$. Our preliminary experiments show that closer landmarks on average give better approximations than distant ones and therefore $L(x)$ consists of up to r closest landmarks for every node.

The described pruned landmark trees can be computed with the help of the modified BFS algorithm that we call PLT-Precompute (see Algorithm 5). Similarly to the regular BFS it is based on the iteration over a queue. This queue contains tuples (l, v, d), where l is a landmark, v is next node to be processed and d is the distance from l to v. The queue is initialized with the set $\{(l, l, 0) : l \in U\}$. This guarantees parallel graph exploration starting from all landmarks. The differ-

ence with the regular BFS is that each node can be visited by different landmarks up to r times. This is implemented by keeping track of the set of the reached landmarks for each node. No further traversal is allowed when a node has already been visited by r landmarks. The algorithm stops when the queue empty. We keep track of the parent pointers $p_l[x]$ in the shortest path tree for each node x and associated to it landmark l. According to this algorithm the set of the selected landmarks for a node x can be defined as $L(x) = \{l : p_l[x] \neq nil\}$.

The PLT-PRECOMPUTE algorithm selects $\min(k', r)$ landmarks for every node x, where k' is the number of landmarks in the same component as x (see Theorem A.1 in the appendix). The selected landmarks $L(x)$ are the closest ones among all possible (see Theorem A.2 in the appendix).

Consider an example graph depicted on Figure 3.1. Assume that nodes u_1, u_2 and u_3 are selected as landmarks. Figure 3.2 demonstrates shortest path trees for each of these landmarks obtained by applying the BFS algorithm. Obviously, there may be other shortest path trees obtained depending on the order in which the neighbors are returned for each node x when $G.adjacentNodes(x)$ method is called (see Algorithm 1). In this example each SPT contains all 10 nodes from the graph.

The algorithm is designed to take undirected graphs as an input. In the case of directed graphs we need to use two PLTs for each landmark rather than one: the first one holding previous nodes on the paths *from* landmark to nodes and the second one with paths *to* a landmark. For simplicity of description all following algorithms also operate undirected graphs.

3.2 Computation time and space complexity

The proposed way of computing pruned landmark trees requires visiting each node and each edge up to r times and therefore pruned trees can be built in $O(r(m+n))$ time, which is more efficient compared to $O(k(m+n))$ complexity of running BFS over the whole graph for all landmarks in the regular landmark-based methods.

There are two strategies for storing pruned landmark trees depending on whether the required output is just a distance or the shortest path itself. In the basic approach we just store distances from each node to the corresponding landmarks, which requires $O(rn)$ space.

If we are interested in computing shortest paths, we need to store the index of a previous node for each node in all associated pruned landmark trees. In this case it is needed to replace every distance with a pointer to a previous node in the shortest path tree, which takes the same $O(rn)$ space.

3.3 Basic method

As described in Section 2.2, the basic landmark-based approximation method is based on the simple triangle inequality upper bound approximation between nodes

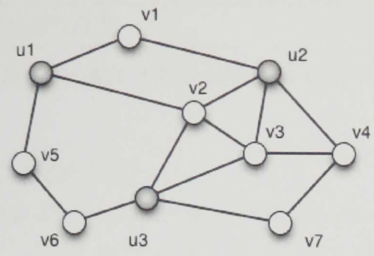
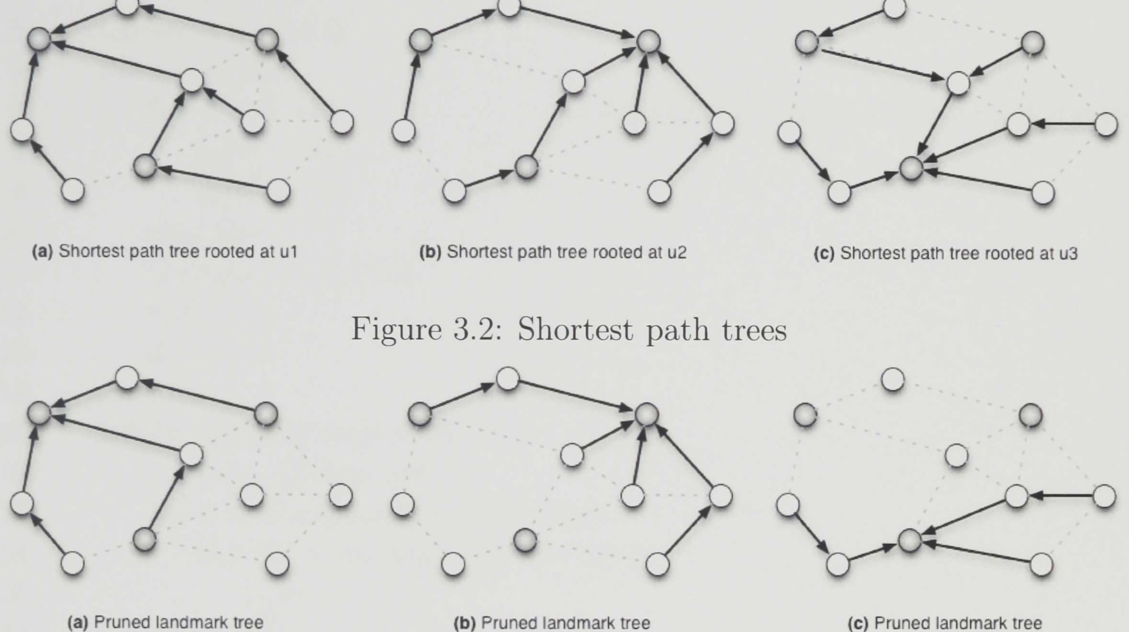


Figure 3.1: Example graph for PLT computation. Selected landmarks u_1, u_2, u_3 are highlighted



(a) Shortest path tree rooted at u1 (b) Shortest path tree rooted at u2 (c) Shortest path tree rooted at u3

Figure 3.2: Shortest path trees

(a) Pruned landmark tree rooted at u1 (b) Pruned landmark tree rooted at u2 (c) Pruned landmark tree rooted at u3

Figure 3.3: Computed pruned landmark trees

s and t by using a landmark $u \in U$. The same algorithm cannot be directly applied to pruned landmark trees as it is not guaranteed that all pairs of start and end nodes (s and t) have common landmarks, i.e. belong to the same landmark shortest path trees. As a solution to this problem we propose to use a pair of landmarks $u \in L(s)$ and $v \in L(t)$ in the shortest path distance approximation. Therefore, we need to include the distance between these landmarks $d_{inter}(u, v)$ into the equation:

$$d^{u,v}_{\text{approx}}(s,t) = d(s,u) + d_{inter}(u,v) + d(v,t) \tag{3.1}$$

To get the best approximation we iterate over all pairs of landmarks (u, v) and select the minimum resulting distance (Algorithm 6). If there are common landmarks between s and t ($L(s) \cap L(t) \neq \varnothing$) on the iterations where $u \equiv v$ this method returns the same result as the LANDMARK-BASIC algorithm.

Consider the graph and pruned landmark trees from Figures 3.1 and 3.3. Sup-

Algorithm 5 PLT-Precompute

Require: Graph $G = (V, E)$, a set of landmarks $U \subset V$, number of landmarks per node r.

```
 1: procedure PLT-Precompute
    Computes p_l[x] that gives the previous node for node x in the pruned landmark
    tree rooted at l and d_l[x] that gives the distance from landmark l to a node x
 2:     for x ∈ V do                     ▷ Initialize number of landmarks per node
 3:         r[x] ← 0
 4:     end for
 5:     Create a queue Q.
 6:     for l ∈ U do
 7:         for x ∈ V do                 ▷ Initialize previous nodes and distances
 8:             p_l[x] ← nil
 9:             d_l[x] ← ∞
10:         end for
11:         Q.enqueue((l, l, 0)).
12:         p_l[l] ← l
13:         d_l[l] ← 0
14:     end for
15:     while Q is not empty do
16:         l, u, d ← Q.dequeue()
17:         for x ∈ G.adjacentNodes(u) do
18:             if p_l[x] = nil and r[x] < r then
19:                 p_l[x] ← u
20:                 d_l[x] ← d + 1
21:                 r[x] ← r[x] + 1
22:                 Q.enqueue((l, x, d + 1)).
23:             end if
24:         end for
25:     end while
26: end procedure
```

pose that we estimate the distance between v_5 and v_4. When we use landmarks u_1 and u_2 and the shortest path between them is computed as (u_1, v_1, u_2) then the described PLT-Basic algorithm will consider the path $(v_5, u_1, v_1, u_2, v_4)$. In this case the two nodes are both present in the landmark tree rooted at u_3 and PLT-Basic algorithm will also find the path $(v_5, v_6, u_3, v_3, v_4)$ also of length 4.

3.4 Interlandmark distance approximation

The straightforward method to compute shortest paths between all pairs of landmarks is to run BFS from each landmark and save distances to all other ones. This

Algorithm 6 PLT-BASIC

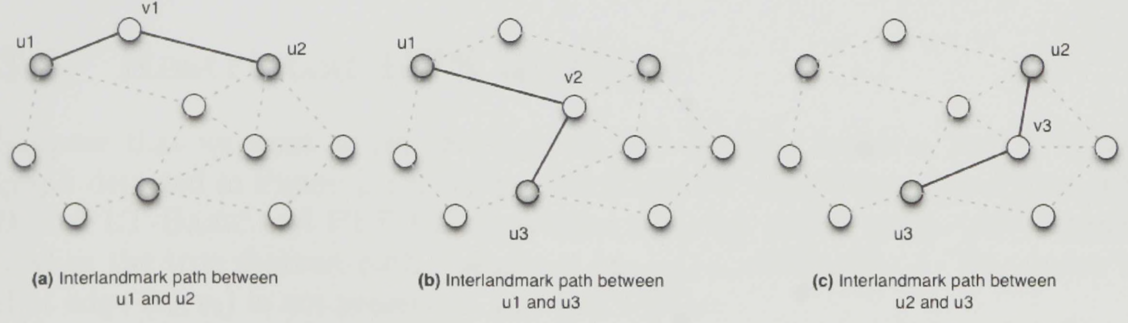
Require: $d(x, l)$: a function returning distance from node x to landmark l, $d_{inter}(u, v)$: a function returning distance between landmarks, $L(x)$: a set of the associated landmarks for a node x.

```
1: function PLT-BASIC(s, t)
2:     d_min ← ∞
3:     for u ∈ L(s) do
4:         for v ∈ L(t) do
5:             d ← d(s, u) + d_inter(u, v) + d(v, t)
6:             d_min ← min(d_min, d)
7:         end for
8:     end for
9:     return d_min
10: end function
```

procedure requires $O(k(m+n))$ time complexity. The linear time dependency on k prevents this algorithm of using the number of landmarks significantly larger than in other regular landmark-based methods and therefore reduces its benefits.

The proposed way to tackle this problem is to calculate approximations of interlandmark shortest path distances. This can be achieved from the data already collected by the PLT-PRECOMPUTE algorithm. The idea is to find a *witness node* $w[u, v]$ for each pair of landmarks $u \in U$ and $v \in U$ such that $w[u, v]$ is present in the pruned landmark trees for both u and v and the approximation of the distance between the landmarks through this node $(d_u[w[u, v]] + d_v[w[u, v]])$ is minimized. The implementation is provided in the CALCULATE-WITNESS-NODES procedure in Algorithm 7. When this procedure finishes the approximated shortest paths between landmarks can be obtained by calling the function PATH-BETWEEN.

The time complexity of the CALCULATE-WITNESS-NODES procedure is $O(nr^2)$, which is much faster than the naïve approach of running the BFS k times for $r \ll k$. Storing distances between all pairs of landmarks requires $O(k^2)$ space.



(a) Interlandmark path between u1 and u2

(b) Interlandmark path between u1 and u3

(c) Interlandmark path between u2 and u3

Figure 3.4: Interlandmark paths

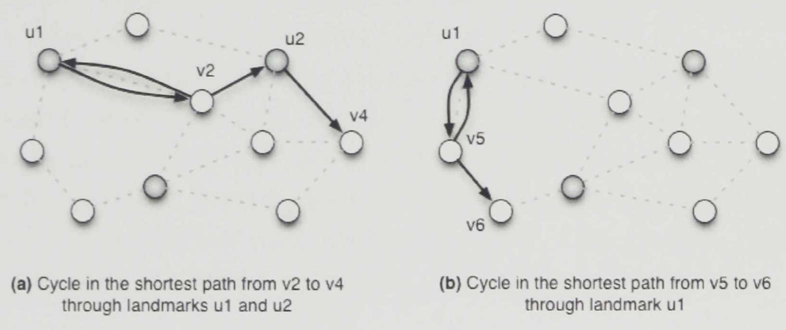
Figure 3.4 depicts an example of interlandmark path for graph from Figure 3.1. In this case the possible values for witness nodes can be $w[u_1, u_2] = v_1$, $w[u_1, u_3] =$

u_3 and $w[u_2, u_3] = v_3$.

3.5 Cycle elimination

Consider the graph and pruned landmark trees from Figures 3.1 and 3.3. If we estimate the distance between v_2 and v_4 through u_1 and u_2 with interlandmark path (u_1, v_2, u_2) then PLT-BASIC method will return the path $(v_2, u_1, v_2, u_2, v_4)$ of distance 4 that contains cycle (v_2, u_1, v_2) (see Figure 3.5a). When it is removed the resulting path (v_2, u_2, v_4) will have distance 2.

If we estimate the distance between v_5 and v_6 we can see that both of these nodes are in the shortest path tree of a landmark u_1. The path returned by the basic method through this single landmark is (v_5, u_1, v_5, v_6) (with distance 3) which, after cycle elimination, turns into (v_5, v_6) of distance 1 (see Figure 3.5b).



(a) Cycle in the shortest path from v2 to v4 through landmarks u1 and u2

(b) Cycle in the shortest path from v5 to v6 through landmark u1

Figure 3.5: Cycle elimination examples.

The PLT-CE algorithm described in Algorithm 8 implements the cycle elimination technique to improve the results of the PLT-BASIC. It uses a stack and a set data structures to find and remove cycles in a path. It can get improved results in both cases, when the start and end nodes have a common landmark (belong to the same landmark shortest path tree) or the distance is approximated via a pair of different landmarks.

3.6 Restricted BFS method

Suppose that we want to get the shortest path between nodes u_1 and v_3 in the graph depicted in Figure 3.1 using pruned landmark trees depicted in Figure 3.3. Both PLT-BASIC and PLT-CE algorithms can only return paths with distance 3 when the true shortest path distance is (u_1, v_2, v_3) of distance 2. The reason is that edge (v_2, v_3) is not present in any used PLTs.

Work by Tretyakov *et al.* [14] proposes a method for running BFS algorithm for the shortest distance approximation on the limited number of nodes in the graph (LANDMARK-BFS). BFS analyzes all nodes on the shortest paths from start and end nodes to all landmarks. Unlike the basic landmark-based method it operates

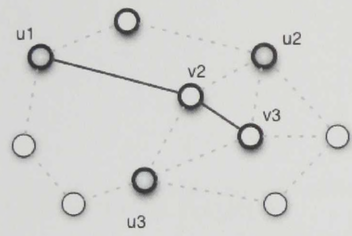


Figure 3.6: Set of nodes analyzed by PLT-BFS algorithm and the resulting shortest path

all paths to landmarks at the same time and therefore is able to find paths that cannot be observed when all landmarks are considered independently.

This method also makes use of *shortcuts* – edges that are present in the graph but are not present in landmark trees and therefore requires the graph itself. Another benefit of running BFS is that it always returns a shortest path that does not contain cycles due to the nature of this algorithm.

The PLT-BFS algorithm is the adapted version of LANDMARK-BFS that operates on pruned landmark trees. In the example graph from Figure 3.1 this algorithm will perform a BFS search from u_1 to v_3 over the nodes $\{u_1, u_2, u_3, v_1, v_2, v_3\}$ (see Figure 3.6). This set is composed of the nodes from the shortest paths between u_1 and v_3 and corresponding landmarks $\{\pi_{u_1,x} | x \in L(u_1)\} \cup \{\pi_{v_3,x} | x \in L(v_3)\}$ and all nodes on the interlandmark paths $\{\pi_{u,v} | u \in L(u_1), v \in L(v_3)\}$.

Algorithm 7 CALCULATE-WITNESS-NODES

Require: Graph $G = (V, E)$, an SPT parent link $p_u[x]$ precomputed for each $u \in L(x), x \in V$.

```
1: procedure CALCULATE-WITNESS-NODES
   Computes witness nodes w[u,v] for all pairs of landmarks (u,v) for u ∈ U and v ∈ U
2:     for x ∈ V do
3:         for u ∈ L(x) do
4:             for v ∈ L(x) do
5:                 if w[u,v] = nil or (d_u[x] + d_v[x] < d_u[w[u,v]] + d_v[w[u,v]]) then
6:                     w[u,v] ← x
7:                 end if
8:             end for
9:         end for
10:    end for
11: end procedure
12: function PATH-TO-LANDMARK(x, u)
       Returns the path in the SPT p_u from the node x to the landmark u
13:    Result ← ()                                        ▷ Empty path.
14:    while x ≠ p_u[x] do
15:        Append x to Result
16:        x ← p_u[x]
17:    end while
18:    Append x to Result
19:    return Result
20: end function
21: function PATH-BETWEEN(u,v)
       Returns the path between landmarks u and v
22:    π ← PATH-TO-LANDMARK(w[u,v], u) + REVERSED(PATH-TO-LANDMARK(w[u,v], v))
23:    return π
24: end function
```

Algorithm 8 PLT-CE

Require: Graph $G = (V, E)$, a landmark $u \in V$, an SPT parent link $p_u[x]$ pre-computed for each $u \in L(x), x \in V$.

```
1: function ELIMINATE-CYCLES(π)
       Returns a subpath of π with eliminated cycles
2:     S ← ∅
3:     T ← Empty stack
4:     for x ∈ π do
5:         if x ∈ S then
6:             while x ≠ T.top() do
7:                 v ← T.pop()
8:                 Remove v from S.
9:             end while
10:        else
11:            Add x to S
12:            T.push(x)
13:        end if
14:    end for
15:    return Path from T
16: end function
17: function PLT-CE(s,t)
18:    d_min ← ∞
19:    for u ∈ L(s) do
20:        for v ∈ L(t) do
21:            π ← PATH-TO-LANDMARK(s, u) + PATH-BETWEEN(u, v)
22: + REVERSED(PATH-TO-LANDMARK(t, v))                ▷ Path concatenation.
23:            d ← |ELIMINATE-CYCLES(π)|
24:            d_min ← min(d_min, d)
25:        end for
26:    end for
27:    return d_min
28: end function
```

Algorithm 9 PLT-BFS

Require: Graph $G = (V, E)$, a set of landmarks $U \subset V$, an SPT parent link $p_u[x]$ precomputed for each $u \in L(x), x \in V$.

1: **function** PLT-BFS(s,t)
2: $S \leftarrow \varnothing$
3: **for** $u \in L(s) \cup L(t)$ **do**
4: $S \leftarrow S \cup$ PATH-TO-LANDMARK(s, u) ▷ (see Algorithm 7)
5: $S \leftarrow S \cup$ PATH-TO-LANDMARK(t, u)
6: **end for**
7: **for** $u \in L(s)$ **do**
8: **for** $v \in L(t)$ **do**
9: $S \leftarrow S \cup$ PATH-BETWEEN(u, v) ▷ (see Algorithm 7)
10: **end for**
11: **end for**
12: Let $G[S]$ be the subgraph of G induced by S.
13: Apply BFS on $G[S]$ to find a path π from s to t.
14: **return** $|\pi|$
15: **end function**

Chapter 4

Experimental Evaluation

The common way to evaluate approximate shortest path algorithms is to check their performance on sample graphs. In this case the important outcome parameters are an average accuracy (or the opposite value – approximation error), query time and precomputation (index building) time.

4.1 Datasets

The described algorithms were tested on four real-world social network graphs that are diverse in terms of number of nodes and edges.

- **DBLP.** The DBLP dataset contains bibliographic information of computer science publications [10]. Every node represents an author, every edge models a collaboration between authors. Edge is present if two authors have at least one common publication. The dataset is obtained on May 15, 2010.
- **Orkut.** Orkut is a social networking website that is owned and operated by Google. It this dataset each user is a node and connections between them is an edge. The snapshot of the Orkut network was published by Mislove *et al.* in 2007 [11].
- **Twitter.** Twitter is a microblogging platform where each user can follow other users. By its nature the formed social network is directed, but for the following experiments we ignore edge direction by keeping a connection if there is an edge in any direction. A snapshot of the Twitter network was published by Kwak *et al.* in 2010 [8].
- **Skype.** The Skype social graph contains users of the Skype peer-to-peer communication network. Each user is represented as a graph node. There is an edge between two nodes if they have each other in their contact lists. We operate the dataset obtained in November 2011.

The properties of the used graphs are summarized in Table 4.1. The table shows the number of vertices $|V|$, number of edges $|E|$, average distance between

Dataset	$\|V\|$	$\|E\|$	$\overline{d}$	Δ	$\|S\|/\|V\|$	Disk Usage
DBLP	770K	2.6M	6.3	25	85%	27M
Orkut	3.1M	117M	5.7	10	100%	918M
Twitter	41.7M	1.2B	4.2	25	100%	9.3G
Skype	539M	2.2B	6.7	59	95%	21G

Table 4.1: Datasets

vertices $\overline{d}$ (computed on a sample vertex pairs), approximate diameter Δ, the fraction of vertices in the largest connected component $|S|/|V|$ and disk usage for storing graph data. For the graph representation there was used a sorted list of edges with additional offset index which allowed to access the list of adjacent nodes in constant time.

All the graphs are undirected. DBLP, Orkut and Twitter datasets are the same as were used in [14], but the Skype graph is a more recent snapshot of the social network with a slightly different edge filtering approach than in the mentioned work. The preliminary data cleaning done on Skype social network is excluding all users not connected to anyone else.

4.2 Landmark selection

Finding a good set of landmarks is very important for the performance of the landmark-based approximation algorithms [6]. As the proposed methods are focused on using larger number of landmarks than the previous techniques it becomes very important to choose scalable selection strategies. We use two strategies:

- **Random Selection**. This is a basic and computationally efficient strategy where all landmarks are selected randomly. We use the same nodes in the experiments with equal landmark set size in order to make results more comparable.

- **Highest Degree Selection**. A landmark set of size k is selected as the top k nodes with the highest degrees. The idea of this method is that a node's degree can be considered as an approximation of its *centrality* i.e. the larger neighborhood a node has, the more shortest paths go through it.

One or both of these strategies have been used in many previous works that involve landmark-based methods [6, 13, 14, 15, 17].

4.3 Experimental setup

In each experiment we randomly choose 500 pairs of nodes (s, t), called *queries*. True distances from s to t are calculated by running the BFS algorithm. We apply

the proposed distance approximation algorithms to these queries and measure their average values of *approximation error* and *query execution time*.

All experiments were run under Scientific Linux release 6.3 operating system, which is based on Red Hat Enterprise distribution, on a server with 8 Intel Xeon E7-2860 processors and 1024GB RAM. Only a small part of the computational resources was used in all experiments.

The described methods were implemented in Java. Graphs and intermediate data were stored on disk and accessed through memory mapping, which helped to reduce the memory overhead of operating thousands of objects in the Java Virtual Machine heap.

4.4 Results

4.4.1 Approximation error

We measure the accuracy in calculating shortest paths between pairs of nodes. For each method, dataset, number of landmarks, and number of landmarks per node we report the *approximation error*. It is computed as

$$e_{approx} = \frac{d_{approx} - d}{d}, \tag{4.1}$$

where d is the actual distance and d_{approx} is the approximation.

Figures 4.1, 4.2, 4.3 and 4.4 show approximation error for DBLP, Orkut, Twitter and Skype graphs correspondingly. The error values are present for different landmark selection strategies (rows), algorithms (columns), numbers of landmarks per node (bar colors) and number of landmarks (x-axis). The dashed black line is the baseline for each of the figures, i.e. the performance of a method with 100 non-pruned landmark trees from the work by Tretyakov *et al.* [14]. We use LANDMARK-BASIC as the baseline for PLT-BASIC, LANDMARK-LCA as the baseline for PLT-CE and LANDMARK-BFS as the baseline for PLT-BFS.

Tables 4.2, 4.3, 4.4 and 4.5 show results for the largest measured number of landmarks per node computed for each graph correspondingly.

Landmark selection strategy is a very significant factor of approximation quality, especially for the PLT-BASIC and PLT-CE algorithms. But for the PLT-BFS method randomly selected landmarks provide comparable accuracy to the highest degree landmarks and can even outperform them, as in the case for the Twitter graph.

Higher numbers of landmarks per node lead to consistent reduction of the approximation error. In most of the cases using larger number of landmarks tends to increase the total accuracy, but we can observe the opposite effect for random landmark selection in Orkut and Twitter graphs. Moreover, for DBLP and Orkut graphs with $r = 5$ and $k = 10000$ there were cases when all three algorithms were wrongly returning infinity distances (the reason why Figures 4.1 and 4.2 are

missing approximation error for this parameter values). The explanation is that in this case the approximate interlandmark distance is equal to infinity for all start and end node's landmark pairs. Further investigation showed that even using exact interlandmark distance for combinations with small values of r and large values of r reduces the total accuracy.

The obtained results demonstrate that the accuracy in all three introduced algorithms highly depends on the internal properties of graphs themselves. While PLT-BFS method can return exact values in almost all cases in the DBLP graph (approximation error is less than 0.01), the lowest obtained error for the Skype graph is still as high as 0.09.

The comparison with regular landmark-based algorithms confirms the idea that our methods can achieve similar accuracy with much less memory usage. For example, in the Skype graph with highest degree landmark selection strategy, 5 landmarks/node and 10000 landmarks we achieve about the same approximate error as regular landmark-based methods that use full shortest path trees for 100 landmarks with about 8 times smaller memory requirements (see Table 4.8).

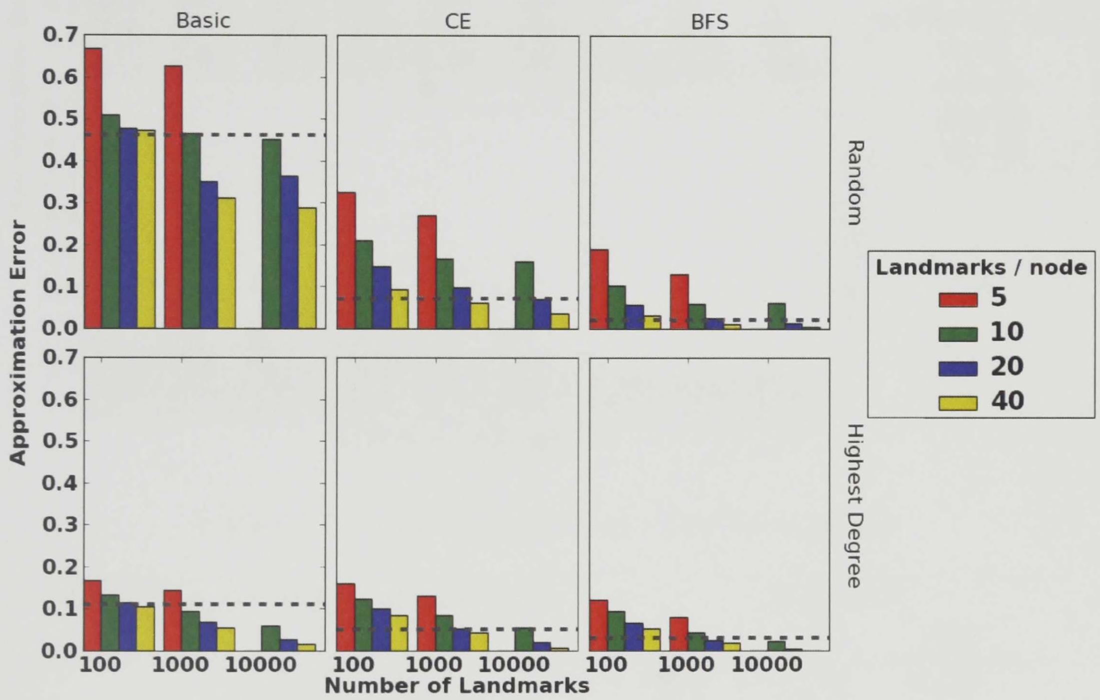


Figure 4.1: Approximation error for DBLP graph

4.4.2 Query execution time

The second important measurement parameter that characterizes and evaluates the proposed distance estimation methods is query execution time.

Selection strategy	Landmarks	Method		
		Basic	SC	BFS
Random	100	0.473	0.094	0.032
	1000	0.313	0.062	0.011
	10000	0.290	0.035	0.003
Highest Degree	100	0.105	0.086	0.054
	1000	0.055	0.044	0.019
	10000	0.017	0.007	0.0005

Table 4.2: Approximation error for 40 landmarks/node in DBLP graph

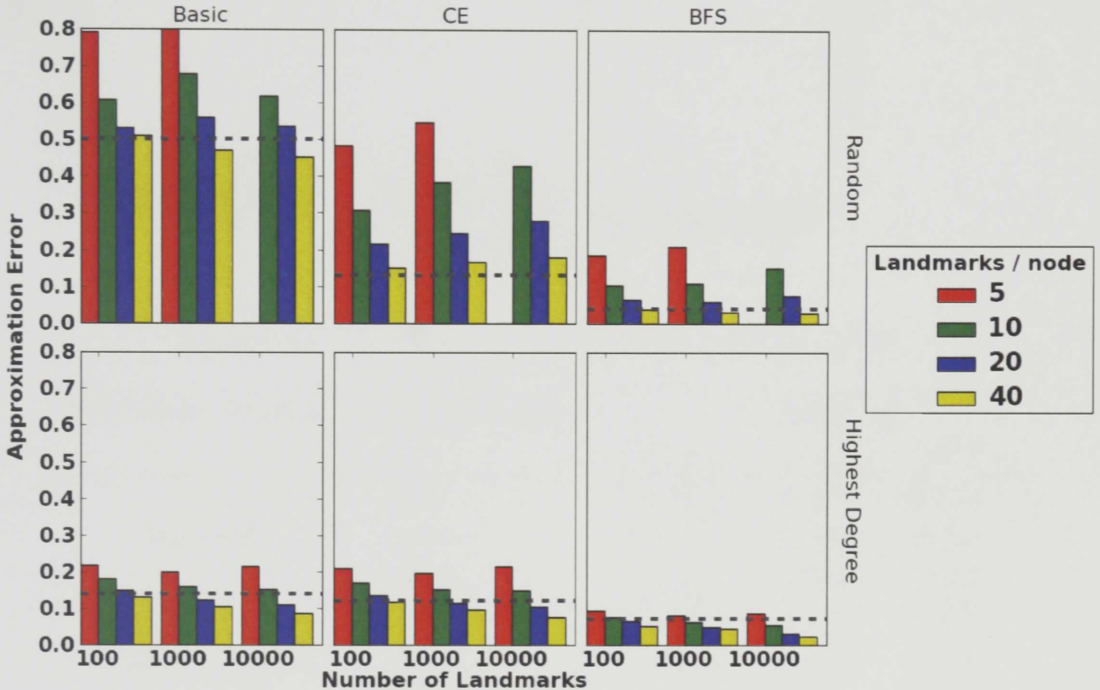


Figure 4.2: Approximation error for Orkut graph

Selection strategy	Landmarks	Method		
		Basic	SC	BFS
Random	100	0.511	0.151	0.038
	1000	0.471	0.168	0.031
	10000	0.454	0.181	0.029
Highest Degree	100	0.131	0.118	0.053
	1000	0.105	0.096	0.044
	10000	0.087	0.077	0.024

Table 4.3: Approximation error for 40 landmarks/node in Orkut graph

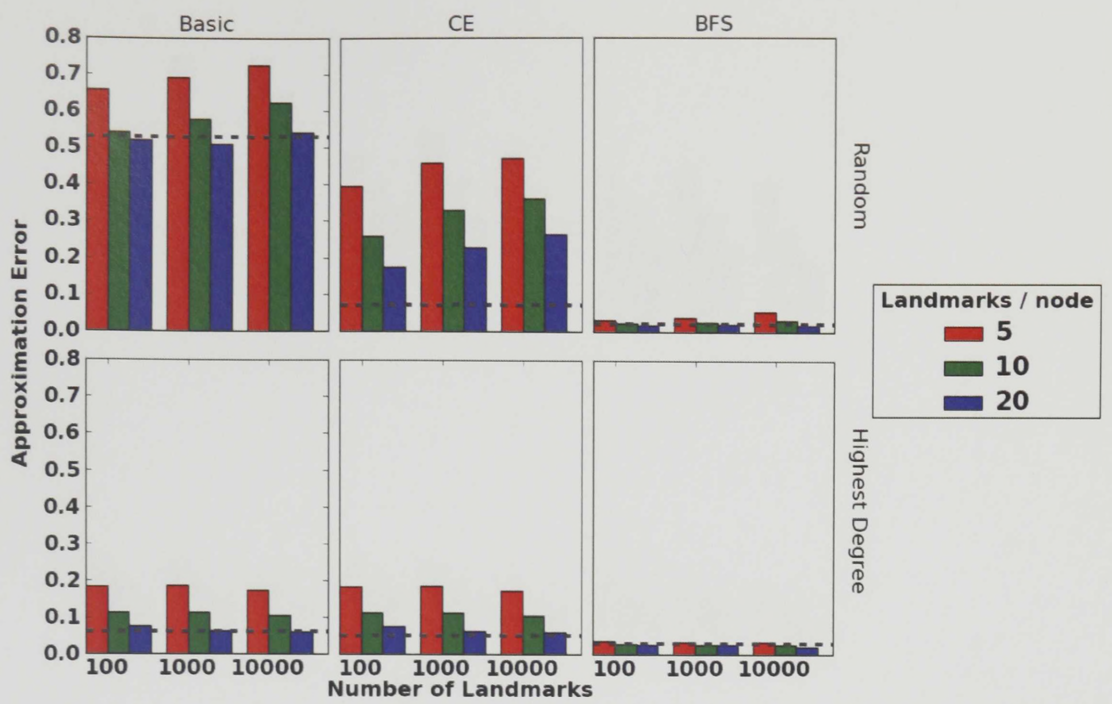


Figure 4.3: Approximation error for Twitter graph

Selection strategy	**Landmarks**	**Method**		
		Basic	SC	BFS
Random	100	0.521	0.175	0.018
	1000	0.511	0.228	0.020
	10000	0.543	0.264	0.018
Highest Degree	100	0.076	0.075	0.026
	1000	0.063	0.063	0.025
	10000	0.061	0.060	0.022

Table 4.4: Approximation error for 20 landmarks/node in Twitter graph

Selection strategy	**Landmarks**	**Method**		
		Basic	SC	BFS
Random	100	0.539	0.332	0.249
	1000	0.466	0.319	0.219
	10000	0.526	0.301	0.181
Highest Degree	100	0.162	0.158	0.134
	1000	0.154	0.148	0.114
	10000	0.134	0.124	0.091

Table 4.5: Approximation error for 20 landmarks/node in Skype graph

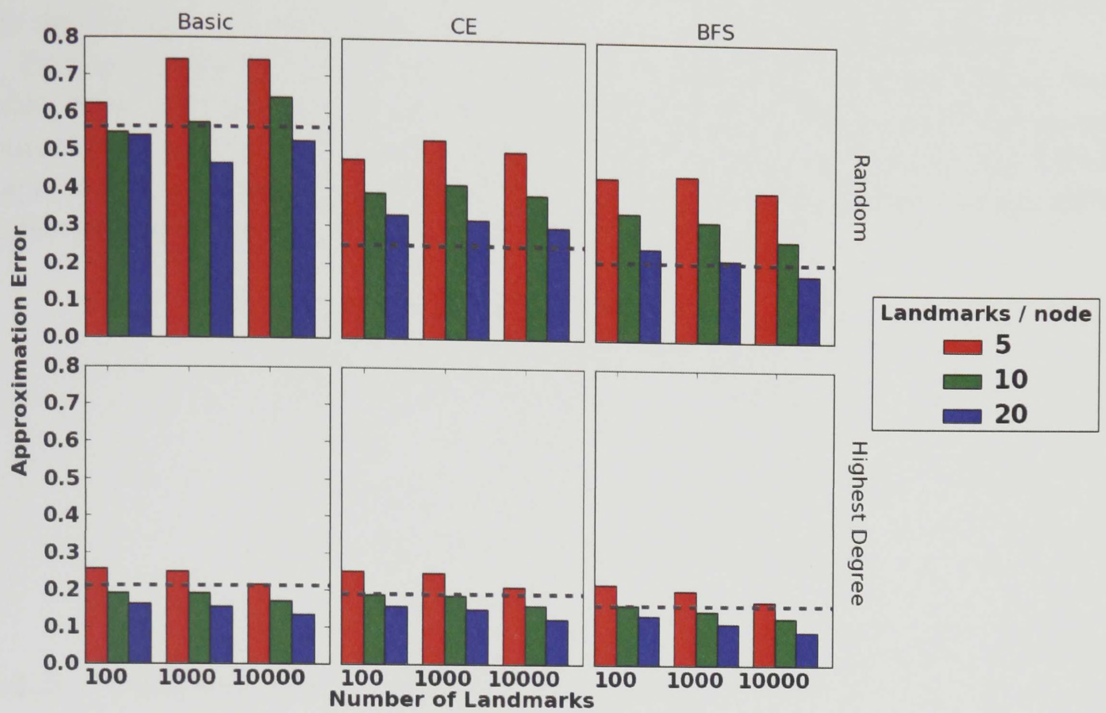


Figure 4.4: Approximation error for Skype graph

Query time was computed as the average value among 500 random queries in each graph. The total measured time excludes loading index into the main memory, but as our implementation uses the **mmap** Linux operating system feature, which does not guarantee that all the data is immediately loaded in RAM, a part of the measured time may also include time for loading parts of the index file.

Figures 4.5, 4.6, 4.7 and 4.8 show average query execution time for DBLP, Orkut, Twitter and Skype graphs correspondingly. The query time has quadratic dependency on the number of landmarks per node as is expected according to all algorithm descriptions. The number of landmarks and their selection strategy do not have a significant influence on this value.

The most influential factors for query time are algorithm selection and graph properties. The average query time of PLT-Basic and PLT-CE methods never exceeds 9 milliseconds for 20 landmarks/node and is even less than 1 millisecond for 5 landmarks/node in most of the cases. Unlike these two methods, the performance of the PLT-BFS highly depends on the dataset and the landmark selection strategy. For example, with 20 landmarks/node and the highest degree strategy the results vary from 9 milliseconds on the DBLP graph to 4.2 seconds on the Twitter graph.

An interesting observation here is that although the analyzed Skype graph has about 13 times more nodes and 1.8 times more edges than Twitter, the latter has longer average query times for the PLT-BFS algorithm. This can be explained by the fact that the BFS procedure in the algorithm is executed over the sets of

nodes of comparable sizes (average path distance is comparable in these graphs), but the Twitter subgraph that consists of these nodes is much more dense.

By the comparison of average query times with BFS times for each graph from Table 4.6 we can make a conclusion that the most benefit of using the presented approximate methods can be obtained on larger graphs. While for the DBLP graph the best achieved query time is about 300 times faster than average BFS, for Skype graph this multiplier is larger than a million.

Dataset	Average BFS	Full BFS
DBLP	156 ms	343 ms
Orkut	4.4 sec	25.4 sec
Twitter	1.3 min	11 min
Skype	62 min	76 min

Table 4.6: Average and full BFS times

4.4.3 Preprocessing time

Our experiments showed that the preprocessing time almost does not depend on the number of landmarks and their selection strategy. Table 4.7 contains time values obtained during the pruned landmark tree computation for different values of number of landmarks per node in each dataset. The data was collected for 1000 highest degree landmarks. Figure 4.9 is the visual representation of the first two rows of Table 4.7 with a common x-axis. Both graphs have similar growth pattern of the preprocessing time depending on the number of landmarks per node.

The pruned landmark tree index building time heavily depends on the size of the graph. For example, for 20 landmarks/node it ranges from about 21 seconds in DBLP to almost 45 hours in Skype. The quadratic dependency of the preprocessing time on number of landmarks per node prevents increasing this parameter for very large graphs, although it is highly important for good distance approximation accuracy.

Graph	Landmarks / Node			
	5	10	20	40
DBLP	3.6 sec	8.6 sec	21.1 sec	67.7 sec
Orkut	87 sec	207 sec	463 sec	1932 sec
Twitter	48 min	105 min	247 min	-
Skype	4.4 h	18.6 h	44.9 h	-

Table 4.7: Preprocessing time for 1000 landmarks with highest degree selection strategy

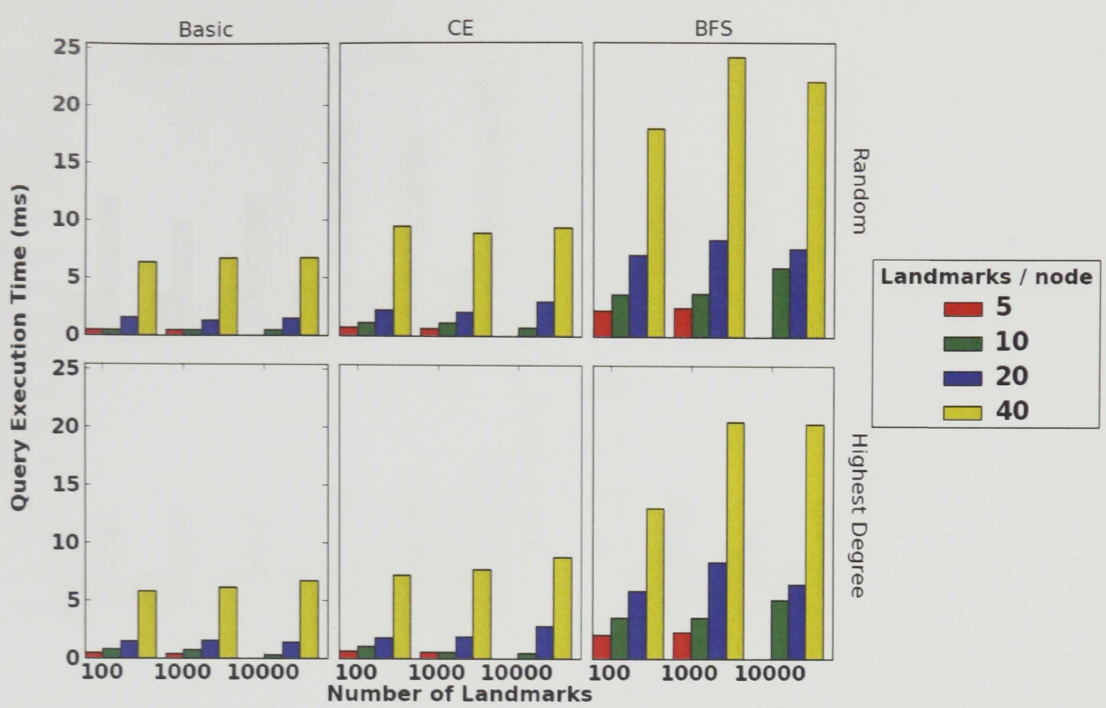


Figure 4.5: Average query time for DBLP graph

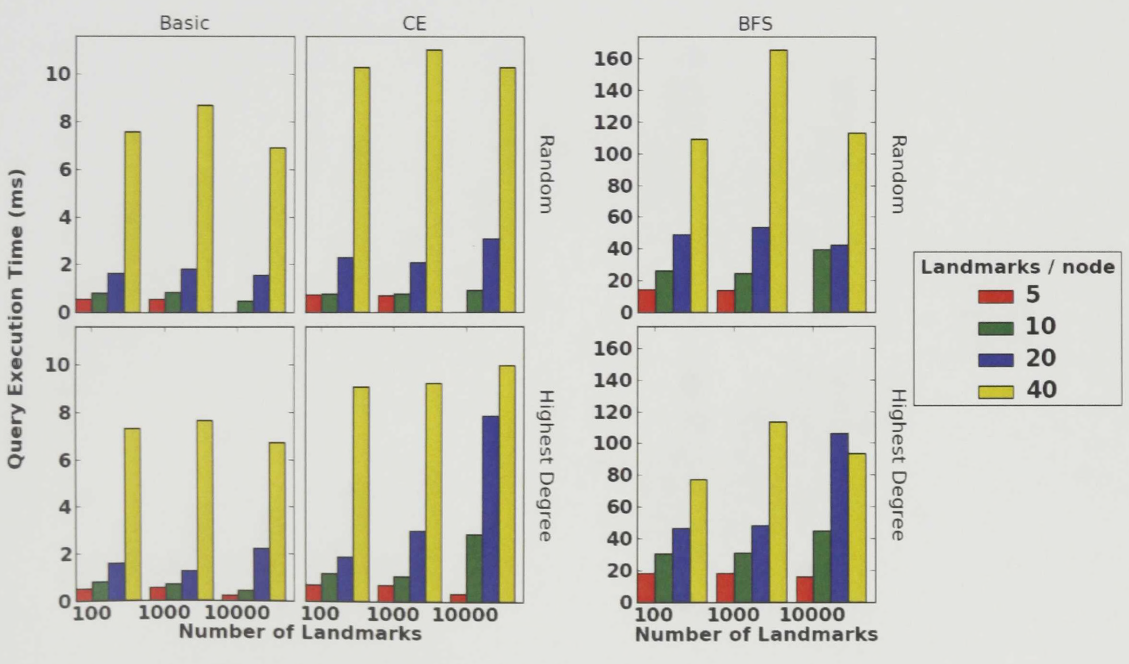


Figure 4.6: Average query time for Orkut graph

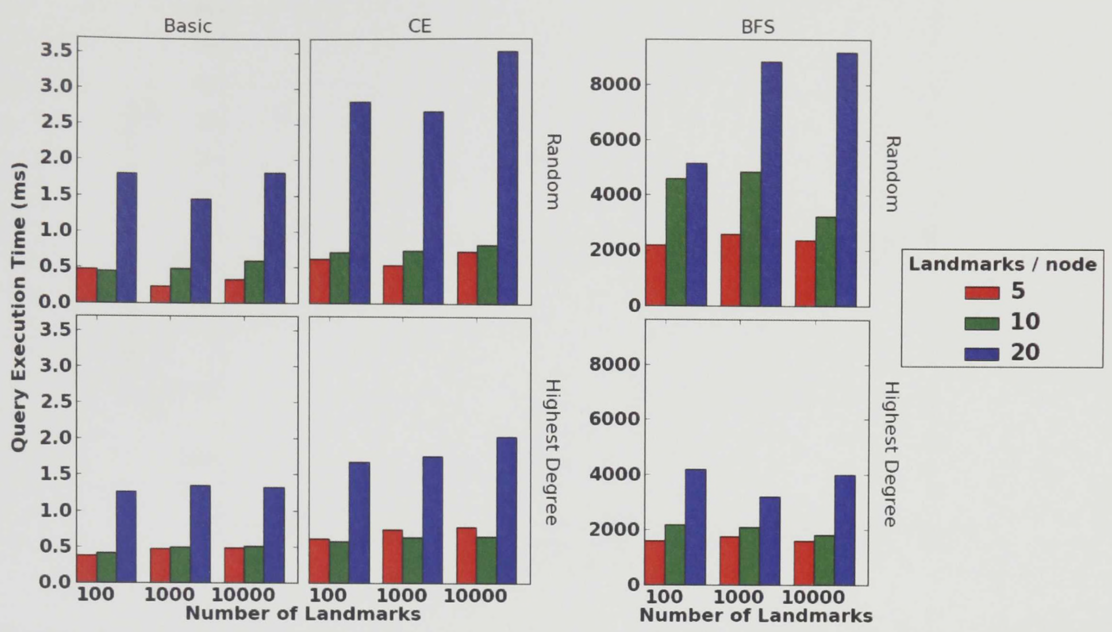


Figure 4.7: Average query time for Twitter graph

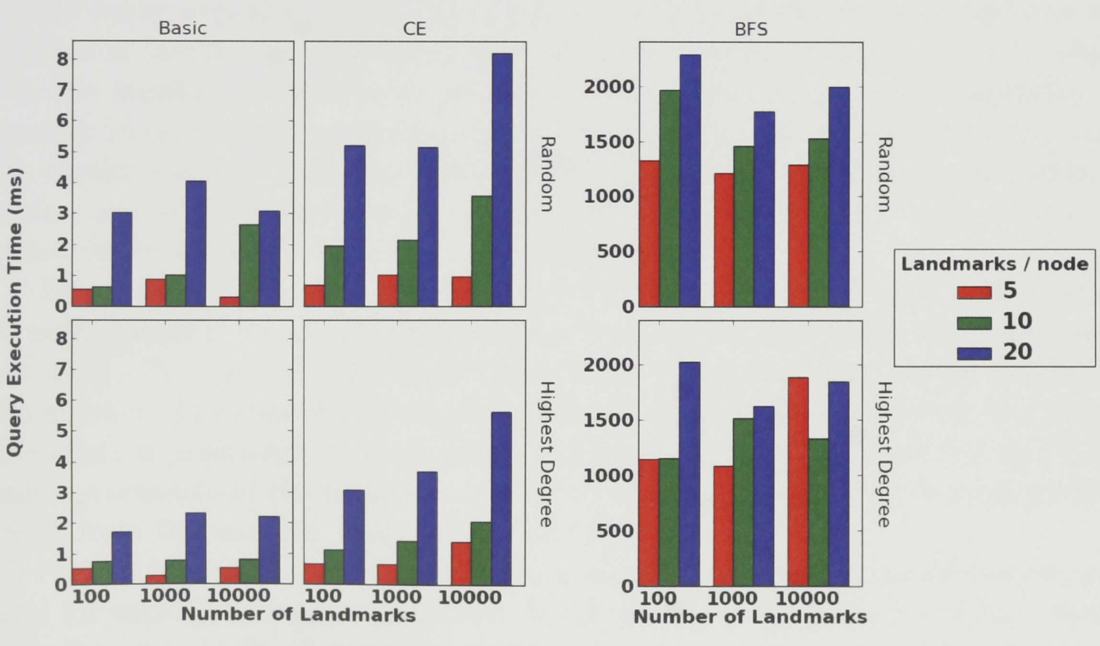


Figure 4.8: Average query time for Skype graph

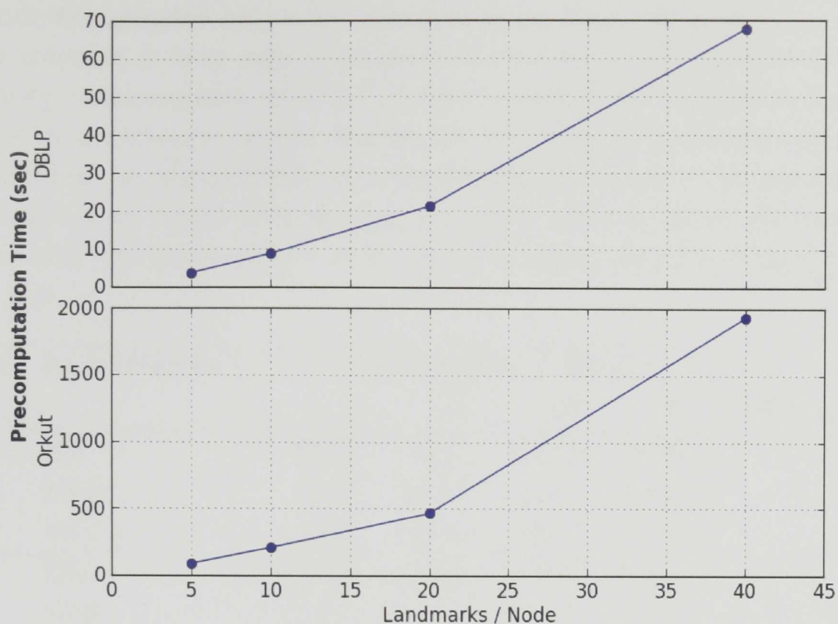


Figure 4.9: Preprocessing time for DBLP and Orkut (1000 landmarks with highest degree selection strategy)

4.4.4 Memory usage

One of the main benefits of the proposed pruned landmark tree structure compared to regular methods is the fact that the amount of the required memory does not have a linear dependency on $n \times k$, where n is the number of nodes and k is the number of landmarks. Instead of that the used memory is $\Theta(n \times r + k^2)$, where r is the number of landmarks per node. The consequence is the possibility to operate larger sets of landmarks and achieve accuracy comparable or better than the regular methods with much less memory consumption. While the second term of the sum is quite significant in small graphs, in graphs with large number of nodes the first term is the clearly the dominant one.

The described property can be observed in Table 4.8 that shows the amount of used memory for all graphs, different numbers of landmarks and landmarks per node. The sizes of DBLP and Orkut index structures significantly depend on the number of landmarks for small landmarks/node values. For Orkut and Skype this effect is practically unnoticeable. The last column of the Table 4.8 shows the baseline scenario of the total size of 100 full landmark shortest path trees for each graph from the work by Tretyakov *et al.* [14].

Pruned landmark trees are stored in a way that a pointer to a previous node is used for each (landmark, node) pair. In all graphs the nodes are assigned values from 0 to $n - 1$. In all our experiments we use the same pointer size of 4 bytes (32 bits). This leaves a room for a memory usage improvement if we use the smallest pointer size possible. For example, in DBLP the number of landmarks does not exceed 1048576 (2^{20}) and therefore it is enough to use 20 bits for each of

the previous node pointers. The drawback of this improvement is an additional source code complexity and a bit manipulation time overhead.

If our interest is only approximation of shortest path distances, we can reduce the memory consumption of PLT-Basic method by replacing a previous node pointer with a distance to the landmark for every (landmark, node) pair. The same can idea be applied for interlandmark distances. As diameters of social graphs tend to be small it is enough to use 1 byte to store all values in a range from 0 to 255. In this case we will obtain 4 times smaller values than presented in Table 4.8.

Graph	**Landmarks**	**Landmarks / Node**				**Baseline**
		5	10	20	40	**(100 landmark SPTs)**
DBLP	100	30M	59M	117M	231M	300M
	1000	34M	63M	121M	235M	
	10000	411M	441M	499M	613M	
Orkut	100	118M	235M	469M	938M	1.2G
	1000	122M	239M	473M	942M	
	10000	499M	616M	851M	1.3G	
Twitter	100	1.6G	3.2G	6.3G	-	16G
	1000	1.6G	3.2G	6.3G	-	
	10000	2.0G	3.5G	6.6G	-	
Skype	100	21G	41G	81G	-	170G
	1000	21G	41G	81G	-	
	10000	21G	41G	81G	-	

Table 4.8: Total PLT index memory usage

4.5 Approximation results for different distances

The approximation error results presented in Section 4.4.1 are based on the shortest path algorithms evaluation approach where distances (paths) are calculated between pairs of randomly selected nodes. This is a classical technique used in many previous works. In practical applications the distance distribution of shortest path queries can significantly differ from a random one. For example, during social search users tend to be looking for people they already know [9] that are much more likely to be closer to these users in a social graph comparing to a randomly selected ones.

To evaluate our methods for different distances we uniformly sample 100 nodes and calculate shortest path distances to all nodes in a graph. As a result we obtain a set of queries consisting from a start node, an end node and distance between them. For each found distance value we select up to 1000 queries. The list of the selected queries is used as an input to PLT-Basic, PLT-CE and PLT-BFS methods.

Figure 4.10 shows dependency of the absolute on distance for each of the methods with different numbers of landmarks k and numbers of landmarks/node r values. For simplicity reasons the results are shown for Orkut social graph only with highest degree landmark selection strategy.

The results show that all methods are more accurate on longer distances than the smaller ones. The only exception here is PLT-BFS, which always returns correct results for distance 1 as the algorithm checks all neighbors of start and end nodes.

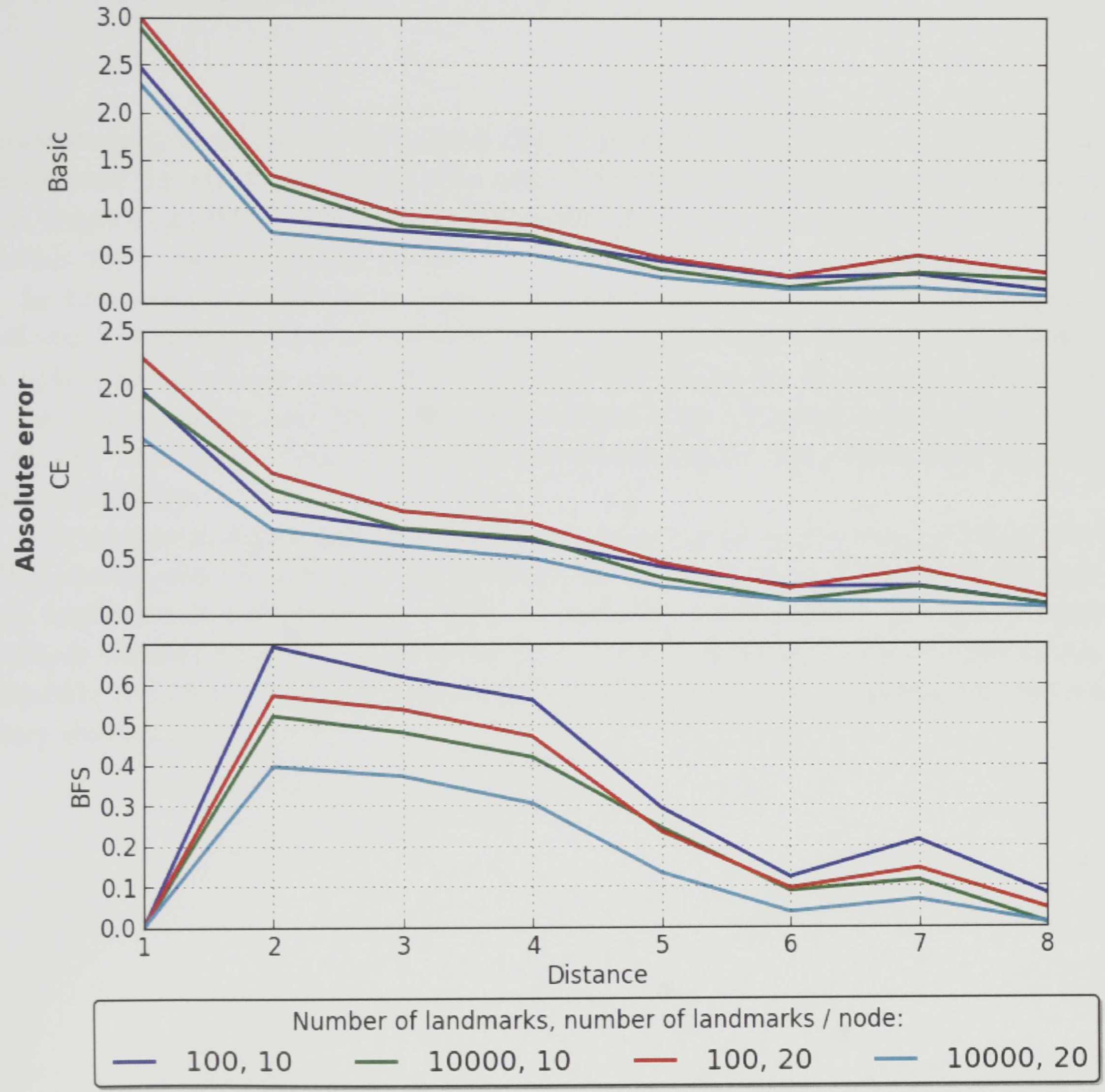


Figure 4.10: Absolute error per distance for Orkut (highest degree selection strategy)

Chapter 5

Conclusions

Shortest path computation is one of the most critical primitives in many graph algorithms. In recent years more and more interest is concentrated on large graphs that require applying approximate methods, as exact algorithms cannot always provide the necessary performance.

In this work we introduce *pruned landmark trees* – an improvement to the well-studied landmark-based shortest path approximation techniques. It is based on optimization of the shortest path tree index structure by storing paths only to the closest landmarks for every node rather than all landmarks as in regular methods. We study three shortest path approximation algorithms that use this optimized index.

The proposed algorithms were evaluated on four real-world large social graphs. All methods provide different trade-offs in terms of accuracy, average query time and implementation simplicity. The comparison with regular landmark-based methods showed that pruned landmark tree-based algorithms can be used to significantly reduce the memory usage while achieving both comparable accuracy and query execution time.

Future work

We believe that the idea behind pruned landmark trees is a promising direction for many scalable graph analysis techniques. One of the areas of future work is to explore more deeply the impact of combination of algorithm parameters such as numbers of landmarks and landmarks per node with properties of different graphs.

Secondly, we would like to study the generalization of the PLT approach to directed and weighted graphs. Brief ideas of generalization to directed graphs are mentioned in Section 3.1. In order to make our algorithms work with weighted edges the PLT-Precompute procedure has to be modified to use priority queue instead of a regular queue.

Finally, it should be noted that the described methods apply to static graphs. Adding even a single node or an edge can change shortest paths for a significant portion of node pairs. We plan to experiment with mechanisms and heuristics for a real-time index modification, which would make this approach applicable for a larger number of real-world practical problems.

Mälusäästlik kiire ligikaudne lühima tee otsing suurtes graafides

Magistritöö (30 EAP)

Volodymyr Floreskul

Resümee

Lühima tee otsing on üks olulisematest graafi algoritmidest. Suurte graafide korral tihti tekib vajadus kasutada selleks aga ligikaudseid meetodeid, kuna täpsed algoritmid on talumatult aeglased. Üks populaarne, lihtne, ning hästi skaleeruv ligikaudsete lühima tee otsimise meetodite pere põhineb *orientiiride (landmarks)* ideel. Nimelt kui ette arvutada kaugusi igast tipust x ühte väljavalitud orientirtippu u, saab iga tipu s ja t vahelise kauguse lähendada kasutades kolmnurga võrratust:

$$d(s,t) \approx d(s,u) + d(u,t),$$

kus $d(s,u)$ ning $d(u,t)$ on ettearvutatud väärtused.

Tulemuse täpsust saab suurendada, suurendades kasutatavate orientiirtippude arvu. Sel juhul tuleb valida k erinevat orientiiri ning arvutada ette kaugused igast tipust igasse orientiiri. Lisaks saab meetodit modifitseerida selliseks, et ta annaks välja mitte ainult kaugust, vaid ka teed tipust s tippu t.

Käesolevas töös me tutvustame lihtsat, kuid võimsat modifikatsiooni sellele lähenemisele, mida nimetame *pügatud orientiiride puuks*. Modifikatsiooni idee baseerub sellel faktil, et enamasti piisab salvestada mitte kõik kaugused sõlmest kõigesse orientiiridesse, vaid ainult r lähima orientiirini (kus r võib olla kõvasti väiksem kui k). Kui lisaks sellele arvutada ette kõik orientiiridevahelised kaugused, on võimalik s ja t vahelist kaugust lähendada järgmisel viisil:

$$d(s,t) \approx d(s,u) + d(u,v) + d(v,t),$$

kus $d(s,u)$ on ettearvutatud kaugus s-st mõnda tema ümbruses asuva orientiirtipuni u; $d(v,t)$ on ettearvutatud kaugus tipust t mõnda tema ümbruses asuva orientiirtipuni v, ning $d(u,v)$ on ettearvutatud kahe orientiirtipu vaheline kaugus. Selle meetodi täpsus on sarnane traditsioonilisele, kuid annab võimaluse kõvasti säästa andmestruktuuride poolt kasutatava ruumi.

Oma töös me pakume algoritmi pügatud orientiiride puu arvutamiseks ning kolm lühima tee algoritmi, mis on inspireeritud traditsiooniliste orientiiride-põhiste meetodite poolt: PLT-Basic, PLT-CE ja PLT-BFS.

Pakutud meetodite lähendamise täpsust ning kiirust testisime suurte sotsiaalvõrkude graafide peal: DBLP, Orkut, Twitter ja Skype. Mõõdetud sai algoritmi parameetrite (kasutatavate orientiiride koguarv, orientiiride arv sõlme kohta, orientiiride valimisstrateegia) mõju tulemuse lähenduse kvaliteedile. Kokkuvõttes saame öelda, et kuigi siiamaani keegi kahjuks eriti juba ei loe, kirjeldatud algoritmid pakuvad erinevaid kompromisse täpsuse, keskmise päringu täitmise aega, ja realisatisooni lihtsuse vahel.

Saadud tulemused olid võrreldud traditsiooniliste orientiiridel baseeruvate algoritmite tulemustega. Võrdlus näitas et pakutud lahendus lubab märgatavalt vähendada algoritmide poolt kasutatava mälu, jättes täpsust ja päringu täitmise aega suuresti samasuguseks.

Appendix A

Theorems and proofs

Theorem A.1. *The number of landmarks for each node x selected by the PLT-Precompute algorithm is equal to $|L(x)| = min(r, k')$, where r is the maximum number of landmarks per node and k' is number of landmarks in the connected component C of node x.*

Informal proof. Case 1: $k' \leq r$. All nodes in the same connected component are reached by all k' landmarks. BFS exploration starting from every landmark never stops as the current number of landmarks can never exceed the maximal value r. $\forall x \in V\, |L(x)| = k'$.

Case 2: $k' > r$. Assume that $\exists x \in V : |L(x)| < r$. According to the algorithm for every node y that is adjacent to the node x: $L(y) \subset L(x)$ (all landmarks from adjacent nodes have reached the current node x). But as $|L(y)| \leq |L(x)| < r$ and x is connected with y, then $L(x) \subset L(y)$. Therefore $L(x) \equiv L(y)$. When we continue to apply this method to all nodes adjacent to every node y and so on, we get that for every node in C the set of selected nodes is equal to $L(x)$. $\bigcup\limits_{z \in C} L(z) \equiv L(x)$. Then the total number of landmarks in the current component is $k' = |L(x)| = r$, which contradicts our assumption. Therefore for every node x: $|L(x)| = r$.

□

Theorem A.2. *The PLT-Precompute algorithm selects the set $L(x)$ of the closest landmarks for each node x.*

Informal proof. Assume that after the completion of the PLT-Precompute algorithm, $\exists x \in V : \exists u \notin L(x) \, and \, \exists l \in L(x) : |\pi_{u,x}| < |\pi_{l,x}|$, where $\pi_{u,x}$ is the shortest path between u and x and $\pi_{l,x}$ is the shortest path between l and x. By definition of the algorithm $u \in L(u)$. Therefore $\exists p : \pi_{u,x} = (u, .., v_{p-1}, v_p, .., x)$, where $u \in L(v_{p-1}) \, and \, u \notin L(v_p)$. π_{u,v_p} and $\pi_{v_p,x}$ are shortest paths as subpaths of the shortest path $\pi_{u,x}$.

Let analyze landmarks $L(v_p)$. Since u is not selected as a landmark for the node v_p then according to the algorithm, this can happen only if $|L(v_p)| = r$, where r is the maximum number of landmarks per node. For each $q \in L(v_p)$: $|\pi_{q,v_p}| \leq |\pi_{u,x}|$, where π_{q,v_p} is some path (does not have to be a shortest path) that

is selected by the PLT-PRECOMPUTE algorithm. Since $\pi_{v_p,x}$ is a shortest path then according to the algorithm every landmark $q \in L(v_p)$ achieves the node x by the path $\pi_{q,x} = \pi_{q,v_p} + \pi_{v_p,x}$, where $\pi_{q,x} \leq \pi_{u,x}$, and is included to $L(x)$ unless there exist some landmark q' that achieves the node x first by the path $\pi_{q',x}$, where $|\pi_{q',x}| \leq |\pi_{q,x}|$. Therefore for each $l \in L(x)$ $\pi_{l,x} \leq \pi_{u,x}$ and thus a contradiction is achieved. $\square$

Bibliography

[1] Caesar M. Godfrey P. B. Agarwal, R. and B. Y. Zhao. Shortest paths in less than a millisecond. In *5th Workshop on Online Social Networks (WOSN)*, 2012.

[2] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[3] Atish Das Sarma, Sreenivas Gollapudi, Marc Najork, and Rina Panigrahy. A sketch-based distance oracle for web-scale graphs. In *Proceedings of the third ACM international conference on Web search and data mining*, WSDM '10, pages 401–410, New York, NY, USA, 2010. ACM.

[4] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[5] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345–, June 1962.

[6] Andrew V. Goldberg and Chris Harrelson. Computing the shortest path: A* search meets graph theory. In *Proc. 16th ACM-SIAM Symposium on Discrete Algorithms*, pages 156–165, 2005.

[7] Andrey Gubichev, Srikanta J. Bedathur, Stephan Seufert, and Gerhard Weikum. Fast and accurate estimation of shortest paths in large graphs. In *CIKM '10: Proceeding of the 19th ACM conference on Information and knowledge management*, pages 499–508. ACM, 2010.

[8] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media?

[9] Cliff Lampe, Nicole Ellison, and Charles Steinfield. A face(book) in the crowd: social searching vs. social browsing. In *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, CSCW '06, pages 167–170, New York, NY, USA, 2006. ACM.

[10] Michael Ley and Patrick Reuther. Maintaining an online bibliographical database: the problem of data quality." in egc, ser. revue des nouvelles technologies de l'information, vol. rnti-e-6. *CépaduèsÉditions*, 2006:5–10, 2006.

[11] Alan Mislove, Massimiliano Marcon, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. Measurement and analysis of online social networks. In *In Proceedings of the 5th ACM/USENIX Internet Measurement Conference (IMC'07*, 2007.

[12] N. J. Nilsson P. E. Hart and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems, Science, and Cybernetics*, SSC-4(2):100–107, 1968.

[13] Michalis Potamias, Francesco Bonchi, Carlos Castillo, and Aristides Gionis. Fast shortest path distance estimation in large networks. In *CIKM '09: Proceeding of the 18th ACM conference on Information and knowledge management*, pages 867–876, New York, NY, USA, 2009. ACM.

[14] Konstantin Tretyakov, Abel Armas-Cervantes, Luciano García-Bañuelos, Jaak Vilo, and Marlon Dumas. Fast fully dynamic landmark-based estimation of shortest path distances in very large graphs. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, CIKM '11, pages 1785–1794, New York, NY, USA, 2011. ACM.

[15] Monique V. Vieira, Bruno M. Fonseca, Rodrigo Damazio, Paulo B. Golgher, Davi de Castro Reis, and Berthier Ribeiro-Neto. Efficient search ranking in social networks. In *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, CIKM '07, pages 563–572, New York, NY, USA, 2007. ACM.

[16] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, Jun 1998.

[17] Xiaohan Zhao, Alessandra Sala, Christo Wilson, Haitao Zheng, and Ben Y. Zhao. Orion: shortest path estimation for large social graphs. In *Proceedings of the 3rd conference on Online social networks*, WOSN'10, pages 9–9, Berkeley, CA, USA, 2010. USENIX Association.