Efficiently Indexing Shortest Paths by Exploiting Symmetry in Graphs[*]

Yanghua Xiao[†]   Wentao Wu[†]   Jian Pei[‡]   Wei Wang[†]   Zhenying He[†]

[†]Department of Computing and Information Technology, Fudan University, Shanghai, China
{shawyh, weiwang1, zhenying}@fudan.edu.cn, {wentaowu1984}@gmail.com
[‡]School of Computing Science, Simon Fraser University, Burnaby, BC, Canada
jpei@cs.sfu.ca

ABSTRACT

Shortest path queries (SPQ) are essential in many graph analysis and mining tasks. However, answering shortest path queries on-the-fly on large graphs is costly. To online answer shortest path queries, we may materialize and index shortest paths. However, a straightforward index of all shortest paths in a graph of N vertices takes $O(N^2)$ space. In this paper, we tackle the problem of indexing shortest paths and online answering shortest path queries. As many large real graphs are shown richly symmetric, the central idea of our approach is to use graph symmetry to reduce the index size while retaining the correctness and the efficiency of shortest path query answering. Technically, we develop a framework to index a large graph at the orbit level instead of the vertex level so that the number of breadth-first search trees materialized is reduced from $O(N)$ to $O(|\Delta|)$, where $|\Delta| \leq N$ is the number of orbits in the graph. We explore orbit adjacency and local symmetry to obtain compact breadth-first-search trees (compact BFS-trees). An extensive empirical study using both synthetic data and real data shows that compact BFS-trees can be built efficiently and the space cost can be reduced substantially. Moreover, online shortest path query answering can be achieved using compact BFS-trees.

1. INTRODUCTION

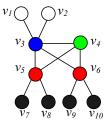Shortest path queries (SPQ) are essential in many graph

Figure 1: Graph G as our Running example

analysis and mining tasks. For example, in metabolic network analysis, for a given pair of compounds, the shortest pathway is particularly interesting [19]. In a large communication network, the shortest paths are important for system resource management [18, 5]. Moreover, shortest paths are also important in characterizing the internal structure of a large graph [27, 23].

Answering shortest path queries on-the-fly on large graphs is costly. To find the shortest path between a pair of vertices u and v, a straightforward approach is to start a recursive breadth-first search from u until v is reached. Such a straightforward method has time complexity $O(N + M)$ on a graph of N vertices and M edges [9].

To achieve online shortest path query answering, a materialization approach is to pre-compute and index the shortest paths between every pair of vertices in a large graph offline so that any shortest path queries can be answered online in almost constant time. In an undirected graph of N vertices, there are $N(N-1)/2$ pairs of vertices and thus at least that many shortest paths. A straightforward implementation of the materialization strategy takes $O(N^2)$ space. For large graphs, the space cost is often a critical concern.

In this paper, we tackle the problem of indexing shortest paths and online answering shortest path queries. The objective is to reduce the space cost of the indexes and simultaneously achieve online shortest path query answering. Our method is motivated by the fact that symmetry extensively exists in large graphs [11, 29, 30, 28]. While we will review the formal definition of graph symmetry in Section 2, let us illustrate the intuition using an example.

EXAMPLE 1 (SYMMETRY). *Consider graph G in Figure 1. Vertices v_1 and v_2 have the following property: for any vertex $v \in (V(G) - \{v_1, v_2\})$, the shortest path between v_1 to v and the shortest path between v_2 to v differ only on edges (v_1, v_3) and (v_2, v_3). As will be shown in Section 2, this property can be captured by an automorphic equivalence rela-*
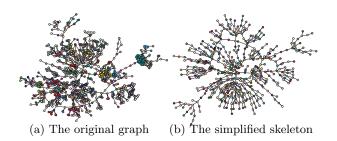
(a) The original graph   (b) The simplified skeleton

Figure 2: The supervision relationship graph between Ph.D. students and their advisors in theoretical computer science, and its simplified skeleton.

tion. Using this property, the shortest paths between v_1 and the vertices other than v_2 can be used to obtain the shortest paths between v_2 and those vertices immediately. In other words, as long as we record the symmetry information about v_1 and v_2, we do not need to compute or store the shortest paths of v_2. Some space can be saved.

Generally, automorphic equivalence may involve more than just a pair of vertices. For example, as will be shown in Section 2, vertices v_7, v_8, v_9, v_{10} are automorphically equivalent, which can lead to saving in computing and storing the shortest paths involving those vertices. □

Roughly speaking, a graph is symmetric if some vertices are automorphically equivalent to each other. Using symmetry we may reduce a large graph substantially to a simplified skeleton which preserves many essential properties of the original graph and removes the structural redundancy. For example, Figure 2(a) shows a social network of the supervision relationship between Ph.D. students and their advisors in theoretical computer science [14]. There are $1,025$ vertices and $1,043$ edges in the graph. After compression using symmetry, the simplified skeleton as shown in Figure 2(b) contains only 511 vertices and 525 edges.

The above observation motivates us to exploit symmetry in large graphs to index shortest paths in a space-efficient way for online shortest path query answering. To the best of our knowledge, we are the first to exploit symmetry for shortest path indexing and query answering. In this paper, we tackle the problem and make the following contributions. First, we propose a theoretical framework of using symmetry to compress BFS-trees for shortest paths, and present the corresponding algorithms. Second, we devise compact BFS-trees which can reduce the space cost even more. Last, we present a systematic experimental study to verify our design.

The rest of the paper is organized as follows. In Section 2, we recall the preliminaries in shortest path analysis and the notion of symmetry in graphs. We present the framework of our approach in Section 3. In Section 4, we justify the correctness of our method and discuss how to compress at the orbit level. In Section 5, we develop compact BFS-trees as the space-efficient indexes for shortest paths. We present a systematic experimental study in Section 6, and discuss related work in Section 7. The paper is concluded in Section 8.

2. GRAPH SYMMETRY

In this section, we review the preliminaries of graphs and give the formal specification of symmetry in graphs.

2.1 Preliminaries

We consider *undirected graphs* $G = (V, E)$ where V is a set of *vertices* and $E \subseteq V \times V$ is a set of *edges*. We also write the set of vertices as $V(G)$ and the set of edges as $E(G)$.

A *path* P in a graph G is a sequence of vertices $v_1, v_2, \ldots, v_k$, where $v_i \in V(G)$ $(1 \le i \le k)$ and $(v_j, v_{j+1}) \in E(G)$ $(1 \le j < k)$. Vertices v_1 and v_k are *linked* by P and are called the *ends* of P. The *length* of P is the number of edges on it, i.e., $len(P) = k - 1$. Path P is *simple* if all vertices on the path are unique, i.e., $v_i \ne v_j$ for any $1 \le i, j \le k$. P is a *cycle* if $v_1 = v_k$. By default, we consider simple paths. For vertices v_j and v_{j+1} on a path P, we also write $(v_j, v_{j+1}) \in P$.

A graph G is *connected* if for any two vertices $u, v \in V(G)$, $u \ne v$, there is a path between u and v. A graph G is *simple* if it has no self loops, i.e., $(v, v) \notin E(G)$ for any $v \in V(G)$. In this paper, by default we consider connected simple graphs.

A graph is *acyclic* if the graph contains no cycle. A *free tree* is a connected, acyclic, undirected graph. A *rooted tree* is a free tree in which one of the vertices (the root) is distinguished from the others. An *ordered tree* is a rooted tree in which the children of each vertex (if any) are ordered.

2.2 Shortest Paths and Breadth-first-search

In a graph G, path P with ends $u, v \in V(G)$ $(u \ne v)$ is a *shortest path* if there does not exist another path P' between u and v such that $len(P') < len(P)$. In a connected graph, there exists at least one shortest path between two vertices.

To keep our discussion simple, by default we only consider one shortest path between a pair of vertices. Our discussion can be straightforwardly extended to find all shortest paths between a pair of vertices.

To find a shortest path between vertices u and v in graph G, we can conduct a breadth-first search starting from u and obtain a *breadth-first search tree* (*BFS-tree* for short) T_u [9]. To start, we initialize T_u as a tree having only the root node u. Then, all neighbors of u are added into T_u as the children of u. That is, if $(u, x) \in E(G)$, x is added into T_u as a child of u. Iteratively, for each leaf node x in the current T_u, we search all neighbors y of x such that $(x, y) \in E(G)$ and y has not been added into T_u yet, and add y into T_u as a child of x. The iteration continues level by level (i.e., in a breadth-first manner) until v is added into T_u. The path from u to v in the BFS-tree T_u gives a shortest path between u and v in G.

The above breadth-first search procedure can be extended to find a shortest path between u and every vertex $v \in V(G) - \{u\}$. We only need to run the iterations until all vertices in $V(G) - \{u\}$ are added into the BFS-tree T_u.

EXAMPLE 2 (BFS-TREES). *Consider graph G in Figure 1. By a breadth-first search starting from vertex v_1, we can obtain the BFS-tree T_{v_1} in Figure 3(a). Each path from the root vertex v_1 to a vertex x $(x \ne v_1)$ in the tree T_{v_1} is the shortest path from v_1 to x in G.* □

If we materialize the BFS-tree for every vertex in a graph, and index the shortest paths between every pair of vertices using an array or a hash table, shortest path queries can be answered online.

In the above simple materialization method, the time complexity to compute all shortest paths is $\Theta(NM)$ where N is the number of vertices and M is the number of edges. The space complexity of the materialization is $\Theta(N^2)$.

(a) T_{v_1} and T_{v_2} (b) T_{v_3} (c) T_{v_4}

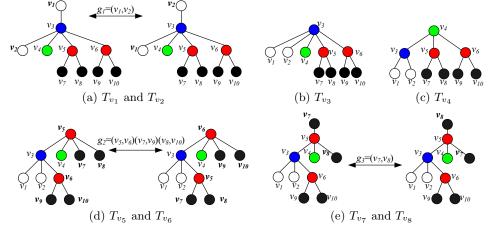(d) T_{v_5} and T_{v_6} (e) T_{v_7} and T_{v_8}

Figure 3: BFS-trees

2.3 Symmetry in Graphs

Graph symmetry is an important topic in algebraic graph theory [4, 12]. Here, we review some basic concepts.

For graph $G = (V, E)$, a *permutation* $f : V \mapsto V$ *acting on* V is a *bijective mapping* from V onto itself. For a vertex $u \in V$, f maps u to vertex u^f. Obviously, *the inverse correspondence* f^{-1} of f is also a permutation.

Let $S(V)$ be the set of all permutations acting on V. For permutations $f, g \in S(V)$, the product $h = f \circ g$ (or simply fg) is the mapping $h : V \mapsto V$ such that for each $u \in V$, $u^h = (u^f)^g$. Apparently, the product of two permutations is also a permutation, i.e., $f \circ g \in S(V)$.

Let $f \in S(V)$. The action of f on V can be extended to an *induced action* on $V \times V$ as follows: for $(a, b) \in V \times V$, we define $(a, b)^f = (a^f, b^f)$. For the set of edges $E \subseteq V \times V$, we define $E^f = \{(a, b)^f | (a, b) \in E\}$. If $E^f = E$, we call f an *automorphism* of E.

Let $Aut(G)$ be the set of all automorphisms of a graph $G = (V, E)$. Mathematically, $(Aut(G), V)$ is a permutation group. In general, a graph G is *asymmetric* if $(Aut(G), V)$ contains no permutations other than the identity permutation e (i.e., $u^e = u$ for all vertices u). Otherwise, the graph is *symmetric*.

Automorphism group $(Aut(G), V)$ defines a relation $=_{Aut(G)}$ on V as follows. For two vertices $u, v \in G$, $u =_{Aut(G)} v$ if there exists an automorphism $f \in Aut(G)$ such that $u^f = v$. Due to the identity permutation $e \in Aut(G)$, relation $=_{Aut(G)}$ is reflexive. Since $u^f = v$ implies $v^{f^{-1}} = u$, relation $=_{Aut(G)}$ is symmetric. Moreover, if $x^f = y$ and $y^g = z$, $x^{fg} = z$. Thus, relation $=_{Aut(G)}$ is transitive. Therefore, $=_{Aut(G)}$ is an equivalence relation, and is usually called *automorphic equivalence*.

We can partition the set $V(G)$ using equivalence relation $=_{Aut(G)}$. Let $\mathbf{\Delta}(G) = V(G)/ =_{Aut(G)} = \{\Delta_1, \Delta_2, \ldots, \Delta_k\}$. That is, $u, v \in \Delta_i$ $(1 \leq i \leq k)$ if $u =_{Aut(G)} v$. $\mathbf{\Delta}(G)$ is called the *automorphism partitioning*. An entry $\Delta_i \in \mathbf{\Delta}(G)$ is called an *orbit* of $Aut(G)$. An orbit is *trivial* if it contains only one vertex. Otherwise, the orbit is *non-trivial*. We also write $\mathbf{\Delta}(G)$ as $\mathbf{\Delta}$ when G is clear from context.

Orbits are important in our study. A non-trivial orbit captures the set of vertices which can be compressed in shortest path computation. We will discuss the details in the later sections.

In $Aut(G)$, a subset $F = \{f_1, \ldots, f_k\} \subseteq Aut(G)$ is a set of *generators* if every permutation $f \in Aut(G)$ can be written as a product of some permutations in F, but for every proper subset F' of F, there exists at least one permutation $f \in Aut(G)$ such that f cannot be written as a product of some permutations in F'.

Let f be a permutation acting on vertex set $V(G)$ of graph $G(V, E)$. The *support* of permutation f is the set of vertices that f moves, that is, $supp(f) = \{v \in V(G) | v^f \neq v\}$. A permutation is usually written as a union of some disjoint cycles. For example, $f = (a, b, c)(d, e)$ means $a^f = b, b^f = c, c^f = a$ and $d^f = e, e^f = d$. In the cycle representation, all vertices that f does not move are omitted.

Two permutations f and g are *disjoint* if their supports are exclusive, i.e., $supp(f) \cap supp(g) = \emptyset$. Moreover, two sets of permutations F_1 and F_2 are *support-disjoint* if for any $f \in F_1$ and $g \in F_2$, f and g are disjoint.

For a non-identity automorphism $f \in Aut(G)$ ($f \neq e$), if $f = f_1 f_2$ ($f_1, f_2 \in Aut(G), supp(f_1) \cap supp(f_2) = \emptyset$) implies $f_1 = e$ or $f_2 = e$, then f is *indecomposable*. Otherwise, f is *decomposable*. In other words, an indecomposable automorphism cannot be rewritten as a product of disjoint non-identity permutations, and thus can be considered as redundancy-free.

EXAMPLE 3 (CONCEPTS). *For graph G in Figure 1, we can find a set of generators of $Aut(G)$ consisting of the following automorphisms $g_1 = (v_1, v_2)$, $g_2 = (v_5, v_6)(v_7, v_9)(v_8, v_{10})$, $g_3 = (v_7, v_8)$, and $g_4 = (v_9, v_{10})$.*

Due to g_2, we have $v_7 =_{Aut(G)} v_9$ and $v_8 =_{Aut(G)} v_{10}$. Furthermore, due to g_3, we have $v_7 =_{Aut(G)} v_8$. Consequently, v_7, v_8, v_9, v_{10} are in the same orbit of the graph. Given all the automorphisms, we can obtain the automorphism partitioning of the graph $\mathbf{\Delta} = \{\Delta_1, \Delta_2, \Delta_3, \Delta_4, \Delta_5\}$, where $\Delta_1 = \{v_1, v_2\}$, $\Delta_2 = \{v_3\}$, $\Delta_3 = \{v_4\}$, $\Delta_4 = \{v_5, v_6\}$ and $\Delta_5 = \{v_7, v_8, v_9, v_{10}\}$. The automorphism partitioning is consistent with our observations in Example 1.

It is easy to check that $supp(g_1) = \{v_1, v_2\}$, $supp(g_3) = \{v_7, v_8\}$ and $supp(g_2) = \{v_5, v_6, v_7, v_8, v_9, v_{10}\}$. Hence, g_1 and g_3 are disjoint, but g_2 is not disjoint with respect to g_1 or g_3. $g_1 g_3 = (v_1, v_2)(v_7, v_8)$ is decomposable since it is the product of two disjoint non-identity automorphisms g_1 and g_3. However, g_1, g_2, g_3 and g_4 are indecomposable. □

Table 1: Automorphism Storage

Orbit	Base vertex	Mirrored vertex and automorphism
Δ_1	v_1	$\langle v_2, g_1 \rangle$
Δ_4	v_5	$\langle v_6, g_2 \rangle$
Δ_5	v_7	$\langle v_8, g_3 \rangle, \langle v_9, g_2 \rangle, \langle v_{10}, g_3 g_2 \rangle$

Algorithm 1: The framework

Input: a graph G;
Output: **T**: A set of compact BFS-trees;
1   Obtain symmetry information and partition $V(G)$ into orbits;
2   **foreach** *non-trivial orbit* Δ_i **do**
3   select a base vertex $u \in \Delta_i$ and compute the automorphisms $f_{u,u'}$ for all $u' \in \Delta_i$ and $u' \neq u$;
4   **end**
5   conduct a breadth-first search to build the compact BFS-trees for the base vertices of all orbits;

---

3. THE FRAMEWORK

As indicated in Example 3, vertices v_1 and v_2 in graph G in Figure 1 are in the same orbit. Figure 3(a) shows the BFS-trees of v_1 and v_2. Interestingly, the two BFS-trees can be mapped to each other under the action of automorphism $g_1 = (v_1, v_2)$. Each path in T_{v_1} can be mapped to a corresponding path in T_{v_2} under the action of g_1.

Generally, for vertices in the same orbit, their BFS-trees can be mapped to each other under an automorphism. Figures 3(b), 3(c), 3(d) and 3(e) show more examples. Table 1 shows the mapping automorphisms in the non-trivial obits in graph G in Figure 1.

Based on the above observation, we can reduce the cost of computing and storing shortest paths. For each orbit Δ, we select only one vertex $u \in \Delta$ in the orbit as the *base vertex*, and generate the corresponding BFS-tree T_u. For other vertices $u' \in \Delta$, we record the automorphism $f_{u,u'}$ that maps the BFS-tree T_u to $T_{u'}$.

If a shortest path query involves at least one base vertex, the query can be answered directly using the BFS-tree of the base vertex. If both vertices u_1 and v_1 in a shortest path query are not the base ones, we can find the base vertex u of the orbit to which u_1 belongs, and the shortest paths between u and v_1 in the BFS-tree T_u. The shortest paths between u_1 and v_1 are given by applying the automorphism f_{u,u_1} on those shortest paths between u and v_1.

In this paper, we adopt the nice implementation in `nauty` [17] to obtain the symmetry information including orbits and a set of generators. The framework of our method is shown in Algorithm 1.

4. ORBIT-BASED COMPRESSION

In this section, we first justify the correctness of our method. Then, we discuss how to compute automorphisms for non-base vertices.

4.1 Shortest Paths Using Orbits

Apparently, automorphisms have the following property which enables us to find isomorphic subgraphs.

LEMMA 1 (AUTOMORPHISMS AND SUBGRAPHS). *Let $H = (V', E')$ be a subgraph of $G = (V, E)$, i.e., $V' \subseteq V$ and $E' \subseteq E$. For any automorphism $f \in Aut(G)$,*

$H^f = (V'^f, E'^f)$ *is a subgraph of G (i.e., $V'^f \subseteq V$ and $E'^f \subseteq E$) and isomorphic to H.* □

Moreover, automorphisms are significant for counting unique isomorphic copies of subgraphs. Let $V' \subseteq V$ be a subset of vertices. The *induced subgraph* of V' is $G(V') = (V', E_{V'})$, where $E_{V'} = \{(u, v) \in E | u, v \in V'\}$.

LEMMA 2. *Consider graph $G = (V, E)$ and a subset $V' \subseteq V$. For an automorphism $f \in Aut(G)$, if $V'^f = V'$ then $G(V')^f = G(V')$.*

PROOF. *We only need to show $E'^f = E'$. Since $f \in Aut(G)$ is an automorphism, $E'^f \subseteq E$. For any edge $(u, v) \in G(V')$, $(u, v)^f \in E'$ due to $V'^f = V'$. Hence, $E'^f \subseteq E \cap V' \times V' = E'$. Since f is bijective, we have $E'^f = E'$, and thereby $G(V')^f = G(V')$.* □

An orbit captures a set of vertices which can be compressed due to symmetry. For a given graph G, how much can we compress? To understand this question, technically we need to explore, given an induced subgraph $G(V')$ of G, how many unique isomorphic copies of $G(V')$ exist in G.

For a graph $G = (V, E)$ and a set of vertices $Q \subseteq V$, an automorphism $f \in Aut(G)$ is a *setwise stabilizer* with respect to Q if $Q^f = Q$, where $Q^f = \{v^f | v \in Q\}$. We denote the set of all setwise stabilizers with respect to Q by $SS(G, Q)$. As shown in [24], $SS(G, Q)$ is a subgroup of $Aut(G)$. Moreover, we have the following result.

THEOREM 1 (COMPRESSIBILITY). *Let $G = (V, E)$ be a graph, $V' \subseteq V$.*

$$|G(V')^{Aut(G)}| = \frac{|Aut(G)|}{|SS(G, V')|}$$

where $G(V')^{Aut(G)} = \{G(V')^f | f \in Aut(G)\}$.

PROOF. *$SS(G, V')$ is a subgroup of $Aut(G)$ [24]. Thus, the theorem is an immediate consequence of the* Lagrange's Theorem *[20], which states that if H is a subgroup of P and P is a finite group then $|P| = |H| \cdot [P : H]$ where $[P : H]$ is the number of different cosets.* □

Theorem 1 indicates that the number of unique isomorphic copies of $G(V')$ depends on two factors, the size of $Aut(G)$ and the setwise stabilizers with respect to V'. One important intuition here is that a setwise stabilizer with respect to V' maps $G(V')$ to itself and thus is redundant in subgraph enumeration.

Automorphisms also have the following nice property.

LEMMA 3 (SHORTEST PATHS AND AUTOMORPHISMS). *Let P be a shortest path between vertices u and v in graph G. For any automorphism $f \in Aut(G)$, P^f is a shortest path between u^f and v^f, where $P^f = \{(x^f, y^f) | (x, y) \in P\}$.*

Proof Sketch: Using Lemma 1, we can show that P^f is a path between u^f and v^f. Lauri and Scapellato [15] showed that automorphisms are geodesic-preserving. That is, $d(u, v) = d(u^f, v^f)$ where $d(,)$ is the geodesic distance that measures shortest distance between a vertex pair. Therefore, $len(P) = len(P^f)$.

Assume that P is a shortest path between u and v. If P^f is not a shortest path between u^f and v^f, there must be a shortest path P' which is shorter than P^f. Clearly, $P'^{f^{-1}}$ is a shorter path than P between u and v. Contradiction. □

Now, we are ready to prove the correctness of our method.

THEOREM 2 (BFS-TREES AND AUTOMORPHISMS). *For graph G, let T_v be a BFS-tree rooted at vertex $v \in V(G)$. For any vertex u which is in the same orbit with v, there exists an automorphism $f \in Aut(G)$ such that $v^f = u$ and T_v^f is a BFS-tree rooted at v^f, where $T_v^f = \{(x^f, y^f)|(x, y) \in T_v\}$.*

Proof Sketch: Since u and v are in the same orbit, there must exist at least one $f \in Aut(G)$ such that $u = v^f$. Now, we show T_v^f is a BFS-tree rooted at u.

According to Lemma 3, each path in T_v^f is a shortest path starting from $u = v^f$. For each vertex $w \in V(G)$, a shortest path between v and $w^{f^{-1}}$ must appear in T_v. Let the path be P. Then, P^f is a path in T_v^f. Moreover, P^f is a shortest path between u and w. ☐

EXAMPLE 4 (BFS-TREES AND AUTOMORPHISMS). *In Figure 3 and Table 1, $T_{v_1}^{g_1} = T_{v_2}$, $T_{v_5}^{g_2} = T_{v_6}$, $T_{v_7}^{g_3} = T_{v_8}$, $T_{v_7}^{g_2} = T_{v_9}$ and $T_{v_7}^{g_3 g_2} = T_{v_{10}}$.* ☐

4.2 Generating BFS-trees for an Orbit

Section 4.1 shows that, in a non-trivial orbit Δ, we can select one vertex $v \in \Delta$ as the *base vertex* and generate the BFS tree. Any vertex in Δ can serve as the base vertex. For other vertices u in the same orbit, we can find an automorphism f to map T_v to answer any shortest path queries involving u. How can we find automorphisms for non-base vertices?

EXAMPLE 5 (CHOICE OF AUTOMORPHISMS). *In our running example (Figure 3 and Table 1), let us choose v_1 as the base vertex in orbit Δ_1. Two automorphisms g_1 and $g_1 g_3$ can map v_1 to v_2. In other words, the automorphism for v_2 is not unique. Which one is more preferable?*

It is easy to see $|supp(g_1)| < |supp(g_1 g_3)|$. Hence, in terms of storage cost, g_1 is a better choice than $g_1 g_3$. ☐

In general, in an orbit Δ where v is the base vertex, for a non-base vertex v', there may exist more than one automorphism mapping v to v'. Let $\mathcal{G}_{v \to v'} = \{f : v^f = v'\}$ be the set of automorphisms that map v to v'. Our task is to choose one automorphism from $\mathcal{G}_{v \to v'}$ for answering the shortest path queries for v'.

The cost of storing an automorphism f can be quantified as $\Theta(|supp(f)|)$. Therefore, it is desirable to use the automorphism f of the minimum $|supp(f)|$ value. However, we conjecture that the following problem is NP-hard.

CONJECTURE 1 (OPTIMAL AUTOMORPHISM SELECTION). *The following problem is NP-hard.*
Instance: *a graph $G(V, E)$, a positive number n, vertices $v, v' \in V$.*
Question: *is there an automorphism $f \in \mathcal{G}_{v \to v'}$ such that $|supp(f)| \le n$?* ☐

To tackle the problem in a practical way, we utilize a set of generators **Gens** of the automorphism group returned by symmetry computation algorithm nauty [17]. Each automorphism in **Gens** is indecomposable (part (1) of Theorem 2.34 in [17]), and thus can be regarded as redundancy-free. As will be shown in our experimental results, in practice the storage cost of those automorphisms is very low. Thus, if one of those automorphisms can map the base vertex v to a non-base vertex v', we use it for v'.

Algorithm 2: getAut(u,v,**P**)

Input: Two vertices u, v
Output: **P**: the set of automorphisms such that the production of all automorphisms in **P** is an automorphism mapping u to v

```
1  foreach p ∈ Gens do
2      if u^p == v then
3          P = P ∪ {p}; return true;
4      end
5      if !visited[u^p] then
6          P = P ∪ {p}; visited[u^p] = true;
7          if getAut(u^p, v, P) then
8              return true;
9          else
10             P = P − {p}; visited[u^p] = false;
11         end
12     end
13 end
```

However, we may not be able to find an indecomposable automorphism $f \in \mathcal{G}_{v \to v'}$ directly from **Gens** for a non-base vertex v' such that $v^f = v'$. In such a case, we have to search the possible products of the automorphisms in **Gens**. The procedure is shown in Algorithm 2.

Before calling the procedure, the variable *visited* for each vertex is initialized to *false* except for vertex u. When we encounter an automorphism p such that u^p has been visited (which implies that the product of a segment of automorphism sequence $p_i, p_{i+1}, \ldots, p_j$ ($p_j = p$) in **P** $u^p = u$), the search process along the current path can be terminated, since we can find a shorter automorphism sequence to transform vertex u to v.

Each possible product of the automorphisms in **Gens** leads to a candidate automorphism. Hence, Algorithm 2 is exponential in the worst case. However, as shown in our experimental results, for real networks, in most cases we can find the desirable automorphisms from **Gens** quickly.
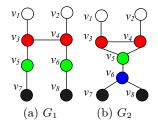
Clearly, the total space cost to store the automorphisms for non-based vertices is $O((|V(G)| - |\Delta|)\bar{p})$, where $|\Delta|$ is the number of orbits and $\bar{p}$ is the average support length of the automorphisms for non-based vertices. Since each orbit has only one BFS-tree, the storage cost for BFS-trees is reduced from $\Theta(|V(G)|^2)$ to $\Theta(|\Delta||V(G)|)$. Now, the last question in this section is how we can estimate $\bar{p}$ and compare it to the cost of storing a BFS-tree.

We say a graph $G(V, E)$ to be *globally symmetric* if there exists an indecomposable automorphism $g \in Aut(G)$ such that $supp(g) = V(G)$. Otherwise, we say the graph to be *locally symmetric*. It has been shown that real large graphs are unlikely globally symmetric [11]. Hence, the key is to quantify the degree to which a graph is locally symmetric. For this purpose, we define a measure φ_G as

$$\varphi_G = \frac{\max_{g \in ID(G)}\{|supp(g)|\}}{|V(G)|},$$

where $ID(G)$ is the set of indecomposable automorphisms of graph G. Intuitively, the larger φ_G, the closer to global symmetric G is.

EXAMPLE 6 (φ_G). *In Figure 4(a), there exists an indecomposable automorphism $f = (v_1, v_2)(v_3, v_4)(v_5, v_6)(v_7, v_8)$*
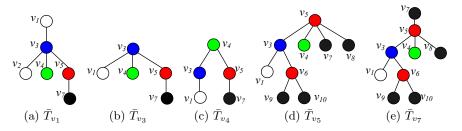
Figure 4: Local symmetry and global symmetry.

(a) $\bar{T}_{v_1}$ (b) $\bar{T}_{v_3}$ (c) $\bar{T}_{v_4}$ (d) $\bar{T}_{v_5}$ (e) $\bar{T}_{v_7}$

Figure 5: compact BFS-trees

Table 2: Statistics of some real graphs (Avg and Max are the average orbit length and the maximal orbit length, respectively)

Graph	N	$|\Delta|$	$r_G\%$	M	$\varphi_G\%_0$	Avg	Max
PPI	1458	1019	69.89	1948	4.11	1.43	46
Yeast	2284	1852	81.09	6646	2.63	1.23	34
Homo	7020	6066	86.41	19811	0.57	1.15	44
P-fei1738	1738	1176	67.66	1876	5.75	1.48	10
Geom	3621	2803	77.41	9461	1.66	1.29	13
Erdos02	6927	2365	34.14	11850	3.46	2.93	142
DutchElite	3621	1907	52.67	4310	2.21	1.90	49
Eva	4475	898	20.07	4652	4.47	4.98	545
California	5925	4009	67.66	15770	1.01	1.48	46
Epa	4253	2212	52.01	8897	0.94	1.92	115
InternetAs	22442	11392	50.76	45550	0.27	1.97	343

moving all vertices, hence $\varphi_{G_1} = 100\%$. In Figure 4(b), $\varphi_{G_2} = 50\%$ since $g = (v_1, v_2)(v_3, v_4)$ is indecomposable and maximal in terms of cardinality of the support. Notice that, automorphism $h = (v_1, v_2)(v_3, v_4)(v_7, v_8)$ of G_2 moves more vertices than g, however h is decomposable into the composition of two disjoint automorphisms of G_2: g and $g' = (v_7, v_8)$, thereby cannot be selected as the maximal one to compute φ_{G_2}. □

In Table 2, we give φ_G for a variety of real large graphs. Please refer to [28] for the details of each graph. It is evident that real graphs are locally symmetric with the value of φ_G at the order of magnitude of 10^{-3} or 10^{-4}. Local symmetry of real graphs implies that the automorphisms required to generate BFS-trees of non-base elements can be stored using very small storage space, thousands or tens of thousands times smaller than storing a BFS-tree. Moreover, the support of an automorphism in a graph G is at most $|V(G)| \cdot \varphi_G$. Thus, the storage cost of automorphisms is $O((|V(G)| - |\Delta|)|V(G)|\varphi_G)$.

5. COMPACT BFS-TREES

The size of BFS-trees can be further reduced by utilizing symmetry in a graph.

EXAMPLE 7 (COMPACTING BFS-TREES). *In our running example (Figure 1), $\Delta_4 = \{v_5, v_6\}$ and $\Delta_5 = \{v_7, v_8, v_9, v_{10}\}$. Let us try to compact the BFS-trees T_{v_1}, T_{v_3} and T_{v_4} in Figure 3 using the two orbits.*

We can compact a BFS-tree such that only one vertex in an orbit is used as long as we can ensure the adjacency of the remaining vertices, as shown in Figure 5. It is easy to see that the shortest path P_{v_1,v_6} can be obtained from path P_{v_1,v_5} (under the action of g_2). Moreover, the shortest paths

from v_1 to v_8, v_9, v_{10}, respectively, can be obtained from the shortest path P_{v_1,v_7} under the action of g_3, g_2 and g_3g_2, respectively.

Not all orbits can be compacted in the above way. For example, in T_{v_7}, $\{v_5, v_6\}$ cannot be compacted, since the length of the shortest path between v_7 and v_5 and that between v_7 and v_6 are different. Consequently, the shortest paths cannot be mapped to each other under any automorphism. □

In this section, we show that whether an orbit can be compacted in a BFS-tree is highly related to the reachability relation between this orbit and the root orbit, the orbit to which the root vertex belongs. We formulate our ideas using orbit adjacency and orbit reachability. Then, we develop the compact BFS-tree index structure and an efficient algorithm for shortest path query answering using compact BFS-trees.

5.1 Orbit Adjacency and Reachability

If some vertex in orbit Δ_i is adjacent to some vertex in orbit Δ_j, we say that orbits Δ_i and Δ_j are *adjacent*. An orbit can be adjacent to itself. A sequence $\Delta_1 \Delta_2 \cdots \Delta_k$ of orbits is called as an *orbit path* if Δ_i is adjacent to Δ_{i+1} for each $1 \leq i < k$. If there is no repeated orbit in an orbit path, we call the orbit path a *simple orbit path*.

Let Δ_i and Δ_j be two orbits in graph G. For vertex $v \in \Delta_i$, $N_j(v) = \{(u, v) \in E(G) | u \in \Delta_j\}$ is the set of neighbors of v that belong to orbit Δ_j. Orbit Δ_i is said to be *strongly adjacent* to Δ_j if for any two vertices $v, v' \in \Delta_i$, $N_j(v) = N_j(v')$.

If there exists an orbit path $\Delta_1 \Delta_2 \cdots \Delta_k$ such that Δ_i is strongly adjacent to Δ_{i+1} for $1 \leq i < k$, we say that orbit Δ_1 is *strongly reachable* to orbit Δ_k along the orbit path.

LEMMA 4 (STRONG ADJACENCY). *If orbit Δ_i is*

strongly adjacent to orbit Δ_j, then for any vertex u in Δ_i, $N_j(u) = \Delta_j$.

PROOF. *For any vertex $u \in \Delta_i$, $N_j(u) \subseteq \Delta_j$. Suppose there exists a vertex $y \in \Delta_j$ but not in $N_j(u)$, which implies that y will not be adjacent to any vertex in Δ_i. Let $x \in N_j(u)$ be a vertex adjacent to vertex $u \in \Delta_i$. There must exist some automorphism $f \in Aut(G)$ such that $x^f = y$ and $(u,x)^f \in E(G)$. Thus, we have $u^f \in \Delta_i$, which implies that y is adjacent to some vertex in Δ_i. Contradiction.* $\square$

Lemma 4 immediately leads to the fact that the strongly adjacent relation is symmetric. Another immediate consequence of Lemma 4 is that the induced subgraph of any two strongly adjacent orbits Δ_i and Δ_j, i.e. $G(\Delta_i \cup \Delta_j)$, is a complete bipartite.

Similarly, we can define weak adjacency and weak reachability between orbits in a graph. Orbit Δ_i is *weakly adjacent* to Δ_j if there exist $u, v \in \Delta_i$ such that $N_j(u) \neq N_j(v)$. Clearly, weak adjacency for orbits is also a symmetric relation. In simple graphs (i.e., no self-loops), an orbit is always weakly-adjacent to itself.

Moreover, orbit Δ_1 is said to be *weakly reachable* to orbit Δ_k if there exists an orbit path $\Delta_1 \Delta_2 \cdots \Delta_k$ such that Δ_i is weakly adjacent to Δ_{i+1} for $1 \leq i < k$. Otherwise, Δ_1 is *not weakly reachable* to orbit Δ_k, which means that there certainly exist two orbits strongly adjacent to each other in all possible orbit paths from Δ_1 to Δ_k.

We can extend the weak reachability relation from orbits to vertices. For a vertex v in a graph G, let $Orb(v)$ be the orbit to which v belongs. For vertices u, v in G, if $Orb(u)$ is weakly reachable to $Orb(v)$, we said u to be *weakly reachable* to v.

Apparently, trivial orbits (i.e., orbits containing only one vertex) have the following property.

LEMMA 5 (TRIVIAL ORBITS). *In graph G, let Δ be a trivial orbit. For any vertex v' such that $(v,v') \in E(G)$, Δ is strongly adjacent to $Orb(v')$. Moreover, for any orbit $\Delta' \neq \Delta$, Δ is not weakly reachable to Δ'.* $\square$

It is easy to see that both the weak reachability relation on orbits and the weak reachability relation on vertices are equivalence relations. Consequently, we can obtain two partitionings using the two relations. Using the weak reachability on orbits, we can obtain partitioning $\bar{\Theta}(G) = \{\bar{\Theta}_1, \bar{\Theta}_2, \ldots, \bar{\Theta}_s\}$ of all orbits in a graph. Using the weak reachability relation on vertices, we can obtain partitioning $\Theta(G) = \{\Theta_1, \Theta_2, \ldots, \Theta_{s'}\}$ of all vertices. It is not difficult to show that, for each $\Delta_j \in \Delta(G)$, there exists a $\Theta_i \in \Theta(G)$ such that $\Delta_j \subseteq \Theta_i$. That is, $\Theta(G)$ is a partitioning coarser than $\Delta(G)$. The induced subgraph of each Θ_i in graph G is called a *weakly reachable area* in the graph.

EXAMPLE 8 ($\Theta(G)$). *In the graph in Figure 4(b), $\Delta(G_2) = \{\Delta_1, \Delta_2, \Delta_3, \Delta_4, \Delta_5\}$ where $\Delta_1 = \{v_1, v_2\}$, $\Delta_2 = \{v_3, v_4\}$, $\Delta_3 = \{v_5\}$, $\Delta_4 = \{v_6\}$ and $\Delta_5 = \{v_7, v_8\}$. Since Δ_1 is weakly adjacent to Δ_2, they are in the same unit in Θ. $\Theta(G_2) = \{\{v_1, v_2, v_3, v_4\}, \{v_5\}, \{v_6\}, \{v_7, v_8\}\}$.* $\square$

Let $A(\Delta_i)$ be the set of all indecomposable automorphisms mapping vertices in the same orbit to each other, that is, $A(\Delta_i) = \{g \in ID(G) : u^g = v$, for any vertex pair $u, v \in \Delta_i\}$. Let $A(\Theta_i) = A(\Delta_{i_1}) \cup A(\Delta_{i_2}), \ldots, \cup A(\Delta_{i_s})$, where

$\Delta_{i_1} \cup, \ldots, \cup \Delta_{i_s} = \Theta_i$. The following result shows that each indecomposable automorphism of a graph only moves vertices within a certain weakly reachable area.

LEMMA 6 (SUPPORT OF INDECOMPOSABLE AUTOMORPHISM). *For each $g \in A(\Theta_i)$, $supp(g) \subseteq \Theta_i$.*

Proof Sketch: A permutation in $S(V)$ is either a cycle or a product of disjoint cycles [20]. Each cycle can be represented as a permutation in $S(V)$. 1-cycle can be represented as the identity permutation e. Hence, any permutation g can be decomposed into $g = g_1 g_2 \cdots g_s g_{s+1} \cdots g_t$. Such a decomposition is called the complete factorization, *which is unique except for the order in which the factors occur [20].*

Let $F(g)$ be the set of factors of complete factorization for an automorphism $g \in Aut(G)$. Then, there exists a surjective mapping from $F(G)$ onto $\Delta(G)$. For any $g_i \in F(g)$, $supp(g_i)$ is a subset of some orbit of the graph. Hence, without loss of generality, we can assume that $supp(g_1) \cup supp(g_2), \ldots, \cup supp(g_s) \subseteq \Theta_i$ and $supp(g_{s+1}) \cup supp(g_{s+2}), \ldots, \cup supp(g_t) \subseteq V - \Theta_i$.

Let $g \in A(\Theta_i)$. If $supp(g)$ is not a subset of Θ_i, then $g'' = g_{s+1}, \ldots, g_t \neq e$. Let $g' = g_1 g_2, \ldots, g_s$. Then, $g = g'' g'$ and $supp(g') \cap supp(g'') = \emptyset$. It's not difficult to show that g' is an automorphism of the graph.

Since g' is an automorphism and $Aut(G)$ is a group, g'^{-1} is an automorphism. Consequently, $g'' = gg'^{-1}$ is an automorphism too. Since $supp(g') \cap supp(g'') = \emptyset$, $g' \neq e$, $g'' \neq e$, g is a decomposable automorphism, which contradicts to the assumption that $g \in A(\Theta_i)$. $\square$

Following Lemma 6, the indecomposable automorphisms of a graph have the following property.

LEMMA 7. *$A(\Theta_i)$ and $A(\Theta_j)$ $(i \neq j)$ are support disjoint. Moreover, $A(\Delta_i)$ and $A(\Delta_j)$ are support disjoint if $\Delta_i \subseteq \Theta_m$, $\Delta_j \subseteq \Theta_n$ and $m \neq n$.* $\square$

Now, we are ready to show in what situations an orbit in a BFS-tree can be further compacted using a representative vertex in the orbit.

THEOREM 3 (CONDITION OF ORBIT COMPACTING). *The length of the shortest path between a vertex in Δ_i and a vertex in Δ_j $(i \neq j)$ is a constant if one of the following conditions holds: (1) $A(\Delta_i)$ and $A(\Delta_j)$ are support-disjoint, (2) $\Delta_i \subseteq \Theta_m$, $\Delta_j \subseteq \Theta_n$ and $m \neq n$, or (3) Δ_i is not weakly reachable to Δ_j.*

Proof Sketch: Consider $u_1, u_2 \in \Delta_i$ and $v_1, v_2 \in \Delta_j$ $(i \neq j)$. There exist $g_1 \in A(\Delta_i)$ and $g_2 \in A(\Delta_j)$ such that $u_1^{g_1} = u_2$, $v_1^{g_2} = v_2$ and $supp(g_1) \cap supp(g_2) = \emptyset$. To show that the first condition in the theorem is sufficient, we need to show that a shortest path P_{u_1,v_1} between u_1 and v_1 has the same length as a shortest path P_{u_2,v_2} between u_2 and v_2. This can be shown using the results in [15] that automorphisms are geodesic-preserving.

Condition 1 is a consequence of Condition 2. Condition 2 is equivalent to condition 3. $\square$

Please note that the conditions in Theorem 3 are sufficient but are not necessary. Moreover, for orbits in a weakly reachable area, the lengths of the shortest paths between orbits are not necessarily a constant. For example, in the graph shown in Figure 4(b), orbit $\Delta_i = \{v_1, v_2\}$ is weakly adjacent to $\Delta_2 = \{v_3, v_4\}$. It is easy to check that the shortest path between v_1 and v_3 has a length different from that between v_1 and v_4.

Algorithm 3: CompactBFS(u)

Input: A vertex u
Output: Compact BFS-tree $\bar{T}_u$
1 if $|Orb(u)| > 1$ **then**
2 $\quad$ $WR(u) \leftarrow WeaklyReachableOrbits(Orb(u))$;
3 end
4 $que.push(u)$;
5 $visited[u] = 1$;
6 while $!que.empty()$ **do**
7 $\quad$ $w \leftarrow que.pop()$;
8 $\quad$ **foreach** $v \in Neighbors(w)$ **do**
9 $\quad\quad$ **if** $|Orb(u)| > 1$ **and** $Orb(v) \in WR(u)$ **then**
10 $\quad\quad\quad$ **if** $!visited[v]$ **then**
11 $\quad\quad\quad\quad$ $visited[v] = 1$;
12 $\quad\quad\quad\quad$ $que.push(v)$;
13 $\quad\quad\quad$ **end**
14 $\quad\quad$ **else**
15 $\quad\quad\quad$ **if** $!visited[v]$ **and** $!orbit\_visited[Orb(v)]$ **then**
16 $\quad\quad\quad\quad$ $visited[v] = 1$;
17 $\quad\quad\quad\quad$ $orbit\_visited[Orb(v)] = 1$;
18 $\quad\quad\quad\quad$ $que.push(v)$;
19 $\quad\quad\quad$ **end**
20 $\quad\quad$ **end**
21 $\quad$ **end**
22 end

LEMMA 8. *Let* $\mathbf{Orb}(g)$ *be the set of orbits of vertices in* $supp(g)$ *where* g *is an indecomposable automorphism of graph* G. *If* $|\mathbf{Orb}(g)| > 1$, *orbits in* $\mathbf{Orb}(g)$ *are weakly reachable to each other.*

PROOF. *We only need to show that* $\mathbf{Orb}(g)$ *is a subset of some* Θ_i. *From Lemma 6, we have* $supp(g) \subseteq \Theta_i$. *Thus, all orbits in* $\mathbf{Orb}(g)$ *are also in* Θ_i. $\qquad\square$

5.2 Compact BFS-trees

In a breadth-first search starting from vertex u, when we meet vertex v, whether $Orb(v)$ can be compacted depends on the reachability relation between $Orb(u)$ and $Orb(v)$. In Theorem 3, we show that only when $Orb(v)$ is not weakly reachable to $Orb(u)$, $Orb(v)$ can be compacted. Let $WR(u)$ be the set of orbits that are weakly reachable to $Orb(u)$. When $Orb(v)$ is weakly reachable to $Orb(u)$ or $Orb(v) \in WR(u)$, $Orb(v)$ cannot be compacted.

Based on the above idea, we define a compact BFS-tree as follows. A compact BFS-tree $\bar{T}_u$ of graph G is a tree generated by the breadth-first search procedure starting from vertex u. For each orbit of $Aut(G)$ that is not weakly reachable to orbit $Orb(u)$, only one vertex in the orbit is traversed.

The algorithm framework of the traverse procedure to generate a compact BFS-tree is shown in Algorithm 3. Before the algorithm starts, for each vertex u, $visited[u]$ and $orbit\_visited[Orb(u)]$ are initialized to 0, which are omitted in Algorithm 3. In the algorithm, when $Orb(u)$ is a trivial orbit, $Orb(v)$ is not weakly reachable to $Orb(u)$ (by Lemma 5). $Orb(v)$ can be compacted (lines 15 to 19). When $Orb(u)$ is non-trivial, if $Orb(v) \in WR(u)$, $Orb(v)$ is weakly reachable to $Orb(u)$; else $Orb(v)$ is not weakly reachable to $Orb(u)$.

EXAMPLE 9 (COMPACT BFS-TREES). *Figure 5 shows all compact BFS-trees in our running example.* $\qquad\square$

Algorithm 4: QueryShortestPath(u,v)

Input: u:the source vertex; v:the destination vertex;
Output: P_{uv}: one shortest path from u to v
1 $u' \leftarrow rep(u)$;
2 Let $f_u \in \mathcal{G}_{u \rightarrow u'}$;
3 $v' \leftarrow v^{f_u}$;
4 if $u'! = u$ **and** $v'! = v$ **then**
5 $\quad$ $P_{u'v'} \leftarrow readpath(\bar{T}_{u'}, v')$;
6 $\quad$ **return** $P_{u'v'}^{f_u^{-1}}$;
7 else
8 $\quad$ **if** $u' = u$ **and** $P_{uv} \in \bar{T}_u$ **then**
9 $\quad\quad$ $P_{uv} \leftarrow readpath(\bar{T}_u, v)$;
10 $\quad\quad$ **return** P_{uv};
11 $\quad$ **end**
12 $\quad$ **foreach** $w \in Orb(v)$ **do**
13 $\quad\quad$ $v' \leftarrow w$;
14 $\quad\quad$ $f_v \in \mathcal{G}_{v \rightarrow v'}$;
15 $\quad\quad$ **if** $P_{u'v'} \in \bar{T}_{u'}$ **then**
16 $\quad\quad\quad$ $P_{u'v'} \leftarrow readpath(\bar{T}_{u'}, v')$;
17 $\quad\quad\quad$ **return** $P_{u'v'}^{f_u^{-1}f_v^{-1}}$;
18 $\quad\quad$ **end**
19 $\quad$ **end**
20 end

Using compact BFS-trees, all orbits that are not weakly reachable to the root orbit are compacted. In the best case where all orbits are not weakly reachable to the root orbit, the storage cost for a compact BFS-tree is $\Theta(|\mathbf{\Delta}|)$. Taking into account the storage cost of automorphisms, which is $O((|V(G)| - |\mathbf{\Delta}|)|V(G)|\varphi_G)$, we have the following result.

THEOREM 4. *For a graph* G, *the space complexity of the compact BFS-trees built in Algorithm 1 is* $O(|\mathbf{\Delta}||V(G)| + \alpha)$, *where* $\alpha = (|V(G)| - |\mathbf{\Delta}|)|V(G)|\varphi_G$. $\qquad\square$

5.3 Answering Shortest Path Queries on Compact BFS-trees

Given a vertex pair u and v, to find a shortest path between them, the key is to find the automorphism that can move u to u' and v to the v' such that $P_{u',v'}$ is a path in the compact BFS-tree $\bar{T}_{u'}$. Then, we only need to recover $P_{u,v}$ from $P_{u',v'}$ by applying the corresponding automorphism. Algorithm 4 shows the procedure, where two cases of the reachability relation between $Orb(u)$ and $Orb(v)$ need to be considered. If $Orb(u)$ and $Orb(v)$ are weakly reachable to each other, we can find an automorphism f that moves both u and v. Otherwise, we need to find two automorphisms that move u and v, respectively. The product of these two automorphisms is the desired automorphism.

Let $rep(u)$ be the base vertex in $Orb(u)$. When $u = u'$, f_u is the identity permutation e, which does not move any vertex. If $P_{u,v} \in \bar{T}_u$ (line 8), we can directly obtain the path without any extra operation (lines 9 and 10). If $P_{u,v}$ does not exist in $\bar{T}_u$, $Orb(v)$ must be not weakly reachable to $Orb(u)$. We need to find the materialized shortest path in $\bar{T}_{u'}$ that can be mapped to $P_{u,v}$, as well as the corresponding automorphism f_v (lines 12 to 19). When $u \neq u'$ and $v \neq v'$ (line 4), $Orb(u)$ and $Orb(v)$ are weakly reachable to each other[1], then f_u is the desired automorphism. Otherwise,

---
[1]Lemma 8 shows that orbits with vertices in the support of

Table 3: Compression rate and construction time on real graphs ($r_c = \dfrac{\text{compact BFS-tree index size}}{\text{BFS-tree index size}}, r_t = \dfrac{T_{BFS}}{T_{compBFS}}$).

| | Index Size | | | | | Index Construction Time (seconds) | | | | |
| | | | | Compact | | | | | Compact | |
Graph	BFS-trees(M)	TSize(M)	PSize(K)	BFS-tree(M)	r_c	BFS-trees	t_1	t_2	BFS-tree	r_t
PPI	12.1637	5.9442	9.88	5.954	48.9%	0.992	0.454	1.093	1.547	64%
Yeast	29.8500	19.4382	7.52	19.4457	65.1%	2.641	3.797	3.781	7.578	35%
Homo	281.9847	210.556	18.2	210.574	74.7%	26.969	1.485	52.5	53.985	50%
p-fei1738	17.2843	7.9168	7.5	7.9243	45.8%	1.219	0.438	2.25	2.688	45%
Geom	75.0254	44.9619	8.8	44.9907	59.97%	6.61	0.549	14.265	14.859	44%
Erdos02	274.5628	32.031	238.5	32.2695	11.8%	29.688	78.531	11.922	90.453	33%
DutchElite	75.0254	20.819	36.83	20.8559	27.8%	5.515	5.875	7.812	13.687	40%
Eva	114.5876	4.6354	909	5.5447	4.8%	7.656	287.422	2.11	289.532	3%
California	200.876	91.9763	49.68	92.026	45.8%	18.843	8.578	25.516	34.094	55%
Epa	103.5004	28.0094	189.8	28.1992	27.2%	9.344	31.156	5.984	37.14	25%
InternetAS	2881.8704	742.658	1416.78	744.07478	25.8%	347.891	1258.97	450.672	1709.64	20%

i.e., $u \neq u'$ and $v = v'$, $Orb(v)$ must be not weakly reachable to $Orb(u)$, we also need to execute lines 12 to 19. To look up a path in the tree (function $readpath()$), we only need to store a parent pointer for each node in the compact BFS-tree and recursively read the parent until the root vertex is reached.

EXAMPLE 10 (QUERY ANSWERING). *In our running example, let us find a shortest path between v_2 and v_9 from $\bar{T}_{v_1}$. Following Algorithm 4, we first need to find the automorphism that moves v_2 to its base vertex $rep(v_2) = v_1$. From Table 1, we find g_1 is the required automorphism. We can check that v_9 is fixed by g_1. Hence, lines 12 to 19 are executed. Since we do not know which vertex in $Orb(v_9)$ is selected to be traversed, we have to try each vertex in the orbit (line 12). As the result, we find that $P_{v_1v_7}$ is materialized in $\bar{T}_{v_1}$ and the automorphism mapping v_9 and v_7 to each other is g_2. We apply automorphism $g_1^{-1}g_2^{-1}$ on $v_1v_3v_5v_7$ to obtain the shortest path between v_2 and v_9, which is $v_2v_3v_6v_9$.*

As another example, let us consider the shortest path between vertex v_6 and v_8. From Table 1, we see that v_5 can be mapped to v_6 under the action of g_2; and v_8 is moved to v_{10} under the action of the permutation. The condition in line 4 is satisfied, which indicates that $Orb(v_6)$ and $Orb(v_8)$ are weakly reachable to each other. Hence, $P_{v_5v_{10}}$ must exist in $\bar{T}_{v_5}$. We obtain this shortest path, which is $v_5v_3v_6v_{10}$. Then we apply g_2^{-1} on $v_5v_3v_6v_{10}$, obtaining a vertex sequence $v_6v_3v_5v_8$, which is the answer to the query. □

The worst case for Algorithm 4 happens when a shortest path to be found is between two vertices that belong to two not weakly reachable orbits. Whether $P_{u',v'}$ is materialized in $\bar{T}_{u'}$ can be determined in constant time, since we can build a hash table for each compact BFS-tree where all vertices that are traversed in the compact BFS-tree are hashed. The time cost of Algorithm 4 is $O(|Orb(v)|)$. That is, the performance of the query answering algorithm is determined by the orbit length. Table 2 summarizes some statistics of real graphs, including the average orbit length (Avg) and the maximal orbit length (Max). It is clear from the table that most of the real graphs have an average orbit length

an indecomposable automorphism must be weakly reachable to each other. Here, in implementation, f_u is always selected from the generator set (**Gens**) of $Aut(G)$. Such a generator set is returned by `nauty` and the automorphisms in this set are indecomposable[17].

less than 2. Therefore, we can answer shortest path queries online using compact BFS-trees.

Alternatively, we may record the traversed vertex for each orbit as part of the index, which enables us to directly access $P_{u'v'}$ that is materialized in $\bar{T}_u'$ without trying each vertex in the orbit (line 12). Clearly, such a strategy can achieve $O(1)$ time complexity in query answering with the extra space overhead $O(|\Delta||V(G)|)$. Considering the fact that real graphs are locally symmetric with relatively small orbit length, it is reasonable to trade off index size for query performance.

6. EXPERIMENT RESULTS

We implemented the algorithms in C++, and carried out our experiments on a PC running Windows XP Professional Operating System with an Intel Pentium 2.0 GHz CPU and 2G main memory.

The space cost of our index structure consists of two parts: the size of compact BFS-trees TSize and the size of mirroring automorphisms PSize. The index construction time also includes two parts: the time to find all automorphisms between non-base vertices and base vertices (t_1) and that to generate all compact BFS-trees for base elements (t_2).

Efficiently indexing shortest paths for a general network is still a challenging open problem. To the best of our knowledge, no efficient solution is available yet. Hence, in our experiments, we compare our compact BFS-tree index with the baseline method which stores a BFS-tree for each vertex.

6.1 Results on Real Graphs

We collect a variety of real graph data including biological networks (PPI, Yeast and Homo), social networks (P-fei1738, Geom, Erdos02, DeutchElite, and Eva), information networks (California and Epa) and technological networks (InternetAS). Some statistics of those graphs are shown in Table 2. All these graphs have been shown to have symmetry to some extent.

We first show the speedup of shortest path query answering using compact BFS-trees in Table 4. In each graph, we randomly generate $10,000$ pairs of vertices and query the shortest paths between each pair of vertices. The query time reported in the table is the average on each data set. For comparison, we also report the query answering time by breadth-first search on-the-fly, that is, no index is used. Although each individual graph can fit in memory, query an-

Table 4: Query answering using compact BFS-trees.

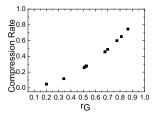| Graph	p-fei1738	Geom	Epa	DutchElite	Eva	California	Erdos02	PPI	Yeast
Compact BFS-trees (ms)	0.0266	0.0219	0.0250	0.0282	0.0438	0.0235	0.0265	0.0235	0.0203
No index (ms)	0.3468	0.9	0.9906	0.7954	0.8844	1.3937	1.5297	0.3125	0.5203
Speedup ($\frac{\text{time}_{\text{no index}}}{\text{time}_{\text{compact BFS-trees}}}$)	13.04	41.10	39.62	28.21	20.20	59.31	57.72	13.30	25.63

Figure 6: Relation between symmetry and compression rate on real graphs

(a) r_G

(b) Index size.

(c) Compression Rate

(d) Index Building Time

Figure 7: Scalability to graph size

swering using compact BFS-trees is an order of magnitude faster than breadth-first search on-the-fly.

We compare in Table 3 the size of the BFS-trees (no orbit compression at all) and the size of our compact BFS-tree indexes in those real graphs. Compact BFS-trees are substantially smaller than BFS-trees. In the best case (Eva), the size of the compact BFS-tree index is only 4.8% of the size of the BFS-trees. Please note that, in the table, the unit of Tsize is megabyte and the unit of Psize is kilobyte. In most cases, the storage cost of automorphisms is at least two orders of magnitude smaller than that of the compact BFS-trees. This clearly justifies the strategy of orbit-based compression: whenever possible, we should store an automorphism to generate a BFS-tree from the base vertex instead of storing a BFS-tree.

We also show in Table 3 the index construction time. Constructing compact BFS-trees takes longer time than constructing simple BFS-trees. However, the index construction is offline. An compact BFS-tree index is constructed once and used many times.

We notice that the more symmetric a graph, the longer the compact BFS-tree index construction time. In Eva, a highly symmetric graph, the time to compute automorphisms between non-base vertices and base vertices (t_1) dominates the index construction time.

To understand the relation between compression rate and symmetry in graphs, we plot Figure 6. We use $r_G = \frac{|\Delta|}{|V(G)|}$ to measure the degree of symmetry in a graph G, where Δ is the set of orbits. Clearly, the lower the value of r_G, the more symmetric a graph. The figure clearly shows that the more symmetric a graph, the more efficient our compact BFS-trees.

6.2 Results on Synthetic Data Sets

To test the scalability of our method, we generate synthetic graphs according to the BA model [3], a widely used model to simulate real graphs. In the data generation, when a new vertex u is added into the graph, u randomly connects to k vertices such that the probability that u connects to a vertex v already in the graph is proportional to the degree of v. The average degree of the graph generated is $2k$.
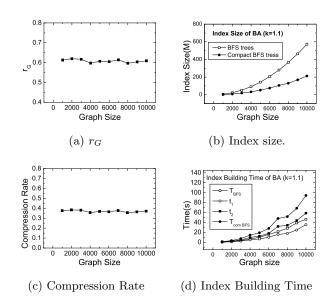
In the experiments in Figure 7, we set $k = 1.1$. We vary the graph size from 1000 to 10,000 vertices by step 1,000. Figure 7(a) plots the $r_G = \frac{|\Delta|}{|V(G)|}$ values of the data sets. It can be seen that the data sets have very similar r_G values. Thus, the graphs generated have similar degree of symmetry.

In Figure 7(b), we compare the index size of simple BFS-trees (no orbit compression) and our compact BFS-trees, respectively. As shown in Figure 7(b), the difference in size between the simple BFS-trees and the compact BFS-trees increases as the graph size increases. Figure 7(c) shows the compression rate, which remains constant with the growth of graph size. This clearly shows that compact BFS-trees can achieve consistent compression quality on large graphs.

In Figure 7(d), we also show the time t_1 for finding all mapping automorphisms and time t_2 for generating compact BFS-trees in our method. Constructing compact BFS-trees costs more in time. Again, index construction is offline.

To explore the influence of symmetry on compact BFS-trees, we generate synthetic graphs following the graph growth model under the principle of "similar linkage pattern" [30], where a parameter α is used to control the degree of symmetry of the graph. By fixing the graph size to 5,000 vertices and varying parameter α from 0 to 1 by step 0.05, we generate graphs of the same size but different degree of symmetry. The average degrees of networks generated are 2.92. Figure 8(a) shows the degree of symmetry (r_G) with respect to parameter α in the graphs generated.

Figures 8(b) and 8(c) show the size and the compression rate, respectively, of BFS-trees and compact BFS-trees with
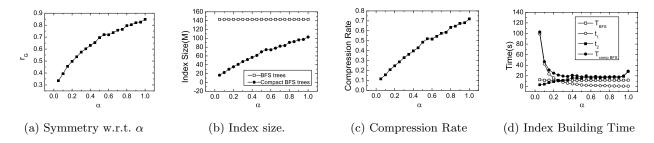
(a) Symmetry w.r.t. α (b) Index size.   (c) Compression Rate   (d) Index Building Time

Figure 8: Effect of symmetry on compact BFS-trees

respect to α. Clearly, the more symmetric a graph, the more effective compact BFS-trees. On the other hand, since the simple BFS-trees do not explore symmetry information in compression, the size of BFS-trees does not change. One interesting observation is that, even when $\alpha = 1$ ($r_G > 0.8$, i.e., the graph has low degree of symmetry), compact BFS-trees still can achieve a compression rate less than 80%. This indicates that even very few non-trivial orbits identified by our method can bring in good compression power.

Figure 8(d) shows the index construction time with respect to parameter α. When the graphs are highly symmetric (i.e., α is very small), the construction time of compact BFS-trees is high since there are many automorphisms need to be calculated. However, when α increases, the compact BFS-tree construction time decreases dramatically and becomes stable, comparable to the construction time of simple BFS-trees. The construction time of simple BFS-trees is relatively insensible to parameter α. In practice, large graphs often have local symmetry, but may not be very close to global symmetry. The offline construction of compact BFS-trees may often take only moderate extra cost than constructing simple BFS-trees, and thus is highly feasible.

In summary, our experimental results clearly show that compact BFS-trees as an index are effective in exploiting symmetry in graphs. Moreover, the construction of compact BFS-trees is feasible and affordable as an offline operation. Using compact BFS-trees, we can online answer shortest path queries efficiently and achieve the performance tens of times faster than not using indexes.

7. RELATED WORK

The most famous and widely used algorithm to solve the shortest path problem is Dijkstra [10], which is fast using heap data structures for priority queues [9]. Some faster algorithms for graphs with special constraints on edge weights are developed [8]. However, one assumption common to all those algorithms is that the whole graph can be stored in main memory. To tackle the problem when the size of a graph is too large to completely fit into main memory, Agrawal and Jagadish [2] first introduce the idea based on graph partitioning and materialization. A recent study [6] proposes an efficient algorithm as long as the materialized data can be held into main memory. All those methods focus on improving shortest path query answering performance, but do not consider materialization of shortest paths. Therefore, when the graph is large, those methods cannot achieve online query answering performance.

Very recently, Samet *et al.* [21] propose an algorithm to find the k nearest neighbors in a spatial graph. They also ex-

ploit the idea to pre-compute the shortest paths between all possible vertices in the graph. Such shortest path information is organized as a shortest path quadtree, which is based on the spatial coherence of spatial graphs. Their algorithm, however, does not study how to index the shortest paths for a general graph. Their method is complement to ours. As future work, it is interesting to explore how to integrate the two methods and how to apply symmetry information into the framework of [21] to further reduce the storage cost.

There are some other graph query problems closely related to shortest path queries, such as reachability queries. The simplest way to answer a reachability query is to traverse the graph at query time using depth-first or breadth-first search [9]. Another option is to pre-compute the transitive closure. The transitive closure of a graph is the set of node pairs (v, w) where a path from v to w exists. Although efficient algorithms are developed for computing transitive closure in relational databases [1, 16], the cost of storing transitive closure is $O(n^2)$, and the computation cost is $O(n^3)$. The high cost makes the transitive closure methods inapplicable to large graphs. Various indexing technologies are proposed [7, 22, 26, 25, 13] which give better performance than the naive method and the transitive closure materialization method. Some methods [7, 26, 13] adopt an interval code to efficiently answer reachability queries.

Recently, graph symmetry has attracted interests in the community of complex networks. It has been shown that various real networks are richly symmetric [11, 29, 30]. Such symmetry is commonly resulted from the presence of locally treelike or biclique-like structures which are present in many empirical networks, and are derived naturally from elementary growth processes such as growth with similar linkage patterns [30]. If we collapse all structural redundancy characterized by network symmetry, we can obtain a structural skeleton of the parent network – network quotient, which preserves various key functional properties of the parent network [28]. To the best of our knowledge, we are the first to exploit symmetry for indexing shortest paths and answering queries on large graphs.

8. CONCLUSIONS

Shortest path queries are important in many applications. In this paper, we tackle the problem of online answering shortest path queries by exploiting rich symmetry in graphs. We develop compact BFS-trees, a novel index structure for online shortest path queries. We show by experiments on real data sets and synthetic data sets that compact BFS-trees are effective and can be constructed efficiently. Moreover, compact BFS-trees can support online shortest path

queries efficiently.

It's worthwhile to point out that the equivalence between automorphically equivalent vertices is far beyond the shortest path. It has been shown that automorphically equivalent vertices have the same property under almost all general structural measurement of vertices, such as clustering coefficient and betweenness [11]. They also exhibits equivalence from other structural perspective, such as reachability to other vertices, DFS-trees rooted at the vertex and neighborhood graph of the vertex. As future work, it is interesting to explore how symmetry can be exploited to tackle other query answering and data analysis problems in large graphs. Moreover, how to extend our compact BFS-trees to weighted and directed graphs remains an interesting and challenging problem. Although `nauty` program is the most efficient algorithm ever known to calculate automorphism information of the network, its capability is limited and it can only handle networks with 20000 nodes or less without extra techniques. Hence, it is interesting to use local symmetry in real networks to improve the scalability of `nauty` program, so that the framework of exploiting network symmetry proposed in this paper can be applied to larger networks.

9. REFERENCES

[1] R. Agrawal and H. V. Jagadish. Direct algorithms for computing the transitive closure of database relations. In *Proceedings of the 13th International Conference on Very Large Data Bases*, 1987.

[2] R. Agrawal and H. V. Jagadish. Algorithms for searching massive graphs. *IEEE Trans. on Knowl. and Data Eng.*, 6(2), 1994.

[3] A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286, 1999.

[4] N. Biggs. *Algebraic Graph Theory*. Cambridge University Press, 1974.

[5] S. Boccalettia, V. Latora, Y. Moreno, M. Chavez, and D.-U. Hwang. Complex networks: Structure and dynamics. *Physics Reports*, 424, 2006.

[6] E. P. F. Chan and N. Zhang. Finding shortest paths in large network systems. In *Proceedings of the 9th ACM international symposium on Advances in geographic information systems*, 2001.

[7] L. Chen, A. Gupta, and M. E. Kurul. Stack-based algorithms for pattern matching on dags. In *Proceedings of the 31st international conference on Very large data bases*, 2005.

[8] B. V. Cherkassky, A. V. Goldberg, and T. Radzik. Shortest path algorithms: Theory and experimental evaluation. *Mathematical Programming*, 73, 1996.

[9] T. H. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to algorithms*. MIT Press, Cambridge, MA, USA, 2001.

[10] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1959.

[11] B. D.MacArthur, R. J.Sánchez-García, and J. W.Anderson. Symmetry in complex networks. *Discrete Applied Mathematics*.

[12] C. Godsil and G. Royle. *Algebraic Graph Theory, volume 207 of Graduate Texts in Mathematics*. Springer, 2001.

[13] H. He, H. Wang, J. Yang, and P. S. Yu. Compact reachability labeling for graph-structured data. In *Proceedings of the 14th ACM international conference on Information and knowledge management*, 2005.

[14] D. S. Johnson. The genealogy of theoretical computer science: a preliminary report. *SIGACT News*, 16(2), 1984.

[15] J. Lauri and R. Scapellato. *Topics in Graph Automorphisms and Reconstruction*. Cambridge University Press, 2003.

[16] H. Lu. New strategies for computing the transitive closure of a database relation. In *Proceedings of the 13th International Conference on Very Large Data Bases*, 1987.

[17] B. D. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30.

[18] R. Pastor-Satorras and A. Vespignani. *Evolution and Structure of the Internet: A Statistical Physics Approach*. Cambridge University Press, New York, NY, USA, 2004.

[19] S. A. Rahman and D. Schomburg. Observing local and global properties of metabolic pathways: 'load points' and 'choke points' in the metabolic networks. *Bioinformatics*, 22(14), 2006.

[20] J. J. Rotman. *An Introduction to the Theory of Groups, Fourth Edition*. Springer, 1999.

[21] H. Samet, J. Sankaranarayanan, and H. Alborzi. Scalable network distance browsing in spatial databases. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, New York, NY, USA, 2008. ACM.

[22] R. Schenkel, A. Theobald, and G. Weikum. Efficient creation and incremental maintenance of the hopi index for complex xml document collections. In *Proceedings of the 21st International Conference on Data Engineering*, 2005.

[23] J. Scott. *Social Network Analysis: A Handbook, Second Edition*. Sage Publications, London, 2000.

[24] G. Tinhofer and M. Klin. *Algebraic combinatorics in mathematical chemistry. Methods and algorithms. III. Graph Invariants and Stabilization Methods (Preliminary Version)*. Technical Report, TUM-M9902, Technische Universitat Munchen, 1999.

[25] S. Trißl and U. Leser. Fast and practical indexing and querying of very large graphs. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, 2007.

[26] H. Wang, H. He, J. Yang, P. S. Yu, and J. X. Yu. Dual labeling: Answering graph reachability queries in constant time. In *Proceedings of the 22nd International Conference on Data Engineering*, 2006.

[27] S. Wasserman and K. Faust. *Social Networks Analysis*. Cambridge University Press, Cambridge, 1994.

[28] Y. Xiao, B. D.MacArthur, H. Wang, M. Xiong, and W. Wang. Network quotients: Structural skeletons of complex systems. *Physical Review E*, 78.

[29] Y. Xiao, W. Wu, H. Wang, M. Xiong, and W. Wang. Symmetry-based structure entropy of complex networks. *Physica A*, 387.

[30] Y. Xiao, M. Xiong, W. Wang, and H. Wang. Emergence of symmetry in complex networks. *Physical Review E*, 77.