Brief Announcement: A Parallel Algorithm for Subgraph Isomorphism

Rohan Yadav
rohany@andrew.cmu.edu
Carnegie Mellon University

Umut A. Acar
umut@cs.cmu.edu
Carnegie Mellon University

Abstract

Subgraph isomorphism is a fundamental property of graphs that requires checking whether the network structure of one graph can be found (embedded) within another graph. It has numerous applications and is a computationally challenging problem: it is NP-complete and known algorithms explore an exponentially large search space. Even though it has been studied extensively, relatively little is known about whether subgraph isomorphism accepts a theoretically and practically efficient parallel solution. In this paper, we present our ongoing work on designing a parallel algorithm for the subgraph (and graph) isomorphism problem, which addresses challenges commonly faced when attempting to obtain a parallel algorithm for isomorphism. Our algorithm appears to scale well up to the 70 cores of our empirical machine.

CCS Concepts

• **Mathematics of computing → Graph algorithms**; • **Theory of computation → Graph algorithms analysis**; • **Computing methodologies → Shared memory algorithms**.

Keywords

parallel computing, algorithms, graphs

ACM Reference Format:
Rohan Yadav and Umut A. Acar. 2019. Brief Announcement: A Parallel Algorithm for Subgraph Isomorphism. In *31st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '19), June 22–24, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3323165.3323170

1 Introduction

Graphs, a fundamental abstraction in computer science, have been studied extensively. Graph and subgraph isomorphism, two variants of an important structural equivalence property of graphs, have likewise received much attention from researchers. Graph isomorphism requires checking whether a graph G_1 encodes the same network structure as another graph G_2; subgraph isomorphism requires checking whether G_1 encodes the same structure as a subgraph of G_2. The subgraph-isomorphism problem is known to be NP Complete, whereas graph isomorphism has recently been shown to be solvable in quasi-polynomial time by Babai [3]. Even though the problems can be considered intractable in their general forms, there has been much work on developing practical algorithms, going back to 1970s and Ullman's algorithm [11].

Given the exponential work requirements of subgraph isomorphism algorithms, a natural question is whether parallelism can be used to improve run time. One difficulty in giving a parallel algorithm for subgraph isomorphism is that efficient subgraph isomorphism algorithms such as VF2 [9] rely on a Depth-First Search (DFS) of the search space of candidate solutions and heavily exploit the sequential nature of DFS. When using a sequential DFS, the backtracking operation is trivial. One key challenge is therefore implementing the backtracking algorithm needed to recover from failed branches while performing the search in parallel. Another challenge is the highly irregular nature of the search itself: any search state could lead to many other states, to few other states, or to none at all. This leads to the classic granularity problem of whether to run some branches of the search in parallel or not.

We describe a possible approach to designing a parallel algorithm for graph and subgraph isomorphism. The key ideas behind our algorithm are 1) to structure the search in such a way that the cost of backtracking remains as efficient as in the sequential setting, and 2) to perform scheduling lazily and amortize the cost scheduling against the total work. One technical observation behind our algorithm is that the search space of the algorithm can be represented by a ***frontier*** data structure that organizes the space in such a way that processor-local operations can be performed efficiently while also allowing the search space to be evenly split and shared with other processors.

2 Algorithm

Graph Matching. A ***matching*** M from directed graphs $G_1 = (V_1, E_1)$ to $G_2 = (V_2, E_2)$ is a partial bijection from V_1 to V_2. In other words, for any $u \in V_1$, there exists at most one $(u, v) \in M$ and there exists no $(w, v) \in M$ for $w \neq u$. We refer to each member of a matching, which is of the form (u, v) as a ***cognate***. We say that a matching is ***adjacency preserving*** if for any $u, v \in \text{dom}(M)$, if $(u, v) \in E_1$, then $(M(u), M(v)) \in E_2$. A matching M from G_1 to G_2 is a ***subgraph isomorphism*** if its domain is V_1, $\text{dom}(M) = V_1$, and it is adjacency preserving. Our algorithm constructs isomorphisms by generating online a matching tree and searching through this tree. Consider two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ such that the vertices of V_1 (V_2) are totally ordered. We define a ***matching tree*** from G_1 to G_2 as a tree with exactly $|V_1| + 1$ levels, $0 \ldots |V_1|$. The nodes in the tree represent matchings from G_1 to G_2, and the root node represents an empty matching. An edge from a node at level i to level $i + 1$ in the matching tree corresponds to adding a cognate for the corresponding node at level i. A cognate is a pair of vertices (u, v), where $u \in V_1$ and $v \in V_2$. Each level of the tree matches a single vertex of V_1 to a vertex in V_2. We will also refer to

SPAA '19, June 22–24, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6184-2/19/06.
https://doi.org/10.1145/3323165.3323170

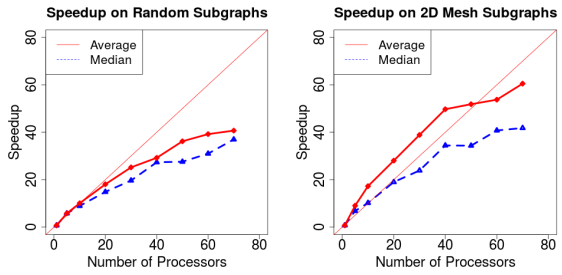

Figure 1: Subgraph Isomorphism Speedup

level as depth in the tree. The leaves of the matching tree include all possible subgraph isomorphisms from G_1 to G_2, but some leaves are not adjacency preserving.

The Algorithm. Our algorithm uses a ***matching*** data structure, which represents a node in the matching tree, consisting of a set of ***cognates***, each of which is labeled the depth at which the cognate is added into the matching.To represent the "work" that must be done by the algorithm, we use a ***frontier*** data structure that keeps a set of edges from the matching tree, sorted by the depth of each edge's parent node in the matching tree. We represent edges from a parent node to child node by the cognate that must be added to the parent to generate the child, and tag the cognate with its depth. The frontier data structure supports push/pop operations, which respectively add/remove a largest element to/from the frontier. Crucially, the frontier supports a split operation, which splits it into a prefix and a suffix such that the depths of the cognates in the prefix are less or equal to those in the suffix.

Our algorithm uses a lazy approach to generating parallelism. We assume a work-stealing based scheduler [1], where each worker mimics the execution of the sequential algorithm when running locally, but is always able to generate parallel work when needed. Every worker maintains a current matching M and frontier F, which are constructed online. If the scheduler requests a worker to share its work, the worker splits its frontier into a prefix and suffix, and returns to the scheduler the prefix and a copy of its matching. If the scheduler requests a worker to perform computation, the worker pop's the deepest cognate (u, v, d) off of its frontier. The worker backtracks the current matching to the depth d by removing all cognates from the matching that have a depth greater than d. Why this operation correctly performs an arbitrary backtrack from failed parallel searches is subtle and requires proof, explored in the full paper. Next, the worker checks to make sure that the addition of this cognate will maintain the adjacency preserving property of the matching. If it does not, the worker returns with an unchanged matching, and frontier with this cognate removed. If the addition of the cognate causes the new matching to be adjacency preserving, then the worker adds all children nodes of the new matching in the matching tree to the frontier.

Implementation and Experiments. We implemented our algorithm by building on publicly available pieces of software [1, 2, 9]. We have conducted a preliminary evaluation of our implementation on a data set called the "ARG" database, specifically designed for benchmarking isomorphism [6]. To compare our algorithm to the VF2 algorithm, we calculate speedup as we calculate speedup as $\frac{T_s}{T_P}$, where T_s is the time taken by the VF2 algorithm on a single core, and T_P is the time taken by our algorithm on P processors. Figure 1 shows the average and median speedups for two different experiments from the ARG database. These preliminary results show that our algorithm is able to achieve good speedup over the VF2 algorithm.

3 Related Work

There are many serial algorithms for the graph and subgraph isomorphism problem, including constraint programming [10], canonical labeling [8], and tree-search based algorithms [12]. Our work is related to tree-search based algorithms. Recent work in these areas develop the heuristics and pruning strategies used during the tree-search, which is orthogonal to this work. We expect that the core ideas of our algorithm can be applied to newer algorithms such as VF3 [12].

There has been relatively little work on the harder problem of parallel subgraph isomorphism. Work by Blankstein et al [4] present a parallel search based algorithm, and encounter the challenges discussed in the introduction. The authors propose several heuristics for overcoming them, but they note that these heuristics don't always work, leading to unpredictable perfomance and work inefficiency. McCreesh et al [5] present a constraint based search algorithm for undirected graphs, which explores a search tree in a parallel depth first fashion. A common problem in analyzing parallel search algorithms is the fact that the search space explored by the parallel algorithm can differ greatly from the one explored by the sequential algorithm due to the speculation in parallel search [7]. Analysis of our parallel algorithm in this context, along with its use of lazy scheduling are avenues to be explored in future work.

References

[1] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. Scheduling parallel programs by work stealing with private deques. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, 2013.

[2] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. *Theory and Practice of Chunked Sequences*, pages 25–36. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[3] László Babai. Graph isomorphism in quasipolynomial time [extended abstract]. In *Proceedings of the Forty-eighth Annual ACM Symposium on Theory of Computing*, STOC '16, pages 684–697, New York, NY, USA, 2016. ACM.

[4] Aaron Blankstein and Matthew Goldstein. Parallel subgraph isomorphism, 2010. Unpublished Manuscript.

[5] McCreesh C. and Prosser P. A parallel, backjumping subgraph isomorphism algorithm using supplemental graphs. *Principles and Practice of Constraint Programming*, 2015.

[6] P. Foggia, C. Sansone, and M. Vento. A database of graphs for isomorphism and sub-graph isomorphism benchmarking. 2001.

[7] Benjamin W. Wah Guo-Jie Li. How to cope with anomalies in parallel approximate branch-and-bound algorithms. 1984.

[8] Brendan D. Mckay and Adolfo Piperno. Practical graph isomorphism, ii. *J. Symb. Comput.*, 60:94–112, January 2014.

[9] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(10):1367–1372, October 2004.

[10] Christine Solnon. Alldifferent-based filtering for subgraph isomorphism. *Artif. Intell.*, 174(12-13):850–864, August 2010.

[11] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, January 1976.

[12] Carletti Vincenzo, Pasquale Foggia, Alessia Saggese, and Mario Vento. Introducing vf3: A new algorithm for subgraph isomorphism. pages 128–139, 05 2017.