Context-Free Path Queries, Planar Graphs and Friends

Alexandra Istomina
Saint-Petersburg State University
Saint-Petersburg, Russia

Alexandra Olemskaya
Inria Paris-Rocquencourt
Rocquencourt, France

Ekaterina Shemetova
Inria Paris-Rocquencourt
Rocquencourt, France

Semyon Grigorev
Rajiv Gandhi University
Doimukh, Arunachal Pradesh, India

ABSTRACT

Abstract is very abstract.

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

KEYWORDS

datasets, neural networks, gaze detection, text tagging

ACM Reference Format:
Alexandra Istomina, Alexandra Olemskaya, Ekaterina Shemetova, and Semyon Grigorev. 2018. Context-Free Path Queries, Planar Graphs and Friends. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Context-Free Path Querying (CFPQ) is a sublcass of Language-constrained path problem, where language is set to be Context-Free.

Importance of CFPQ. Application areas. RDF, Graph database querying, Graph Segmentation in Data provenance, Biological data analysis, static code analysis.

1.1 An Example

Example of graph and query. Should be used in explataion below.

1.2 Existing CFPQ Algorithms

Number of problem-specific solutions in static code analysis.

Hellings, Ciro et al, Kujpers, Sevon, Verbitskaya, Azimov, Ragozina

1.3 Existing Theoretical Results

Existing theoretical results
Linear input. Valiant [**?**], Lee [**?**].
Yannacacis [**?**]? Reps?
Bradford [**?**]
RSM [**?**].
C alias analysis [**?**]
Chatterjee [**?**]
For trees
Truly-subcubic for Language Editing Distance [**?**].
Truly-subcubic algorithm is stil an open problem.

1.4 Our Contribution

This paper is organized as follows. !!!!!

2 PRELIMINARIES

We introduce !!!!

2.1 Context-Free Path Querying

Graph, grammar, etc.

Let $i\pi j$ denote a unique path between nodes i and j of the graph and $l(\pi)$ denotes a unique string which is obtained from the concatenation of edge labels along the path π. For a context-free grammar $G = (\Sigma, N, P, S)$ and directed labelled graph $D = (Q, \Sigma, \delta)$, a triple (A, i, j) is *realizable* iff there is a path $i\pi j$ such that nonterminal $A \in N$ derives $l(\pi)$.

2.2 Tensor-Based algorithm for CFPQ

Algorithm 1 Kronecker product context-free recognizer for graphs

1: **function** contextFreePathQuerying(D, G)

2.3 Planar Graphs

A planar graph $G = (V, E)$ is a graph that can be embedded in the plane.

Outer face - unbounded face in specific embedding.
Directed graph (*digraph*)
...

2.4 Dynamic reachability algorithms

We consider algorithms that solve the problem of reachability in planar directed graphs. In the *dynamic reachability problem* we are given a graph G subject to edge updates (insertions or deletions) and the goal is to design a data structure that would allow answering queries about the existence of a path.

We need to answer the queries of type: "Is there a directed path from u to v in G?". If vertex u in all the queries is fixed we say that algorithm is *single-source*. It is said to be *all-pairs* if vertices u, v can

Woodstock '18, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

be any vertices of planar digraph G, in this case it can be also called *dynamic transitive closure*.

We say that the algorithm is *fully dynamic* if it supports both additions and deletions of edges. It is said to be *semi dynamic* if it supports only one of these updates. If semi dynamic algorithm supports additions only it is called *incremental*, if deletions only - *decremental*.

3 CFPQ ON PLANAR DIGRAPHS

We want to consider algorithms for semi-dynamic transitive closure for the case of a planar graph. Our algorithm needs to support only edge insertions that do not violate planarity.

There are several algorithms that look into the problem of dynamic reachability in planar digraphs and have sublinear both update and query time [**?**], [**?**], [**?**].

Some algorithms [**?**], [**?**] restrict insertions to only that edges that respect the specific embedding of the planar graph.

maybe we can choose one embedding at random and hope that nothing will violate it. So it will be monte-carlo (why it will have good probability? where from can we get all the possible non-isomorphic embeddings?)

The algorithms that allow edge insertions not with respect to the embedding of the graph, but to the planarity, are described in [**?**], [**?**]. The main idea of both articles is the same: a planar graph is separated into *clusters* (edge induced subgraphs) for which reachability is shown by their sparse substitute S. *Sparse substitute* is a graph that contains the reachability information between some set of vertices that lie in the same face of an embedded graph.

We show the results of [**?**] more closely as they differ from the results of [**?**] only by a logarithmic factor, have the same idea and are easier for understanding.

Updates and queries are the following: adding or deleting the edge between vertices u, v, checking reachability, checking if new edge (u, v) violates planarity.

Every update procedure modifies a constant number of clusters, for which we reconstruct their sparse substitutes. If the procedure added the edge, new edge forms its own cluster. Moreover after every $O(n^{1/3})$ insertions we rebuild cluster partition and sparse substitutes from the scratch so that clusters do not lose their properties, this rebuilding time is distributed between these $O(n^{1/3})$ insertions, so insertion time is amortized.

After splitting graph into clusters and looking into its sparse substitute update or query about vertices u, v can be done by placing clusters of the vertices u, v in the original graph G into the graph of sparse substitutes S.

For maintaining dynamic transitive closure in this way we only need planarity to divide graph G into clusters and for each cluster to build its sparse substitute. As we rebuild cluster partition and substitutes at the beginning and after every $O(n^{\frac{1}{3}})$ edge-insertions, graph needs to remain planar during the updates.

The results are the following.

(1) The amount of time for preprocessing is $O(n \log n)$ — in this time we find the separation of the given graph G into $O(n^{\frac{1}{3}})$ clusters (each consists of no more than $n^{\frac{2}{3}}$ edges of G) and build sparse substitution graph S of total size $O(n^{\frac{2}{3}} \log n)$ for them.
(2) After that we can perform add operation in $O(n^{\frac{2}{3}} \log n)$ amortized time, delete operation in $O(n^{\frac{2}{3}} \log n)$ worst-case time.
(3) Reachability query takes $O(n^{\frac{2}{3}} \log n)$ worst-case time.
(4) Checking if the graph is planar can be done in $O(n^{\frac{1}{2}})$ amortized time.

Let us look closely on how can we use the power of sparse substitution on our incremental transitive closure problem.

Proposition 3.1. *We can maintain the structure described in [**?**] not only for planar graphs, but for planar graphs with $O(n^{\frac{1}{3}})$ edges that violate planarity.*

□ In [**?**] the number of clusters is $O(n^{\frac{1}{3}})$. In the beginning and after every $O(n^{\frac{1}{3}})$ edge insertions we rebuild sparse substitute graph S. and in these moments we need planarity. We can additionally maintain the set of non-planar edges, each edge of this set will present its own cluster and will not take part in the rebuilding of sparse substitution. Edge is *non-planar* if its insertion to current planar graph G makes it non-planar. ■

Proposition 3.2. *We can add edges in the graph and print out new pairs of reachable vertices for every insertion in $O(n^{\frac{5}{3}})$ total time for every sequence of insertions if our model allows parallel computations.*

□ From [**?**] we can add an edge (u, v) in $O(n^{\frac{2}{3}} \log n)$ amortized time. We want to get all pairs of vertices that are connected through the new edge.

We can run DFS from v and DFS on reversed edges from u in graph S of sparse substitutes. DFS runs in time proportional to the size of the graph $S - O(n^{\frac{2}{3}} \log n)$. After that for every cluster in the original graph G we create dummy vertex s and edges from s to all boundary vertices in the cluster that were reached from u and v respectively. Then run DFS from this dummy vertex s. We print out every vertex that was reached by our DFSs, so we get two lists U and V for beginnings and the endings of the paths, going through (u, v).

If any vertex w is reachable from v (without loss of generation), then it is a boundary vertex or lie in some cluster and is reachable from some boundary vertex of this cluster. In both cases, one of DFS's will print it out and the algorithm is correct.

If we can run these DFS's in parallel (there are $O(n^{\frac{1}{3}})$ clusters and the same number of DFS's), then each of them will take $O(n^{\frac{2}{3}})$ amount of time (all cluster have $O(n^{\frac{2}{3}})$ edges by definition in [**?**]). As maximum number of edges in the planar graph is linear, the total amount of time will be $O(n^{\frac{5}{3}})$ for every sequence of edge insertions. ■

Conjecture 3.3. *If our model does not allow parallel computations total amount of time for every sequence of insertions is at most $O(n^2)$ and planarity does not get us any advantage in solving the problem of dynamic reachability (in amortized time per edge and query).*

[Thoughts] If our model can not run algorithms in parallel, then the amount of time taken by iterating DFS's described above for

every cluster is linear — $O(n)$. This means that usual DFS on the original graph G has the same complexity and we will spend in total $O(n^2)$ time. ■

maybe we can spare some space ($O(n^2)$?) so we can store list of reachable vertices from any boundary vertex and somehow update them during the edge additions?

4 CFPQ ON UNDIRECTED GRAPHS

4.1 Motivation

The bottleneck of the Tensor-based algorithm is computing Incremental transitive closure, which cannot be solved in subcubic time.

So to speed up the algorithm, it is necessary to modify this particular part somehow.

We decided to relax the task and find incremental transitive closure, not in a directed graph, but an undirected one. The motivation behind this decision is simple: the problem of Incremental transitive closure in an undirected graph is easier than directed and can be solved in less running time.

4.2 Definitions

define G – input grammar
define q_i — grammar states
define $\mathcal{G}$ — input graph
define u, v — graph vertices

4.3 Algorithm

We use two data structures: queue Q, storing edges, that should be added, but have not been yet, and Disjoint Set D, maintaining components in Kronecker product graph.

Firstly, we add all arcs from initial Kronecker product into Q and then start to iterate over it. At each step we take next arc from queue and join corresponding sets in D. If they are in differents sets and adding the edge leads to appearance of new path $(q_s, u) \rightsquigarrow (q_f, v)$ from start to final terminal of some box S, we add new arc form u to v labeled with S. After that we add this arc into Kronecker product: we iterate over all arcs $q_i \to q_j$ in grammar G labeled with S and add new edge $(q_i, u) \to (q_j, v)$ into Q.

Listing 2 Undirected Kronecker product based CFPQ

```
1:  function UndirectedContextFreePathQuerying(G, 𝒢)
2:      R ← Recursive automata for G
3:      M1 ← Adjacency matrix for R
4:      M2 ← Adjacency matrix for 𝒢
5:      M3 ← M1 ⊗ M2
6:      Q ← Empty Queue
7:      D ← Empty Disjoint Set
8:      n ← dim(M3)
9:      for i ∈ 0..n − 1 do
10:         D.MakeSet(i) ▷ Create empty set for every element in
        Kronecker product
11:     for x ∈ 0..n − 1 do
12:         for y ∈ 0..n − 1 do
13:             if M3[x, y] then
14:                 Q.Add(x, y)                    ▷ Add initial edges
15:     while Q is not empty do
16:         (x, y) ← Q.Pop()
17:         for (qs, u) ∈ D.Find(x).GetInitialStates() do
18:             for (qf, v) ∈ D.Find(y).GetFinalStates() do
19:                 S ← G.GetLabelByState(qs)
20:                 for (qi, qj) ∈ GetEdgesByLabel(S) do
21:                     Q.Push((qi, u), (qj, v))
22:                     M3[(qi, u), (qj, v)] ← 1    ▷ Add new edges
23:         D.Merge(x, y)
24:     return M3
```

To find new paths from start to final terminal quickly, we store for every set in D all initial and final states in it. To get this information we use supporting methods $GetInitialStates()$ and $GetFinalStates()$. To get RSM box label by state id we use supporting method $GetLabelByState()$. To get all edges in grammar labeled with S we use supporting method $GetEdgesByLabel()$.

TODO: rewrite in other notations

TODO: write about dealing with eps-transitions.

4.4 Complexity

TODO

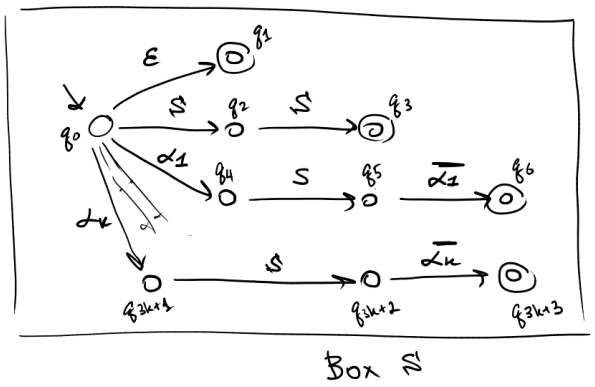
4.5 Bidirected graphs

Definition 4.1. **Bidirected Graphs.**

A Σ_k labeled graph $G = (V, E)$ is called bidirected if for every pair of nodes $u, v \in V$ for all $1 \le i \le k$ we have that $(u, v, \alpha_i) \in E$ iff $(v, u, \overline{\alpha_i}) \in E$. Informally, the edge relation is symmetric, and the labels of symmetric edges are complimentary wrt to opening and closing parenthesis.

Remark. *For bidirected graphs the Dyck-reachability relation forms an equivalence, i.e., for all bidirected graphs G, for every pair of nodes u m v, we have that v is Dyck-reachable from u iff u is Dyck-reachable from v.*

Note. *Dyck language RSM-grammar box (there is only one).*



Theorem 4.2. *The Algorithm 2 works correctly on bidirected graphs and Dyck grammars.*

Proof. We only need to prove, that for every pair $u, v \in V(\mathcal{G})$ there is a path $(q_s, u) \rightsquigarrow (q_{f_i}, v)$ in (**directed**) Kronecker product $G \otimes \mathcal{G}$ iff there is such path $(q_s, u) \rightsquigarrow (q_{f_j}, v)$ in an **undirected** product (there q_s is the initial state and q_{f_i}, q_{f_j} are some final states of G respectively).

$\Rightarrow$

Obviously (if (q_f, v) is reachable from (q_s, u) by directed edges, it is all the more reachable by undirected edges).

$\Leftarrow$

At first, note that the Kronecker product $G \otimes \mathcal{G}$ forms some kind of a layered structure — i-th layer consists of vertices (q_i, v), where q_i is i-th RSM state. Because RSM is topologically sorted (TODO), every edge $(q_i, u) \to (q_j, v)$ goes forward.

We will call path *simple* if it visits every layer no more than once.

We prove the claim by induction on the l (path length).

Clearly the result is true for $l \leq 3$, because the only way to achive final vertex in $1, 2$ or 3 edges is by a simple vertical path (which exists in the original graph too).

Otherwise (if $l \geq 4$), path is not simple.

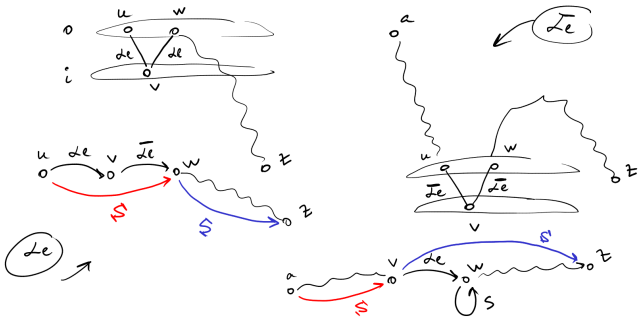
Consider the first flex point of the path, that is the vertex (q_i, v) such that edges $(q_j, u) \to (q_i, v)$ and $(q_i, v) \to (q_k, w)$ are in the path and $j, k \leq i$ (so, the path is convex at this point).

Looking at the grammar graph we can notice, that every state has indegree ≤ 1. So at the flex point there are actually two same-labeled edges (that is, $j = k$).

There can be three different types of labels on those edges:

- α_l-label
  path: $(q_0, u) \to (q_i, v) \to (q_0, w) \to \cdots \to (q_f, z)$.
  Since α_l-labeled edges could only be added on the initialization stage, graph $\mathcal{G}$ contains edges $u \xrightarrow{\alpha_l} v$ and $w \xrightarrow{\alpha_l} v$. Notice, that cause $\mathcal{G}$ is bidirected, it also has to contain edges $v \xrightarrow{\overline{\alpha_l}} u$ and $v \xrightarrow{\overline{\alpha_l}} w$.
  Now we can notice, that w is Dyck-reachable (by the path $\alpha_l \overline{\alpha_l}$) from u, so there is an S-labeled edge from u to w. We can also conclude, that (by induction) there is a directed path from (q_0, w) to (q_f, z) (there z is the end of the path and q_f is some final state of G), so there is an S-labeled edge from w to z.

  Using this two observation we can construct a directed path from u to z: $u \xrightarrow{S} w \xrightarrow{S} z$.
- S-label
  path: $(q_0, a) \to \cdots \to (q_j, u) \to (q_i, v) \to (q_j, w) \to \cdots \to (q_f, z)$.
  $\mathcal{G}$ contains S-labeled edges $u \xrightarrow{S} v$ and $w \xrightarrow{S} v$. Since $\mathcal{G}$ is bidirected, then by 4.5 $v \xrightarrow{S} u$ and $v \xrightarrow{S} w$. Combining $u \xrightarrow{S} v$ and $v \xrightarrow{S} w$ we get that $u \xrightarrow{S} w$.
  No we want to sort of contract this edge, joining u and w (on the j-th level). Then we can get (by induction hypothesis) the directed path from $(q_0, a) \rightsquigarrow (q_f, z)$. If new path does not contain joined uw vertex, then that's the answer. Otherwise we can split this vertex back, inserting between u and w the S-labeled path (that one, from $u \xrightarrow{S} w$ edge). We can do it, because the both of these paths form correctly matched parenthesis (TODO: we can prove this using stack-based checking algorithm).
  Two other cases can be proved the same way, but I find it a little dishonest
- $\overline{\alpha_l}$-label
  path: $(q_0, a) \to \cdots \to (q_j, u) \to (q_f, v) \to (q_j, w) \to \cdots \to (q_f, z)$.
  Since α_l-labeled edges could only be added on the initialization stage, graph $\mathcal{G}$ contains edges $u \xrightarrow{\overline{\alpha_l}} v$ and $w \xrightarrow{\overline{\alpha_l}} v$. Notice, that cause $\mathcal{G}$ is bidirected, it also has to contain edges $v \xrightarrow{\alpha_l} u$ and $v \xrightarrow{\alpha_l} w$.
  By induction, we get that $a \xrightarrow{S} v$. Now we will construct a second part of the path: $(q_0, v) \xrightarrow{\alpha_l} (q_{j-1}, w) \xrightarrow{S} (q_j, w) \rightsquigarrow (q_f, z)$ ($(q_{j-1}, w) \xrightarrow{S} (q_j, w)$ — initial S-loop). By induction, we have directed simple version of this path, so $v \xrightarrow{S} z$.
  Combining this two paths ($a \xrightarrow{S} v$ and $v \xrightarrow{S} z$) we get $a \xrightarrow{SS} z \Rightarrow a \xrightarrow{S} z$ — desired path.



□

4.6 MYCOPKA

bidirected guys [?]

resalt about Dyck-reachability components [?]

5 CONCLUSION

Conclusion and future work.

Efficient implementation?
!!!