

ACM ISBN XXX-X-XXXXX-XXX-X...\$15.00

- (2) Subcubic for planar graphs. This criterion is output-sensitive, so it is not practical, but opens a theoretical way to find more subclass with subcubic complexity.
- (3) Interconnection between CFPQ and dynamic transitive closure. Conjecture on sublinear dynamic transitive closure and subcubic CFPQ.
- (4) Evaluation. RPQ, CFPQ.

2 PRELIMINARIES

In this section we introduce basic notation and definitions from graph theory and formal language theory which are used in our work.

2.1 Context-Free Path Querying Problem

We introduce *Context-Free Path Querying Problem (CFPQ)* over directed edge-labelled graphs. Diraph $\mathcal{G} = \langle V, E, L \rangle$

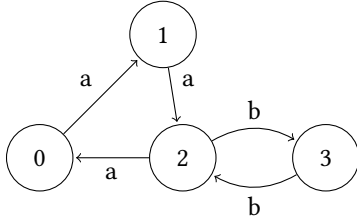


Figure 1: The example of input graph \mathcal{G}

Each graph can be represented as adjacency matrix M . We use decomposition to the matrix to the set of Boolean matrices

$$\mathcal{M} = \{M^l \mid l \in L, M^l[i, j] = 1 \iff l \in M[i, j]\}.$$

This way we reduce operations over custom semirings to Boolean semiring.

Grammar G

CFPQ with different semantics

Reachability semantics:

12

All paths semantics:

23

2.2 Recursive State Machines

?Finite state machine. Regexp to FSM.?

Also known as recursive networks [?], recursive automata [?], !!!

Definition

Properties.

Grammar to RSM conversion algorithm. Example of conversion.

Boolean decomposition of adjacency matrix

2.3 Graph Kronecker Product

Kronecker product definition.

Tensor product definition.

Tensor product of adjacency matrices.

Tensor product for FSM intersection.

Tensor product for FSM intersection over Boolean semiring.

Rsm and FSM intersection classical theorem proof?

Dynamic graph problems?

3 CONTEXT-FREE PATH QUERYING BY KRONECKER PRODUCT

In this section, we introduce the algorithm for CFPQ which is based on Kronecker product of Boolean matrices. The algorithm provides the ability to solve all-pairs CFPQ in all-paths semantics (according to Hellings [?]) and consists of two the following parts.

- (1) Index creation. In the first step, the algorithm computes an index which contains information which is necessary to restore paths for specified pairs of vertices. This index can be used to solve the reachability problem without paths extraction. Note that this index is finite even if the set of paths is infinite.
- (2) Paths extraction. All paths for the given pair of vertices can be enumerated by using the index computed at the previous step. As far as the set of paths can be infinite, all paths cannot be enumerated explicitly, and advanced techniques such as lazy evaluation are required for implementation. Anyway, a single path can be always extracted by using standard techniques.

We describe both these steps, prove correctness, and provide time complexity estimations. For the first step we firstly introduce naïve algorithm. After that we show how to achieve cubic time complexity by using dynamic transitive closure algorithm and demonstrate that this technique allows us to get truly subcubic CFPQ algorithm for planar graphs.

After that we provide step-by-step example of query evaluation by using the proposed algorithm.

3.1 Index Creation Algorithm

In this section, we introduce the algorithm for the computation of context-free reachability in a graph \mathcal{G} . The algorithm determines the existence of a path, which forms a sentence of the language defined by the input RSM R , between each pair of vertices in the graph \mathcal{G} . The algorithm is based on the generalization of the FSM intersection for an RSM, and an input graph. Since a graph can be interpreted as a FSM, in which transitions correspond to the labeled edges between vertices of the graph, and an RSM is composed of a set of FSMs, the intersection of such machines can be computed

using the classical algorithm for FSM intersection, presented in [4].

The intersection can be computed as a Kronecker product of the corresponding adjacency matrices for an RSM and a graph. Since we are only determining the reachability of vertices, it is enough to represent intersection result as a Boolean matrix. It simplifies the algorithm implementation and allows one to express it in terms of basic matrix operations.

3.1.1 Naïve Version. Listing 1 shows main steps of the algorithm. The algorithm accepts context-free grammar $G = (\Sigma, N, P)$ and graph $\mathcal{G} = (V, E, L)$ as an input. An RSM R is created from the grammar G . Note, that R must have no ε -transitions. M_1 and M_2 are the adjacency matrices for the machine R and the graph \mathcal{G} correspondingly.

Then for each vertex i of the graph \mathcal{G} , the algorithm adds loops with non-terminals, which allows deriving ε -word. Here the following rule is implied: each vertex of the graph is reachable by itself through an ε -transition. Since the machine R does not have any ε -transitions, the ε -word could be derived only if a state s in the box B of the R is both initial and final. This data is queried by the `getNonterminals()` function for each state s .

The algorithm terminates when the matrix M_2 stops changing. Kronecker product of matrices M_1 and M_2 is evaluated for each iteration. The result is stored in M_3 as a Boolean matrix. For the given M_3 a C_3 matrix is evaluated by the `transitiveClosure()` function call. The M_3 could be interpreted as an adjacency matrix for an directed graph with no labels, used to evaluate transitive closure in terms of classical graph definition of this operation. Then the algorithm iterates over cells of the C_3 . For the pair of indices (i, j) , it computes s and f – the initial and final states in the recursive automata R which relate to the concrete $C_3[i, j]$ of the closure matrix. If the given s and f belong to the same box B of R , $s = q_B^0$, and $f \in F_B$, then `getNonterminals()` returns the respective non-terminal. If the condition holds then the algorithm adds the computed non-terminals to the respective cell of the adjacency matrix M_2 of the graph.

The functions `getStates` and `getCoordinates` (see listing 2) are used to map indices between Kronecker product arguments and the result matrix. The Implementation appeals to the blocked structure of the matrix C_3 , where each block corresponds to some automata and graph edge.

The algorithm returns the updated matrix M_2 which contains the initial graph \mathcal{G} data as well as non-terminals from N . If a cell $M_2[i, j]$ for any valid indices i and j contains symbol $S \in N$, then vertex j is reachable from vertex i in grammar G for non-terminal S .

LEMMA 3.1. *Let $\mathcal{G} = (V, E, L)$ be a graph and $G = (\Sigma, N, P)$ be a grammar. Let $\mathcal{G}_k = (V, E_k, L \cup N)$ be graph and M_k its*

Listing 1 Kronecker product based CFPQ

```

1: function CONTEXTFREEPATHQUERYING( $G, \mathcal{G}$ )
2:    $R \leftarrow$  Recursive automata for  $G$ 
3:    $M_1 \leftarrow$  Adjacency matrix for  $R$ 
4:    $M_2 \leftarrow$  Adjacency matrix for  $\mathcal{G}$ 
5:   for  $s \in 0..dim(M_1) - 1$  do
6:     for  $i \in 0..dim(M_2) - 1$  do
7:        $M_2[i, i] \leftarrow M_2[i, i] \cup \text{getNonterminals}(R, s, s)$ 
8:   while Matrix  $M_2$  is changing do
9:      $M_3 \leftarrow M_1 \otimes M_2$  ▷ Evaluate Kronecker product
10:     $C_3 \leftarrow \text{transitiveClosure}(M_3)$ 
11:     $n \leftarrow \text{dim}(M_3)$  ▷ Matrix  $M_3$  size =  $n \times n$ 
12:    for  $(i, j) \in [0..n - 1] \times [0..n - 1]$  do
13:      if  $C_3[i, j]$  then
14:         $s, f \leftarrow \text{getStates}(C_3, i, j)$ 
15:        if  $\text{getNonterminals}(R, s, f) \neq \emptyset$  then
16:           $x, y \leftarrow \text{getCoordinates}(C_3, i, j)$ 
17:           $M_2[x, y] \leftarrow M_2[x, y] \cup \text{getNonterminals}(R, s, f)$ 
18:  return  $M_2$ 

```

Listing 2 Help functions for Kronecker product based CFPQ

```

1: function GETSTATES( $C, i, j$ )
2:    $r \leftarrow \text{dim}(M_1)$  ▷  $M_1$  is adjacency matrix for automata  $R$ 
3:   return  $\lfloor i/r \rfloor, \lfloor j/r \rfloor$ 
4: function GETCOORDINATES( $C, i, j$ )
5:    $n \leftarrow \text{dim}(M_2)$  ▷  $M_2$  is adjacency matrix for graph  $\mathcal{G}$ 
6:   return  $i \bmod n, j \bmod n$ 

```

adjacency matrix of the execution some iteration $k \geq 0$ of the algorithm ???. Then for each edge $e = (m, S, n) \in E_k$, where $S \in N$, the following statement holds: $\exists m\pi n : S \rightarrow_G l(\pi)$.

PROOF. (Proof by induction)

Basis: For $k = 0$ and the statement of the lemma holds, since $M_0 = M$, M where is adjacency matrix of the graph G . Non-terminals, which allow to derive ε -word, are also added at algorithm preprocessing step, since each vertex of the graph is reachable by itself through an ε -transition.

Inductive step: Assume that the statement of the lemma holds for any $k \leq (p - 1)$ and show that it also holds for $k = p$, where $p \geq 1$.

For the algorithm iteration p the Kronecker product K_p and transitive closure C_p are evaluated as described in the algorithm. By the properties of this operations, some edge $e = ((s, m), (f, n))$ exists in the directed graph, represented by adjacency matrix C_p , if and only if $\exists s\pi'f$ in the RSM graph, represented by matrix M_r , and $\exists m\pi n$ in graph, represented by M_{p-1} . Concatenated symbols along the path π' form some derivation string v , composed from terminals and non-terminals, where $v \rightarrow_G l(\pi)$ by the inductive assumption.

The new edge $e = (m, S, n)$ will be added to the E_p only if s and f are initial and final states of some box B of the RSM corresponding to the non-terminal S_B . In this case, the grammar G has the derivation rule $S_B \rightarrow_G v$, by the inductive assumption $v \rightarrow_G l(\pi)$. Therefore, $S_B \rightarrow_G l(\pi)$ and this completes the proof of the lemma. □

LEMMA 3.2. Let $\mathcal{G} = (V, E, L)$ be a graph and $G = (\Sigma, N, P)$ be a grammar. Let $\mathcal{G}_k = (V, E_k, L \cup N)$ be graph and M_k its adjacency matrix of the execution some iteration $k \geq 1$ of the algorithm $??$. For any path $m\pi n$ in graph \mathcal{G} with word $l = l(\pi)$ if exists the derivation tree of l for the grammar G and starting non-terminal S with the height $h \leq k$, then $\exists e = (m, S, n) : e \in E_k$.

PROOF. (Proof by induction)

Basis: Show that statement of the lemma holds for the $k = 1$. Matrix M and edges of the graph \mathcal{G} contains only labels from L . Since the derivation tree of height $h = 1$ contains only one non-terminal S as a root and only symbols from $\Sigma \cup \varepsilon$ as leaves, for all paths, which form a word with derivation tree of the height $h = 1$, the corresponding nonterminals will be added to the M_1 via preprocessing step and first iteration of the algorithm. Thus, the lemma statement holds for the $k = 1$.

Inductive step: Assume that the statement of the lemma hold for any $k \leq (p - 1)$ and show that it also holds for $k = p$, where $p \geq 2$.

For the algorithm iteration p the Kronecker product K_p and transitive closure C_p are evaluated as described in the algorithm. By the properties of this operations, some edge $e = ((s, m), (f, n))$ exists in the directed graph, represented by adjacency matrix C_p , if and only if $\exists s\pi_1 f$ in the RSM graph, represented by matrix M_{RSM} , and $\exists m\pi n$ in graph, represented by M_{p-1} .

For any path $m\pi n$, such that exist derivation tree of height $h < k$ for the word $l(\pi)$ with root non-terminal S , there exists edge $e = (m, S, n) : e \in E_k$ by inductive assumption.

Suppose, that exists derivation tree T of height $h = p$ with the root non-terminal S for the path $m\pi n$. The tree T is formed as $S \rightarrow a_1..a_d, d \geq 1$ where $\forall i \in [1..d]$ a_i is sub-tree of height $h_i \leq p - 1$ for the sub-path $m_i\pi_i n_i$. By inductive hypothesis, there exists path π_i for each derivation sub-tree, such that $m = m_1\pi_1 m_2..m_d\pi_d m_{d+1} = n$ and concatenation of these paths forms $m\pi n$, and the root non-terminals of this sub-trees are included in the matrix M_{p-1} .

Therefore, vertices $m_i \forall i \in [1..d]$ form path in the graph, represented by matrix M_{p-1} , with complete set of labels. Thus, new edge between vertices m and n with the respective non-terminal S will be added to the matrix M_p and this completes the proof of the lemma. \square

THEOREM 3.3. Let $\mathcal{G} = (V, E, L)$ be a graph and $G = (\Sigma, N, P)$ be a grammar. Let $\mathcal{G}_R = (V, E_R, L)$ be a result graph for the execution of the algorithm $??$. The following statement holds: $e = (m, S, n) \in E_R$, where $S \in N$, if and only if $\exists m\pi n : S \rightarrow_G l(\pi)$.

PROOF. This theorem is a consequence of the Lemma 3.1 and Lemma 3.2. \square

THEOREM 3.4. Let $\mathcal{G} = (V, E, L)$ be a graph and $G = (\Sigma, N, P)$ be a grammar. The algorithm $??$ terminates in finite number of steps.

PROOF. The main algorithm *while-loop* is executed while graph adjacency matrix M is changing. Since the algorithm only adds the edges with non-terminals from N , the maximum required number of iterations is $|N| \times |V| \times |V|$, where each component has finite size. This completes the proof of the theorem. \square

3.1.2 *Application of Dynamic Transitive Closure.* In this subsection we show how to reduce the time complexity of the Algorithm 1 by avoiding redundant calculations.

It is easy to see that the most time-consuming steps in the Algorithm 1 are the Kronecker product and transitive closure computations. Recall that the matrix M_2 is always changed in incremental manner i. e. elements (edges) are added to M_2 (and are never deleted from it) on every iteration of the Algorithm 1. So one does not need to recompute the whole product or transitive closure if an appropriate data structure is maintained.

To deal with the Kronecker product computation, we use the left-distributivity of the Kronecker product. Let A_2 be a matrix with newly added elements and B_2 be a matrix with the all previously found elements, such that $M_2 = A_2 + B_2$. Then by the left-distributivity of the Kronecker product we have $M_1 \otimes M_2 = M_1 \otimes (A_2 + B_2) = M_1 \otimes A_2 + M_1 \otimes B_2$. Notice that $M_1 \otimes B_2$ is known and is already in the matrix M_3 and its transitive closure also is already in the matrix C_3 , because it was calculated on the previous iterations, so it is left to update some elements of M_3 by computing $M_1 \otimes A_2$, which can be done in $O(|A_2||M_1|)$ time, where $|A|$ denotes the number of non-zero elements in a matrix A .

The fast computation of transitive closure can be obtained by using incremental dynamic transitive closure technique. We use an approach by Ibaraki and Katoh [5] to maintain dynamic transitive closure. The key idea of their algorithm is to recalculate reachability information only for those vertices, which become reachable after insertion of the certain edge (see Figure 2 for details). The algorithm is presented in Listing 3 (we have slightly modified it to efficiently track new elements of the matrix C_3).

Final version of the modified Algorithm 1 is shown in Listing 4.

THEOREM 3.5. Let $\mathcal{G} = (V, E, L)$ be a graph and $G = (\Sigma, N, P)$ be a grammar. The Algorithm 4 calculates a result graph $\mathcal{G}_R = (V, E_R, L)$ in $O(n^3)$ time.

Listing 3 The dynamic transitive closure procedure

```

1: function ADD( $C_3, i, j$ )
2:    $n \leftarrow$  Number of rows in  $C_3$ 
3:    $C'_3 \leftarrow$  Empty matrix
4:   for  $u \in 0 \dots n \mid u \neq j \ \& \ C_3[u, i] = 1 \ \& \ C_3[u, j] = 0$  do
5:     for  $v \in 0 \dots n$  do
6:       if  $C_3[u, v] = 0 \ \& \ C_3[j, v] = 1$  then
7:          $C'_3[u, v] \leftarrow 1$ 
8:   return  $C'_3$ 

```

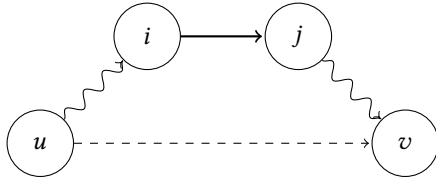


Figure 2: The vertex j become reachable from the vertex u after the addition of edge (i, j) . Then the vertex v is reachable from u after inserting the edge (i, j) if v is reachable from j .

Listing 4 Kronecker product based CFPQ using dynamic transitive closure

```

1: function CONTEXTFREEPATHQUERYING( $G, \mathcal{G}$ )
2:    $R \leftarrow$  Recursive automata for  $G$ 
3:    $M_1 \leftarrow$  Adjacency matrix for  $R$ 
4:    $M_2 \leftarrow$  Adjacency matrix for  $\mathcal{G}$ 
5:    $A_2 \leftarrow$  Adjacency matrix for  $\mathcal{G}$ 
6:    $C_3 \leftarrow$  The empty matrix
7:   for  $s \in 0 \dots \dim(M_1) - 1$  do
8:     for  $i \in 0 \dots \dim(M_2) - 1$  do
9:        $M_2[i, i] \leftarrow M_2[i, i] \cup \text{getNonterminals}(R, s, s)$ 
10:  while Matrix  $M_2$  is changing do
11:     $M'_3 \leftarrow M_1 \otimes A_2$ 
12:     $A_2 \leftarrow$  The empty matrix of size  $n \times n$ 
13:    for  $M'_3[i, j] \mid M'_3[i, j] = 1$  do
14:       $C_3[i, j] \leftarrow 1$ 
15:       $C'_3 \leftarrow \bigcup_{(i,j)} \text{add}(C_3, i, j)$  ▷ Updating the transitive closure
16:       $C_3 \leftarrow C_3 + C'_3$ 
17:   $n \leftarrow \dim(M_3)$ 
18:  for  $(i, j) \in [0 \dots n - 1] \times [0 \dots n - 1]$  do
19:    if  $C'_3[i, j]$  then
20:       $s, f \leftarrow \text{getStates}(C'_3, i, j)$ 
21:      if  $\text{getNonterminals}(R, s, f) \neq \emptyset$  then
22:         $x, y \leftarrow \text{getCoordinates}(C'_3, i, j)$ 
23:         $M_2[x, y] \leftarrow M_2[x, y] \cup \text{getNonterminals}(R, s, f)$ 
24:         $A_2[x, y] \leftarrow A_2[x, y] \cup \text{getNonterminals}(R, s, f)$ 
25:  return  $M_2$ 

```

PROOF. Let $|A|$ be a number of non-zero elements in a matrix A . Consider the total time which is needed for computing the Kronecker products. The elements of the matrices $A_2^{(i)}$ are pairwise distinct on every i -th iteration of the Algorithm therefore we have $\sum_i T(M_1 \otimes A_2^{(i)}) = |M_1| \otimes \sum_i |A_2^{(i)}| = |M_1| O(n^2)$ operations in total.

Now we derive the time complexity of maintaining the dynamic transitive closure. Notice that C_3 has size of $O(n^2)$ so no more than $O(n^2)$ edges will be added during all iterations of the Algorithm. The condition in the line 4 in Listing 3 is calculated $O(n)$ times for every inserted edge (i, j) . Thus we

have $O(n^2 n) = O(n^3)$ operations in total. The operation from line 6 requires $O(n)$ time for a given vertex u . This operation is performed for every pair (j, v) of vertices such that a vertex j became reachable from the vertex u . There are no more than $O(n^2)$ such pairs, so line 6 will be executed at most $O(n^2 n) = O(n^3)$ times during the entire computation. Therefore $O(n^3)$ operations are performed to maintain dynamic transitive closure during all iteration of the Algorithm 4.

Notice that the matrix C'_3 contains only new elements, therefore C_3 can be updated directly using only $|C'_3|$ operations and hence $O(n^2)$ operations in total. The same holds for cycle in line 18 of the Algorithm 4, because operations are performed only for non-zero elements of the matrix $|C'_3|$. Finally, we have that the time complexity of the Algorithm 4 is $O(n^2) + O(n^3) + O(n^2) + O(n^2) = O(n^3)$. \square

Notice that the obtained cubic time bound is close to the currently best known upper bound for the CFPQ evaluation (the asymptotically fastest known method has a complexity of $O(n^3 / \log n)$) [2]. However it is open problem whether a truly sub-cubic algorithm exists for the CFL-reachability problem (and hence, for CFPQ evaluation) [1].

Subcubic for planar graphs using [6].

Cojecture on sublinear dynamic transitive closure and subcubic CFPQ.

3.2 Paths Extraction Algorithm

After index created one can enumerate all paths between specified vertices.

Listing 5 Paths extraction algorithm

```

1:  $C_3 \leftarrow$  result of index creation algorithm: final transitive closure
2:  $M_1 \leftarrow$  the set of adjacency matrices of the final graph
3:  $M_2 \leftarrow$  the set of adjacency matrices of the input RSM
4: function GETPATHS( $v_s, v_f, N$ )
5:    $s \leftarrow$  Start states of automata for  $N$ 
6:    $f \leftarrow$  Final states of automata for  $N$ 
7:    $res \leftarrow \text{getPathsInner}(\text{getVNum}(s, v_s), \text{getVNum}(f, v_f))$ 
8:   return  $res$ 
9: function GETSUBPATHS( $i, j, k$ )
10:   $l \leftarrow \{(i.g, t, k.g) \mid M_1[t][i.r, k.r] = 1 \ \& \ M_2[t][i.g, k.g] \in \bigcup_{N \mid M_1[N][i.r, k.r]} \text{GETPATHS}(i.g, k.g, N, C_3, M_1, M_2)\}$ 
11:   $r \leftarrow \{(k.g, t, j.g) \mid M_1[t][k.r, j.r] = 1 \ \& \ M_2[t][k.g, j.g] \in \bigcup_{N \mid M_1[N][k.r, j.r]} \text{GETPATHS}(k.g, j.g, N, C_3, M_1, M_2)\}$ 
12:  return  $l \cdot r$ 
13: function GETPATHSINNER( $i, j$ )
14:  parts  $\leftarrow \{k \mid C_3[i, k] = 1 \ \& \ C_3[k, j] = 1\}$ 
15:  return  $\bigcup_{k \in \text{parts}} \text{GETSUBPATHS}(i, j, k, C_3, M_1, M_2)$ 

```

Ideas and description.

Correctness.

Time complexity.

3.3 An example

In this section we introduce detailed example to demonstrate steps of the proposed algorithm. Our example is based on

5.3 Grammar transformation

On query optimization.

Memory aliases.

Synthetic???

6 RELATED WORK

CFPQ algorithms: Hellings [?], Bradford [?], Azimov [?], Verbitskaya [?], Ciro [?], form static code analysis [?],

RPQ algorithms: derivatives [?], Glushkov [?], etc.!!!! [?]

Linear algebra based approaches to evaluate queries (data-log, SPARQL, etc) [?] Not focused on types of queries.

Subcubic CFPQ: Bradford, Chatterjee, RSM-s, Smth else?

7 CONCLUSION

!!!! Was presented.

Subcubic CFPQ in general case — sublinear transitive closure.

On RSM optimization and query optimization.

We evaluate naïve implementation. Try to use advanced dynamic algorithms [3].

HiCOO format.

GPGPU-based implementation. Multi-GPU version.

Full integration with Graph DB.

Other semantics: shortest path, simple path and so on.

Streaming graph querying.

Specialization on query.

!!!

REFERENCES

- [1] Krishnendu Chatterjee, Bhavya Choudhary, and Andreas Pavlogiannis. 2017. Optimal Dyck Reachability for Data-Dependence and Alias Analysis. *Proc. ACM Program. Lang.* 2, POPL, Article 30 (Dec. 2017), 30 pages. <https://doi.org/10.1145/3158118>
- [2] Swarat Chaudhuri. 2008. Subcubic algorithms for recursive state machines. In *POPL '08*.
- [3] Kathrin Hanauer, Monika Henzinger, and Christian Schulz. 2020. Faster Fully Dynamic Transitive Closure in Practice. In *18th Symposium on Experimental Algorithms (SEA 2020)*. <http://eprints.cs.univie.ac.at/6345/>
- [4] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2006. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA.
- [5] T. Ibaraki and N. Katoh. 1983. On-line computation of transitive closures of graphs. *Inform. Process. Lett.* 16, 2 (1983), 95 – 97. [https://doi.org/10.1016/0020-0190\(83\)90033-9](https://doi.org/10.1016/0020-0190(83)90033-9)
- [6] Sairam Subramanian. 1993. A fully dynamic data structure for reachability in planar digraphs. In *Algorithms—ESA '93*, Thomas Lengauer (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 372–383.