Solving Graph Isomorphism using Parameterized Matching

Juan Mendivelso[1], Sunghwan Kim[2], Sameh Elnikety[3], Yuxiong He[3], Seung-won Hwang[2], and Yoan Pinzón[1]

[1] Universidad Nacional de Colombia, Colombia
[2] POSTECH, Republic of Korea
[3] Microsoft Research, Redmond, WA, USA

Abstract. We propose a new approach to solve graph isomorphism using parameterized matching. To find isomorphism between two graphs, one graph is linearized, *i.e.*, represented as a graph walk that covers all nodes and edges such that each element is represented by a parameter. Next, we match the graph linearization on the second graph, searching for a bijective function that maps each element of the first graph to an element of the second graph. We develop an efficient linearization algorithm that generates short linearization with an approximation guarantee, and develop a graph matching algorithm. We evaluate our approach experimentally on graphs of different types and sizes, and compare to the performance of VF2, which is a prominent algorithm for graph isomorphism. Our empirical measurements show that graph linearization finds a matching graph faster than VF2 in many cases because of better pruning of the search space.

1 Introduction and Related Work

Graphs are widely used in many application domains, and graph isomorphism is a fundamental problem that appears in graph processing techniques of many applications including pattern analysis, pattern recognition and computer vision as discussed in a recent survey [8]. Graph isomorphism is a challenging problem: Given two graphs, we search for a bijective mapping from each element of the first graph to an element of the second graph such that both data and structural properties match. Data properties include node and edge attributes and types, and structural properties maintain the adjacency relations.

A naive solution could search for all possible mappings, facing an exponential search space. Surprisingly, the exact complexity of graph isomorphism is not determined yet [9], but likely to be NP-Complete. Notice however, graph subisomorphism, which is a closely related but a different problem, is NP-Complete [9]. Existing algorithms for graph isomorphism include Nauty Algorithm [19], Ullmann Algorithm [22] and VF2, a more recent algorithm [9]. All these algorithms have exponential worst case performance (since isomorphism is a hard problem). Except for some easy cases, solving isomorphism generally takes much longer time if there is no match, since all possible mappings are progressively

searched until shown not to lead to an isomorphism. Several heuristics, however, are employed to find likely mappings quickly. A good algorithm for graph isomorphism should find isomorphic graphs quickly in many cases.

In this paper we apply parameterized matching to solve graph isomorphism. Parameterized matching [4] was introduced to efficiently track down duplicate code in large software systems. It determines if two strings have the same structure. Specifically, two equal-length strings parameterized-match if there exists a bijective function f for which every text symbol in one string is equal to the image under f of the corresponding symbol in the other string. Brenda Baker [4] introduced this problem in 1993, and research work [1–3, 5–7, 10, 12–17, 20, 21] extends parameterized matching. A survey on parameterized matching is presented in [18].

Our approach to solve graph isomorphism has two main steps, linearization, and matching. First, in the linearization step, one of the graphs is represented as a graph walk that visits each node and edge, such that each element is represented as a parameter. This linear sequence is used in the second step for matching, which parameterized-matches the graph linearization against the other graph, to search for mapping.

This approach allows us to incorporate optimizations for both linearization and parameterized matching steps. Although we focus on presenting and evaluating the fundamental approach, we point out several attractive features of this approach. For example, this approach supports general graph models, such as attributed multi-graphs (in which nodes and edges may have arbitrary attributes, and several edges may connect two nodes). The graph statistics, such as node degree distribution and histograms of attribute values can be easily integrated in the linearization step to provide better linearization. The matching algorithm is embarrassingly parallel, enabling efficient implementation on multi-core machine and distributed frameworks.

We present the algorithms, correctness and complexity analysis of these two steps and implement them for experimental evaluation using graph of several types and sizes. We also compare to an optimized implementation of VF2, which is one of most widely used algorithms for graph isomorphism. Our empirical results show that in many cases, the graph linearization approach provides shorter response times, and the improvements increase with the graph size.

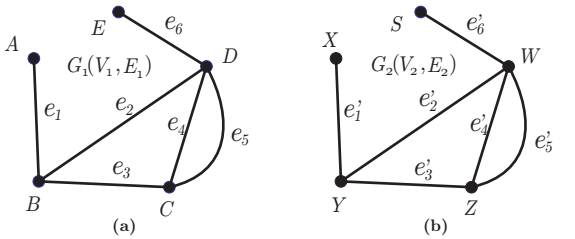
Our contributions are the following: (1) We propose a new approach to graph isomorphism using parameterized matching (Section 3). (2) We develop an efficient linearization algorithm to represent a graph as a parameterized walk, and we establish a bound on the linearization length (Section 4). (3) We introduce an algorithm to parameterized-match the linearization on graph (Section 5). (4) We evaluate our approach experimentally (Section 6).

2 Preliminaries

This section defines the graph isomorphism problem and points out its similarity to parameterized matching in strings. In this paper, we consider multigraphs.

A multigraph $G(V, E)$ is comprised of a set of vertices V, $n = |V|$, and a set of undirected edges $E \subseteq V \times V$, $m = |E|$, where multiple edges between two distinct vertices and self loops are permitted. We distinguish the edges that have the same end vertices by the notation of the edge; for example, $e = (u, v)$ and $e' = (u, v)$. Let $\mathcal{E}_G = V \cup E$ denote the set of *graph elements* of G, *i.e.* the set of vertices and edges in G. Also, let *u.degree* denote the number of adjacent edges that vertex $u \in V$ has. In this paper, we consider undirected multigraphs; however our algorithms can be easily extended to support directed multigraphs. Next we define the *Graph Isomorphism* problem.

Problem 1 (Graph Isomorphism). Let $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$ be two multigraphs such that $n = |V_1| = |V_2|$ and $m = |E_1| = |E_2|$. The graph isomorphism problem determines whether there exists a bijective mapping function $f : \mathcal{E}_{G_1} \to \mathcal{E}_{G_2}$, such that $\forall_{u,v \in V_1}, e = (u, v) \in E_1 \iff f(u), f(v) \in V_2 \wedge f(e) = (f(u), f(v)) \in E_2$.



$V_1 \cup E_1$	A	B	C	D	E	e_1	e_2	e_3	e_4	e_5	e_6
f_1	X	Y	Z	W	S	e'_1	e'_2	e'_3	e'_4	e'_5	e'_6
f_2	X	Y	Z	W	S	e'_1	e'_2	e'_3	e'_5	e'_4	e'_6

(c)

Fig. 1. Isomorphism example: the multigraphs presented in (a) and (b) are isomorphic; the functions that define the isomorphism are presented in (c). The difference between f_1 and f_2 is that $f_1(e_4) = e'_4$ and $f_1(e_5) = e'_5$ while $f_2(e_4) = e'_5$ and $f_2(e_5) = e'_4$.

For example, the graphs in Figure 1(a,b) are isomorphic; furthermore there are two possible mapping functions that define the isomorphism (see Figure 1(c)). Notice that the graph isomorphism determines whether the topological structures of two multigraphs are the same. It is very similar to what parameterized matching does with strings: checking whether two strings have the same structure. Next we define parameterized matching on strings:

Definition 1. *Let $X = X_{1\ldots\ell}$ and $Y = Y_{1\ldots\ell}$ be two equal-length strings defined over alphabet Σ. Each symbol in the alphabet is called a parameter. Strings X and Y are said to parameterized-match iff there exists a bijective function $f : \Sigma \to \Sigma$ such that $f(X_i) = Y_i$, for all $1 \leq i \leq \ell$.*

For example, let $X = abacab$ and $Y = bcbabc$ be two strings defined over $\Sigma = \{a, b, c\}$. They parameterized-match as X is equal to Y by means of $f : (a, b, c) \to (b, c, a)$. In Section 3.1, we define parameterized matching for walks to solve the graph isomorphism problem.

3 Graph Linearization

Our approach for solving graph isomorphism consists of two main steps: (i) linearizing G_1 into a walk $p_{1\ldots\ell}$; and (ii) exploring all the walks in G_2 to determine whether there is one that parameterized matches $p_{1\ldots\ell}$. In this section, we define graph linearization and parameterized matching on graph walks (Section 3.1). Then, we discuss characteristics and algorithms for linearization (Section 3.2).

3.1 Definition of Graph Linearization

Definition 2 (Linearization). *Let $G(V, E)$ be a connected undirected multigraph. A walk $p = p_{1\ldots\ell}$ of vertices and edges is a linearization of G iff:*

1. *p_i is a vertex $v \in V$ if i is odd, $1 \leq i \leq \ell$.*
2. *p_i is an edge $e \in E$ if i is even, $1 \leq i \leq \ell$, such that $e = (p_{i-1}, p_{i+1})$.*
3. *Each vertex $v \in V$ and each edge $e \in E$ appears at least once in p.*

Our motivation for defining graph linearization is to represent the topology of a multigraph through a walk. Specifically, the linearization p of G is a walk that represents all its adjacency relation, which we use to solve the graph isomorphism problem by comparing walks instead of multigraphs. For this purpose, we define parameterized matching on walks as follows:

Definition 3 (Parameterized Matching on Graph Walks). *Let $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$ be two connected undirected multigraphs. Also, let $V_1' \subseteq V_1$ and $E_1' \subseteq E_1$ be subsets of vertices and edges in G_1; similarly, $V_2' \subseteq V_2$ and $E_2' \subseteq E_2$ are subsets of vertices and edges in G_2. Consider the walk $p_{1\ldots k}$ in G_1 and the walk $q_{1\ldots k}$ in G_2. The walk $p_{1\ldots k}$ is said to parameterized-match the walk $q_{1\ldots k}$ if and only if there exists a bijective function $f : \mathcal{E}_{G_1} \to \mathcal{E}_{G_2}$ such that $q_i = f(p_i)$ for $1 \leq i \leq k$.*

The core idea of using parameterized matching to solve the graph isomorphism problem is as follows. Let p be a linearization of G_1. Recall that, p represents the topology of G_1. Thus, if a walk q in G_2 parameterized-matches p, then p and q have the same topology. Furthermore, as q represents G_2, we conclude that G_1 and G_2 are isomorphic. This is formally presented in the next theorem:

Theorem 1. *Let $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$ be two connected undirected multigraphs such that $n = |V_1| = |V_2|$ and $m = |E_1| = |E_2|$. Also, let $p_{1\ldots\ell}$ be the linearization of G_1. Then, G_1 and G_2 are isomorphic if and only if there exists a walk $q_{1\ldots\ell}$ in G_2 such that $p_{1\ldots\ell}$ parameterized-matches $q_{1\ldots\ell}$.*

3.2 Characteristics and Algorithms for Graph Linearization

There may be many linearizations that represent the same graph. However, a compact representation is preferable. For solving graph isomorphism, the length of the linearization is an important measure on the matching time. This is because a shorter linearization often leads to a smaller cost at the matching stage. Next, we define *length-optimal linearization*.

Definition 4 (Length-Optimal Linearization). *The linearization $p = p_{1...\ell}$ of a connected undirected multigraph is length-optimal if the length of p,* i.e. *ℓ, is minimum.*

The *Graph Linearization* problem is very similar to the *Chinese Postman Problem* (CPP). CPP finds a walk that visits all the edges (and all the vertices) in the multigraph at least once; the only difference is that Graph Linearization does not require the starting vertex to be the same final vertex. In [11], an $O(n^3 + m^2)$ algorithm for the CPP was proposed. We can adapt this algorithm to calculate a length-optimal linearization. However, for large multigraphs, it is desirable to have algorithms with lower time complexity even if they do not produce length-optimal linearizations. As an attractive trade-off between length-optimality and efficiency, we propose a greedy approximation algorithm with an approximation guarantee.

4 Graph Linearization Algorithm - GLA

This section presents the GLA or *Graph Linearization Algorithm*. First, we describe the key ideas of the algorithm in Section 4.1; then we go through the details in Section 4.2. In Section 4.3 we present an upper bound for the length of GLA linearizations. Finally, in Section 4.4, we present the complexity analysis.

4.1 Key Ideas

One of the challenges of linearization algorithms is visiting all the edges with short linearization length. To address the challenge, we develop three heuristics: (1) the traversal starts from the vertex with the lowest degree; (2) the unexplored edges that lead to already explored vertices are visited before the ones that lead to unexplored vertices; and (3) the edges that lead to unexplored vertices are considered sorted, in ascending order, on the number of unexplored edges they have. Heuristics (1) and (3) aim to put the vertices that are close to be covered in the top levels of the DFS tree. Furthermore, heuristic (2) aims to cover the vertices in the highest levels of the DFS tree at an early stage. The three heuristics make the traversal explore one region of the multigraph before visiting another one; then, the produced linearization is shorter.

The proposed linearization approach also allows us to incorporate optimizations for both linearization and parameterized matching steps. For instance, the matching time will not only depend on the length of the linearization, but also

on the order of comparisons. Specifically, the graph statistics of the multigraphs can be used to produce a linearization that prunes the search space during the matching phase. For example, if the frequency of some vertices of a certain degree (or a certain attribute in attributed graphs) is low, it would be appropriate to start the linearization from such vertices. However, for clarity, in this paper, we focus on the fundamental approach only.

4.2 Algorithm

The pseudocode of the Graph Linearization Algorithm (GLA) is listed in Figures 2 and 3. The linearization produced by GLA for the graph presented in Figure 1(a) is $Ae_1Be_3Ce_4De_5Ce_5De_2\ Be_2De_6E$; its length is 17.

Algorithm 1: GLA **Algorithm**

Input: $G(V, E)$ **Output**: p

1. **for every** $e \in E$ **do** $e.Explored \leftarrow false$
2. **for every** $v \in V$ **do**
3. $\quad v.Explored \leftarrow false$
4. $\quad \mathcal{S} \leftarrow \{(u, v) \mid v \in V \wedge (u, v) \in E\}$
5. $\quad v.NumUnexploredEdges \leftarrow |\mathcal{S}|$
6. **choose** $u \in V_P$ **with** $min(u.NumUnexploredEdges)$
7. $p \leftarrow \langle\rangle$, $unexplGE \leftarrow |V| + |E|$
8. $TraverseGraph(G, u, p, unexplGE)$
9. **return** p

Fig. 2. GLA Algorithm.

4.3 Length of GLA Linearization

Theorem 2 shows that given the multigraph $G = (V, E)$, the length of the walk generated by GLA is at most 2 times the length of an optimal linearization. Therefore, the length produced by GLA is asympotically optimal.

Theorem 2. GLA *is* 2*-approximate with respect to the length of the length-optimal linearization.*

This theorem is based on the fact that each edge in the multigraph G appears at most twice in the linearization $p = p_{1\ldots\ell}$ generated by GLA. Then, ℓ is compared to a lower bound that visits each edge only once to show worst-case approximation ratio. However, even an optimal linearization may not achieve the lower bound for many graph structures. Thus, for average cases in practice, GLA linearization is much closer to the optimal.

Algorithm 2: TraverseGraph **Procedure**

Input: $G(V, E), u, p, unexplGE$

```
1.  p.Add(u), u.Explored ← true, unexplGE--
2.  for every e ∈ E such that e = (u, v) do
3.      if !e.Explored ∧ v.Explored then
4.          p.Add(e), e.Explored ← true, unexplGE--, p.Add(v)
5.          u.NumUnexplEdges--, v.NumUnexplEdges--
6.          if unexplGE > 0 do
7.              p.Add(e), p.Add(u)
8.  while there are unexplored edges e = (u, v)
9.      choose e with min(v.NumUnexploredEdges)
10.     p.Add(e), e.Explored ← true, unexplGE--
11.     u.NumUnexplEdges--, v.NumUnexplEdges--
12.     TraverseGraph(G, v, p, unexplGE)
13.     if unexplGE = 0 then break
14.     p.Add(e), p.Add(u)
```

Fig. 3. TraverseGraph Procedure.

4.4 Complexity Analysis

The complexity of GLA is dominated by the walk traversed (line 8, Figure 2) which corresponds to the linearization. Notice that p has at most $2m$ edges and $2m + 1$ vertices. Each insertion takes constant time as it is always done at the end of p. But when a vertex is inserted for the first time, it is necessary to consider the unexplored adjacent edges e that lead to unexplored vertices v sorted on $v.NumUnexplEdges$ (lines $8 - 9$, Figure 3). This sorting operation takes $O(d \lg d)$, where d is the maximum degree of the vertices in G_1; specifically $d = \max_{v \in V_1} v.degree$. Thus, the time complexity of GLA is $O(2m + (2m + 1)(d \lg d)) = O(dm \lg d)$.

5 Matching a Linearized Graph

The *Parameterized Matching on multi-Graphs* (PMG) algorithm uses a linearization of $G_1(V_1, E_1)$, denoted as $p = p_{1 \ldots \ell}$, and matches it against $G_2(V_2, E_2)$ to determine whether G_1 and G_2 are isomorphic by using Theorem 1.

5.1 Key Ideas

PMG considers all the possible injective functions $f : \mathcal{E}_{G_1} \rightarrow \mathcal{E}_{G_2}$ to determine whether there is mapping with two properties: (i) f is bijective; and (ii) there exists a walk $q_{1 \ldots \ell}$ in G_2 for which $q_i = f(p_i)$ (*i.e.* q parameterized-matches p). These possible injective functions are explored by traversing p and G_2 simultaneously; specifically, a graph element p_i is compared to a graph element ge in G_2 to determine whether an injective mapping is possible. We progressively

extend a successful mapping by considering p_{i+1} and an adjacent graph element of ge. The graph elements of G_2 are traversed in a depth-first manner while p is traversed from left to right. Let us consider the DFS tree that represents the traversal of G_2. Then, the idea of this traversal of G_2 is considering the possible injective mappings by attempting to set $f(p_i) = ge$ where $ge \in \mathcal{E}_{G_2}$ is a graph element at level i of the DFS tree. Notice that the walk from the root to a leaf in the DFS tree parameterized-matches $p_{1\ldots\ell}$ under f; hence G_1 and G_2 are isomorphic.

Next, we show our heuristics to prune the search space. At each step of the process, a vertex $u \in V_2$ and a vertex in p_i are compared. Let us say that we set $f(p_i) = u$. In order to extend the match, we use vertex degrees and previous assignments in f to prune the search space. Specifically, we consider two cases:

Case 1: Vertex p_{i+2} is unassigned. We consider all the possible assignments $f(p_{i+1}) = e$ and $f(p_{i+2}) = v$ for edges $e = (u, v) \in E_2$ such that: (i) both e and v are unassigned; and (ii) $v.degree = p_{i+2}.degree$. Condition (i) is to guarantee that f is injective; condition (ii) is a pruning criterion based on that fact that, if G_1 and G_2 are isomorphic, then analogous vertices must have the same degree. Notice that if p_{i+2} is unassigned, p_{i+1} is unassigned as well; this is because the assignment of an edge in p is done at the same time (or after) the assignment of its end vertices. The process continues by considering p_{i+2} and each v.

Case 2: Vertex p_{i+2} is assigned to $v \in V_2$. There are two sub-cases. (a) Edge p_{i+1} is already assigned: it is not necessary to check adjacency as this was done when the mapping was set. We continue by considering p_{i+2} and v. (b) Edge p_{i+1} is unassigned: the algorithm considers all the possible assignments $f(p_{i+1}) = e$ for the unassigned edges $e = (u, v)$. The process continues at p_{i+2} and v.

If the algorithm reaches a successful assignment for p_ℓ, then the algorithm reports that the multigraphs are isomorphic.

5.2 Pseudocode

Figure 4 lists the pseudocode of PMG. The mapping function is represented as the array f. On the other hand, boolean array g indicates if each graph element in $\mathcal{E}_{G_2}$ is already assigned to a graph element in $\mathcal{E}_{G_1}$ (through function f). When we run PMG for G_2 and the linearization $p = Ae_1Be_3Ce_4De_5Ce_5De_2Be_2De_6E$ of G_1, the match is returned when any of the following walks are traversed: $q_1 = Xe'_1Ye'_3Ze'_4We'_5Ze'_5We'_2Ye'_2We'_6S$ or $q_2 = Xe'_1Ye'_3Ze'_5We'_4Ze'_4We'_2Ye'_2We'_6S$. Notice that both q_1 and q_2 parameterized-match p. The mapping functions of these matches correspond to the functions f_1 and f_2 presented in Figure 1(c).

5.3 Complexity Analysis

The time complexity of PMG is given by the number of executions of the recursive procedure ExtendMatch; each execution requires constant time. This number is equal to the number of vertices and edges in the DFS search trees. As the number of edges in a DFS tree is equivalent to the number of vertices — each vertex, except the root, is associated to an edge that leads to its parent,

Algorithm 3: PMG **Algorithm**

Input: $G_1(V_1, E_1), G_2(V_2, E_2)$ **Output**: $true/false$

```
1.  p = GLA(G_1)
2.  for every ge ∈ (V_1 ∪ E_1) do f[ge] ← undef
3.  for every ge ∈ (V_2 ∪ E_2) do g[ge] ← false
4.  for every u ∈ V_2 do
5.      if u.degree = p_1.degree
6.          f' ← copyOf(f),  f'[p_1] ← u
7.          g' ← copyOf(g),  g'[u] ← true
8.          if ExtendMatch(u, p, 1, f', g', G_2) = true
9.              return true
10. return false
```

Fig. 4. PMG Algorithm.

Algorithm 4: ExtendMatch **Algorithm**

Input: $u, p = p_{1\ldots\ell}, i, f, g, G_2(V_2, E_2)$ **Output**: $true/false$

```
1.  if i = ℓ then return true
2.  if f[p_{i+2}] = undef
3.      for every e = (u, v) ∈ E_2 do
4.          if g[v] = false and g[e] = false and v.degree = p_{i+2}.degree
5.              f' ← copyOf(f),  f'[p_{i+1}] ← e,  f'[p_{i+2}] ← v
6.              g' ← copyOf(g),  g'[e] ← true,  g'[v] ← true
7.              if ExtendMatch(v, p, i + 2, f', g', G_2) = true
8.                  return true
9.  else
10.     v = f[p_{i+2}]
11.     if  p_{i+1} = undef
12.         for every  e = (u, v) ∈ E_2 such that g[e] = false
13.             f' ← copyOf(f),  f'[p_{i+1}] ← e
14.             g' ← copyOf(g),  g'[e] ← true
15.             if ExtendMatch(v, p, i + 2, f', g', G_2) = true
16.                 return true
17.     else
18.         if ExtendMatch(v, p, i + 2, f, g, G_2) = true
19.             return true
20. return false
```

Fig. 5. ExtendMatch Algorithm.

the asymptotic behavior of PMG depends on the number of vertices in the DFS trees. Next theorem gives an upper bound for this number.

Theorem 3. *Let $p_{1\ldots\ell}$ be a linearization of G_1. Also, let d be the maximum degree of the vertices in G_2; specifically $d = \max_{v \in V_1} v.degree$. The DFS tree that represents the traversal of G_2 done by* PMG *has at most $O(d^{\lfloor \ell/2 \rfloor})$ vertices.*

This theorem is based on the following facts: (i) there are $O(\lfloor \ell/2 \rfloor)$ branching vertices in the DFS search tree associated to the vertices in the linearization $p = p_{1\ldots\ell}$; and (ii) the branching factor for each of such vertices in the search tree is $O(d)$. As a DFS tree starts at each vertex in G_2, the total number of vertices visited, and hence the time complexity of PMG, is $O(nd^{\lfloor \ell/2 \rfloor})$. Note that if G_2 is complete, *i.e.*, $d = n-1$, the time complexity is $O(n(n-1)^{\lfloor \ell/2 \rfloor}) = O(n^{\lceil \ell/2 \rceil})$.

However, it is important to remark that Theorem 3 gives an upper bound for the worst-case complexity. It assumes that, at every level of vertices, all the possible neighbors are explored. The average-case situations in practice are often not that "bad" because (i) when a vertex p_i has already been assigned, only such assigned vertex is considered; and (ii) when the multigraph has varied vertex degrees, the pruning criterion highly reduces the number of adjacent vertex to be visited.

6 Experimental Evalution

We assess the performance of our proposed approach experimentally. We implement the linearization and the matching algorithms in C#. We employ a set of synthetic graphs generated for benchmarking. We compare our approach to VF2, using an optimized implementation from the networkX library [1]. All evaluations are performed on a server running under a Windows platform on a 3.40GHz CPU with 16GB memory.

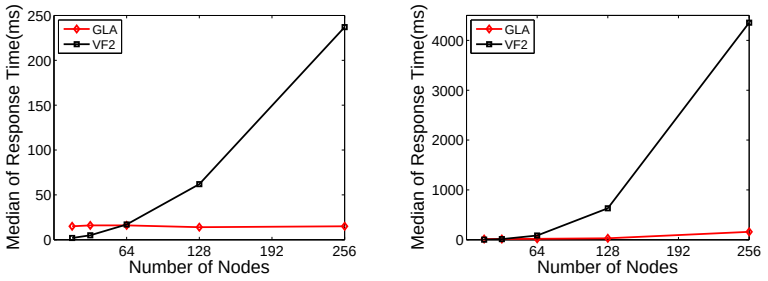
For graph generation, we deliberately avoid the "trivial cases". For example, consider a graph where vertex v_i is connected to $v_1, \ldots, v_{i-1}$. As the degree of each node is unique, testing isomorphism can be done trivially by a simple heuristic like sorting nodes by degree. In contrast, we consider cases where no such simple heuristic wins. Graphs where every node has the identical degree would much more challenging in that sense.

Meanwhile, we also avoid topologies that are always isomorphic, such as a grid or a complete graph. For this reason, we generate random graph pairs of identical-degree nodes. As the complexity of isomorphism testing algorithm is reported to vary significantly over degree, from $O(n^2)$ to $O(nn!)$ [9], we consider both low- and high-degree cases to evaluate algorithms in a wide spectrum of settings. The lower end of this spectrum is observed when the matching graphs are found early in a sparse graph, while the opposite case of dense graphs often leads to long running times. More specifically, we generate sparse and dense identical-degree graphs as follow: **1-Sparse:** We generate a random graph G where every node has degree three, with $3N$ total edges for N nodes. We first build a random binary tree with $N-1$ edges. Then, the nodes with the degree

[1] http://networkx.github.io

less than three connect to another such node chosen at random. **2-Dense:** We generate graph G' by subtracting G from a complete graph. Every node of G has the same degree of $N - 4$.

In each setting, we vary the number of nodes from 16 to 256, and evaluate the response time of GLA (our proposed approach) and VF2 (baseline). For each point in the figures, we randomly generate 45 graphs and report the median response time. We choose median response time as our performance metric because the running time on different graphs significantly varies over graph complexity (as discussed above) while the optimization margin is narrow for easy cases and hard extremes. Our target problems are thus neither of these, and using the average or min/max as the main performance metric would bias the results to represent either extreme. In contrast, median would filter out extreme results.



(a) *Sparse* median response time. (b) *Dense* median response time.

Fig. 6. Response time of GLA and VF2 on sparse and dense graphs.

Figure 6(a) and (b) show the results for *Sparse* and *Dense* respectively. The X-axis is the number of nodes (in log scale) and Y-axis is the median response time in milliseconds. Note the two graphs have different scales, and number of edges is linear with the number of nodes for sparse graphs and quadratic for dense graphs. In Figure 6(a), the median running time of GLA remains more or less constant to 10 milliseconds, despite the increase in graph size. As a result, when $v = 256$, GLA outperforms VF2 by an order of magnitude. In Figure 6(b), we observe a consistent trend, except that the performance gap is larger. In particular, for $N = 256$, GLA is faster by two orders of magnitude. These figures show that GLA has low response time, shorter than VF2, by effectively pruning the notoriously large search space, guided by linearization rules leveraging node degree and exploration history.

7 Conclusions

This paper presents a novel approach to solve graph isomorphism. The key idea is to linearize one graph into a parameterized sequence — a walk that covers

every node and edge — and parameterized-match the linearization on the second graph. We develop a fast linearization algorithm that produces a short linearization, and a parameterized matching algorithm. We implement the algorithms and evaluate them experimentally against VF2, and observe lower response times for sparse and dense graphs with varying sizes.

References

1. Amir, A., Aumann, Y., Cole, R., Lewenstein, M., Porat, E.: Function matching: Algorithms, applications, and a lower bound. In: Proc. 30th International Colloquium on Automata, Languages and Programming (2003)
2. Amir, A., Farach, M., Muthukrishnan, S.: Alphabet dependence in parameterized matching. Information Processing Letters 49(3), 111–115 (1994)
3. Apostolico, A., Giancarlo, R.: Periodicity and repetitions in parameterized strings. Discrete Applied Mathematics 156(9), 1389–1398 (2008)
4. Baker, B.: A theory of parameterized pattern matching: algorithms and applications. In: Proc. 25th Annual Symposium on Theory of Computing (1993)
5. Baker, B.: Parameterized pattern matching by Boyer-Moore-type algorithms. In: Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms. p. 550. Society for Industrial and Applied Mathematics (1995)
6. Baker, B.: Parameterized pattern matching: algorithms and applications. J. Comput. Syst. Sci. 52(1), 28–42 (1996)
7. Baker, B.: Parameterized duplication in strings: algorithms and an application to softwaremaintenance. SIAM Journal on Computing 26(5), 1343–1362 (1997)
8. Conte, D., Foggia, P., Sansone, C., Vento, M.: Thirty years of graph matching in pattern recognition. International journal of pattern recognition and artificial intelligence 18(03), 265–298 (2004)
9. Cordella, L.P., Foggia, P., Sansone, C., Vento, M.: A (sub) graph isomorphism algorithm for matching large graphs. Pattern Analysis and Machine Intelligence, IEEE Transactions on 26(10), 1367–1372 (2004)
10. Du Mouza, C., Rigaux, P., Scholl, M.: Parameterized pattern queries. Data & Knowledge Engineering 63(2), 433–456 (2007)
11. Edmonds, J., Johnson, E.L.: Matching, euler tours and the chinese postman. Mathematical programming 5(1), 88–124 (1973)
12. Fredriksson, K., Mozgovoy, M.: Efficient parameterized string matching. Information Processing Letters 100(3), 91–96 (2006)
13. Hazay, C.: Parameterized matching. Master's thesis, Bar-Ilan University (2004)
14. Hazay, C., Lewenstein, M., Sokol, D.: Approximate parameterized matching. ACM Transactions on Algorithms 3(3), 29 (2007)
15. Hazay, C., Lewenstein, M., Tsur, D.: Two dimensional parameterized matching. In: CPM. pp. 266–279 (2005)
16. Kosaraju, S.: Faster algorithms for the construction of parameterized suffix trees. In: Proceedings of the 36th Annual Symposium on Foundations of Computer Science. IEEE Computer Society Washington, DC, USA (1995)
17. Lee, I., Mendivelso, J., Pinzón, Y.J.: $\delta\gamma$—parameterized matching. In: Proceedings of the 15th International Symposium on String Processing and Information Retrieval. pp. 236–248. Springer-Verlag (2008)
18. Lewenstein, M.: Parameterized matching. In: Encyclopedia of Algorithms. Springer (2008)
19. McKay, B.D.: Practical graph isomorphism. Congressus Numerantium 30, 45 (1981)
20. Mendivelso, J., Lee, I., Pinzón, Y.: Approximate function matching under δ-and γ-distances. In: String Processing and Information Retrieval. pp. 348–359. Springer (2012)
21. Salmela, L., Tarhio, J.: Sublinear algorithms for parameterized matching. In: Proc. 17th Annual Symposium on Combinatorial Pattern Matching (2006)
22. Ullmann, J.R.: An algorithm for subgraph isomorphism. Journal of the ACM (JACM) 23(1), 31–42 (1976)