Warsaw University

Faculty of Mathematics,
Informatics and Mechanics

Piotr Sankowski

ALGEBRAIC GRAPH ALGORITHMS

Ph.D. Dissertation

Supervisor
dr hab. Krzysztof Diks
Institute of Informatics
Warsaw University

February 2005

Author's declaration

Aware of legal responsibility I hereby declare that I have written this dissertation myself and all the contents of the dissertation have been obtained by legal means.

Date	Author's signature

Supervisor's declaration

This dissertation is ready to be reviewed.

Date	Supervisor's signature

Abstract

The main topic of this dissertation is dynamic evaluation of algebraic functions such as matrix determinant, matrix adjoint, matrix inverse, and solving linear system of equations. We show that in dynamic setup the above problems can be solved faster than evaluating everything from scratch. In case when rows and columns of the matrix can change, we show an algorithm that needs $O(n^2)$ arithmetic operations per update and $O(1)$ arithmetic operations per query. In case when we allow only to change single entries of the matrix, we show two tradeoff algorithms. The fastest update for the first tradeoff costs $O(n^{1.575})$ arithmetic operations per update and $O(n^{0.575})$ arithmetic operations per query. The second tradeoff gives $O(n^{1.495})$ arithmetic operations per update and $O(n^{1.495})$ arithmetic operations per query.

We use dynamic determinant computations to solve the following problems in dynamic setup: computing the number of spanning trees of a graph and testing if a general graph has a perfect matching. These are the first dynamic algorithms for these problems. Next, with the use of the dynamic matrix inverse, we solve fully dynamic transitive closure in general directed graphs. We obtain for the first time an algorithm with $O(n^2)$ update time (worst-case) and constant query time and two algorithms for transitive closure in general digraphs with sub-quadratic update and query times. Our algorithms for transitive closure are randomized with one-sided error.

Next, we introduce dynamic algorithms for computing matrix determinant and matrix adjoint over general and Euclidean rings. These algorithms are used to construct an algorithm for dynamic shortest distances in unweighted graphs. Our algorithm supports updates in $O(n^{1.932})$ time and queries in $O(n^{1.288})$ time. These bound improve over previous results and solve a long-standing open problem if sub-quadratic dynamic algorithms for computing all pair shortest distances and single source shortest distances exist.

We also show that the dynamic matrix algorithms can be used to obtain efficient static algorithms for the perfect matching problem. Using the $O(n^2)$ algorithms for the dynamic matrix inverse, we obtain a very simple randomized algorithm for computing perfect matchings in $O(n^3)$ time. When the fast matrix multiplication is used, the complexity of this algorithm can be improved to $O(n^{\omega})$ time, where ω is the exponent of the best known matrix multiplication algorithm. Since $\omega < 2.38$, this algorithm breaks through the $O(n^{2.5})$ barrier for the matching problem.

Keywords: dynamic algorithms, determinant, matrix inverse, adjoint, transitive closure, shortest paths, perfect matchings, spanning trees.

ACM Classification: F.2.2, G.2.2.

Streszczenie

Głównym tematem rozprawy jest dynamiczne obliczanie funkcji algebraicznych takich jak wyznacznik, macierz minorów, macierz odwrotna, oraz rozwiązania układu równań liniowych. Pokazujemy tutaj, że w przypadku dynamicznym wszystkie te problemy mogą być rozwiązane szybciej niż obliczenie wszystkiego od nowa. Dla zmian kolumn i wierszy macierzy konstruujemy algorytmy wyliczające nowe wartości tych funkcji przy użyciu $O(n^2)$ operacji arytmetycznych. W przypadku zmian pojedynczych elementów macierzy, podajemy algorytm realizujący zmiany macierzy w $O(n^{1.575})$ operacjach arytmetycznych i odpowiadający na zapytania w $O(n^{0.575})$ operacjach. Prezentujemy także algorytm realizujący zmiany szybciej, w $O(n^{1.495})$ operacjach, kosztem droższych zapytań – $O(n^{1.495})$ operacji.

Przy użyciu algorytmów dla dynamicznego wyznacznika, rozwiązujemy po raz pierwszy następujące dwa problemy dynamiczne: zliczanie drzew rozpinających w grafach, oraz testowanie, czy graf ma doskonałe skojarzenie. Następnie, wykorzystując dynamiczne algorytmy obliczające macierz odwrotności, rozwiązujemy problem dynamicznego domknięcia przechodniego. Otrzymujemy pierwszy znany algorytm wykonujący modyfikacje grafu w pesymistycznym czasie $O(n^2)$, oraz odpowiadający na zapytania w czasie stałym. Otrzymujemy również pierwsze algorytmy dla tego problemu o podkwadratowym koszcie. Nasze algorytmy dla domknięcia przechodniego są randomizowane z jednostronnym błędem.

Następnie podajemy algorytmy obliczania wyznacznika oraz macierzy minorów dla macierzy określonych nad pierścieniem przemiennym lub Euklidesowym. Algorytmy te znajdują zastosowanie w konstrukcji dynamicznego algorytmu obliczającego długości ścieżek w grafie. Nasz algorytm modyfikuje graf w czasie $O(n^{1.932})$, i odpowiada na zapytania w czasie $O(n^{1.288})$. Wynik ten jest rozwiązaniem otwartego dotychczas problemu istnia podkwadratowych dynamicznych algorytmów dla obliczania długości najkrótszych ścieżek w grafie, oraz odległości od jednego wybranego wierzchołka.

Pokazujemy także, że dynamiczne algorytmy macierzowe mogą zostać użyte do konstrukcji wydajnych statycznych algorytmów dla problemu doskonałych skojarzeń. Używając tych algorytmów otrzymujemy bardzo prosty randomizowany algorytm na znajdowanie doskonałych skojarzeń w czasie $O(n^3)$. Stosując szybkie mnożenie macierzy złożoność tego algorytmu możemy poprawić do $O(n^\omega)$, gdzie ω jest wykładnikiem najszybszego znanego algorytmu mnożenia macierzy.

Keywords: dynamiczne algorytmy, wyznacznik, odwrotność, macierz minorów, domknięcie przechodnie, najkrótsze ścierzki, doskonałe skojarzenia, drzewa rozpinające.

Klasyfikacja tematyczna ACM: F.2.2, G.2.2.

Contents

Chapter 1

Introduction

In this thesis we describe how algebraic methods can be used to construct efficient graph algorithms. The algebraic methods turned out to be very useful in many graph applications, starting from transitive closure computations and ending on counting perfect matchings. The constructed algorithms use matrix operations, such as multiplication or computing determinant, as a basic building block. Thanks to this the algorithms usually gain on clearness. Also in many cases the algebraic approach yields the asymptotically fastest solutions. The basic example is the transitive closure problem. One of the classical results states that the problem is equivalent to boolean matrix multiplication.

The main topic of this thesis are dynamic graph algorithms. We show that also in dynamic computations the algebraic approach is very useful and can be used to obtain asymptotically fastest algorithms. The presented algorithms are also relatively simpler in comparison to the earlier solutions to these problems, because of the modular composition. They are based on a new building block, namely data structures supporting dynamic matrix operations such as matrix inverse and matrix determinant. Using this building block we are able to show the asymptotically fastest algorithms for dynamic transitive closure. We also state a more general result and show that the dynamic transitive closure problem is not harder than the dynamic matrix inverse problem, i.e., when we know how to solve the second one with given dynamic complexities, the first one can also be solved with the same complexities. These results use a simple, yet astonishing observation that the inverse matrix encodes the transitive closure.

Moreover, we obtain the first dynamic algorithms for the problem of counting spanning trees in a graph and testing if a general graph has a perfect matching. These results are direct implications of our dynamic matrix algorithms, because the problems can be solved by computing the determinant of the appropriately defined adjacency matrix of the graph. Next, we extend our algorithms to matrices over commutative rings. This result implies the asymptotically fastest algorithm for the dynamic shortest paths problem in the case of unweighted graphs.

The problem of testing, wether a dynamic graph has a perfect matching, can be used for development of very efficient algorithms for finding perfect matchings in a graph. By extending these ideas we can show that the perfect matching problem is not harder than the matrix multiplication. This reduction gives the asymptotically fastest algorithms for the perfect matching problem. It is one of the first results that shows that an efficient solution to a static problem can be obtained with the use of dynamic algorithms.

Organization of this work

We start by giving the definitions and the historical background of the problems studied in this work - Chapter 2. In Chapter 3 we recall some basic definitions and concepts which will be used in the next chapters. In Chapter 4 we present our algorithms for the dynamic matrix problems. Next in Chapter 5, we extend the algorithms to the case of matrices over commutative rings. In Chapters 6, 7 and 8 we show how the problems of dynamic computing spanning tress, transitive closure, and distances in graphs can be solved with the use of our algorithms. The application of the dynamic matrix algorithms to computing perfect matchings in graphs is presented in Chapter 9.

Chapter 2

Thesis Overview

2.1 Dynamic Algebraic Algorithms

Let $\mathcal{R} = (S, +, \cdot, 0, 1)$ be a ring with elements from set S and appropriately defined addition $+$ and multiplication $\cdot$. Let $f : S^n \to S^m$ be an algebraic function over this ring. A *dynamic algebraic algorithm* must handle the following types of requests:

- **initialize**$(x_1, x_2, \dots, x_n)$: initialization with an input vector $(x_1, x_2, \dots, x_n)$;
- **update**$(k,\ x'_k)$: change input k to a new value x'_k;
- **query**(k): return the value of output k.

Our goal is to construct algorithms that support updates and queries as fast as possible. In particular, the updates must be faster than recomputing everything from scratch. The initialization time is generally not very important. However, we also try to make the initialization step as fast as possible.

An extensive study of dynamic algebraic problems was started by Reif and Tate [40]. They presented an $\Omega(n)$ lower bound for some simple dynamic algebraic problems such as multipoint polynomial evaluation, polynomial reciprocal, and extended polynomial GCD. They proved two time-space trade-off theorems applicable to dynamic algorithms for many algebraic functions. Moreover, they provided some general-purpose design techniques of dynamic algebraic algorithms. With the use of these techniques, they showed an $O(\sqrt{n})$ time per request algorithm for dynamic DFT and an $O(\sqrt{n \log n})$ time per request algorithm for polynomial multiplication. They also provided a technique for constructing parallel algorithms with optimal work.

Other lower bounds for dynamic algebraic problems were shown by Frandsen, Hansen and Miltersen [16]. They proved an almost tight $\Omega(\sqrt{n})$ lower bound for dynamic polynomial multiplication. They were also able to prove an $\Omega(n)$ lower bounds for the problems of computing determinant, adjoint, inverse, and solving linear system

Problem	**Lower Bound**	**Upper Bound**
matrix-vector multiplication matrix multiplication	$\Omega(n)$ [16]	$O(n)$ [simple multiplication]
polynomial multiplication	$\Omega(\sqrt{n})$ [16]	$O(\sqrt{n}\log n)$ [40]
polynomial evaluation	$\Omega(n)$ [40, 16]	$O(n)$ [Horner's algorithm]
matrix adjoint matrix inverse determinant linear system of equations	$\Omega(n)$ [40, 16]	$O(n^{1.495})$ [this thesis]
DFT	$\Omega(\frac{\log^2 n}{\log\log n})$ [16]	$O(\sqrt{n})$ [40]
two dimensional DFT	$\Omega(\frac{\log^2 n}{\log\log n})$ [16]	$O(\sqrt{n}\log n)$ [40]
polynomial reciprocal	$\Omega(n)$ [40]	$O(n\log n)$ [40]
extended polynomial GCD	$\Omega(n)$ [40]	$O(n\log^2 n)$ [40]
prefix sum		$O(\log n)$ [17]

Figure 2.1: Upper and lower bounds for some dynamic algebraic problems. The bounds are with respect to both update and query operations.

of equations. However, till now the best solution to these matrix problems required reevaluating everything from scratch. This gives only an $O(n^\omega)$ upper bound, where ω is the exponent in the best known matrix multiplication algorithm (see Section 3.4). The table in Figure 2.1 summarizes some of the known results.

2.1.1 Dynamic Matrix Algorithms

We consider the following types of dynamic matrix problems:

- **determinant** $\mathcal{R}^{n^2} \to \mathcal{R}$: The input is interpreted as an $n \times n$ matrix. The output is its determinant.
- **adjoint** $\mathcal{R}^{n^2} \to \mathcal{R}^{n^2}$: The input is interpreted as an $n \times n$ matrix A. The output is interpreted as $n \times n$ adjoint $\operatorname{adj}(A)$ of the input matrix.
- **inverse** $\mathcal{R}^{n^2} \to \mathcal{R}^{n^2}$, where $\mathcal{R}$ is a field: This is a function from non-singular $n \times n$ matrices that maps matrix A into the corresponding inverse matrix A^{-1}.
- **linear system of equations** $\mathcal{R}^{n^2+n} \to \mathcal{R}^n$, where $\mathcal{R}$ is a field: This is a function from non-singular $n \times n$ matrices and n dimensional vectors that maps matrix A and vector b into the solution x to the matrix equation $Ax = b$.

We consider three types of dynamic matrix operations.

- *Column operations*:

column-update(i, v_i)**:** change $(A)^i$ — the i-th column of A to v_i;

column-update-linear-system(i, v_i)**:** change $(A)^i$ to v;

vector-update-linear-system(v)**:** change b to v;

query-inverse(i)**:** return $(A^{-1})_i$;

query-adjoint(i)**:** return $(\text{adj}(A))_i$;

query-linear-system: return $A^{-1}b$.

Row operations are defined in a similar way.

- *Simple operations*:

  update(i, j, x)**:** change $A_{i,j}$ to x;

  matrix-update-linear-system(i, j, x)**:** change $A_{i,j}$ to x;

  vector-update-linear-system(i, x)**:** change b_i to x;

  query-inverse(i, j)**:** return $(A^{-1})_{i,j}$;

  query-determinant: return $\det(A)$;

  query-adjoint(i, j)**:** return $(\text{adj}(A))_{i,j}$;

  query-linear-system(i)**:** return $(A^{-1}b)_i$.

- *Thick operations*:

  update$((i_1, v_1), \ldots, (i_k, v_k))$**:** change columns i_p to vectors v_p, for all $1 \leqslant p \leqslant k$;

  query-inverse($i_1, \ldots, i_k$**,**$j_1, \ldots, j_k$**):** return $(A^{-1})_{i_p,j_q}$, for all $1 \leqslant p, q \leqslant k$;

  query-adjoint($i_1, \ldots, i_k$**,**$j_1, \ldots, j_k$**):** return $(\text{adj}(A))_{i_p,j_q}$, for all $1 \leqslant p, q \leqslant k$.

The problem of solving dynamic linear systems of equations was not considered in [16]. However, it is easy to establish an $\Omega(n)$ lower bound for the problem in the RAM model. The computation of matrix inverse can be reduced to solving a linear system of equations. Suppose we want to dynamically compute the inverse of a matrix A. In the linear system of equations problem we maintain the matrix A as the coefficient matrix. When we want to query the (i, j) entry of the inverse we do the following:

- set the j-th entry of the vector b to 1 and the other elements to zero,
- query the i-th entry of the solution vector.

We have:

$$x_i = e_i^T A^{-1} b = e_i^T A^{-1} e_j = A^{-1}_{i,j}.$$

Thus, the element of the inverse can be computed by querying the solution x.

The lower bounds in [16] apply only to simple operations. Using the following result of Savage [47] we can prove a lower bound in the case of column operations, in the model of stright line programs.

Lemma 2.1.1 (Savage '74)**.** *There exist $n \times n$ matrices A over a field F such that computing $x \to Ax$ with straight line program requires $\Omega(\frac{n^2}{\log_{|F|} n})$ operations.*

This bound can be used to prove the following.

Lemma 2.1.2. *A dynamic straight line program for matrix inverse requires in the worst case $\Omega(n^2)$ arithmetic operations to support column operations.*

Proof. Let A be a matrix satisfying Lemma 2.1.1 and x be an n dimensional vector. Consider a $3n \times 3n$ matrix X defined as follows

$$X = \begin{bmatrix} I & A & 0 \\ 0 & I & xe_n^T \\ 0 & 0 & I \end{bmatrix}.$$

The inverse of X is equal to

$$X^{-1} = \begin{bmatrix} I & -A & Axe_n^T \\ 0 & I & -xe_n^T \\ 0 & 0 & I \end{bmatrix}.$$

Notice that we can read out the vector Ax from the last column of the inverse with column query and in order to change x we need column update of X. Thus Lemma 2.1.1 implies that we need $\Omega(n^2)$ arithmetic operations either for the update or for the query. $\square$

For the definition of dynamic straight line programs see [16]. All of our algorithms fall in this class. In the case of column operation, the algorithms from Section 4.1 are optimal because they support column updates in $O(n^2)$ operations. Note that this corollary does not rule out the possibility of devising asymptotically faster amortized algorithms and also the possibility of devising faster algorithms in the RAM model.

Our Results

The first problem to be solved in designing dynamic algorithms for algebraic functions is a choice of information which should be maintained during the updates. On the one hand, this information should be large enough to allow answering queries fast, but on the other hand it should not be too large if we want to update it fast. It appears that the right choice is to construct a data structure for maintaining the inverse of the matrix. We show that with the use of this data structure the updates and the queries can be computed efficiently.

We assume that the matrix remains non-singular throughout the updates. The algorithms can be modified to the case when the matrix can become singular but the complexity of our algorithms is then much worse. The problem of handling singular matrices without increasing the operation costs remains open. However, for our applications the algorithms supporting only non-singular updates are sufficient.

For row and column updates we devise an algorithm supporting updates in worst-case $O(n^2)$ arithmetic operations. The algorithm maintains the inverse of the matrix explicitly and thus answers queries in $O(1)$ operations. Since an update can change as many as $\Omega(n^2)$ entries of the the matrix, this algorithm seems to be the fastest possible assuming answering queries in constant number of operations. Although we consider here a more general standard RAM model, our algorithm is a straight line program and it performs as fast as Lemma 2.1.2 allows. Next we extend the algorithm to support so called *thick updates*, i.e. updates that are allowed to change n^ϵ rows or columns of the matrix, for arbitrary $\epsilon \in [0, 1]$. This new algorithm requires $O(n^{\omega(1,\epsilon,1)})$ arithmetic operations per update.

In order to break through the $O(n^2)$ barrier, we restrict ourself only to *simple operations*. We show two trade-off algorithms for this problem. Both algorithms maintain the inverse in a lazy form, what allows to perform updates faster but increases the query time. The first algorithm uses $O(n^{\omega(1,\epsilon,1)-\epsilon} + n^{1+\epsilon})$ operations (worst-case) for updates, $O(1)$ operations to query the determinant and the solution of linear system of equations, and $O(n^\epsilon)$ operations to query the adjoint matrix and the inverse. The second algorithm uses $O(n^{\omega(1,\epsilon,1)-\epsilon} + n^{2\epsilon})$ operations (worst-case) for updates, $O(1)$ operations to query the determinant and the solution of linear system of equations, and $O(n^{2\epsilon})$ operations to query the adjoint matrix and the inverse. Using the best known bounds on $\omega(1, \epsilon, 1)$ and minimizing the update cost, we get algorithms supporting updates in $O(n^{1.575})/O(n^{1.495})$ operations and queries in $O(n^{0.575})/O(n^{1.495})$ operations.

Next, we consider the case of dynamic computing the matrix inverse over rings. Strassen [51] gave an $O(n^{\omega+1})$ time algorithm for computing matrix determinant without divisions. We show here that his technique can also be used in the case of dynamic computations. Moreover, we also show special algorithms for the case when some notion of divisibility can be used. We devise faster algorithms for Euclidean rings, which include polynomials and integers.

2.2 Dynamic Graph Algorithms

A dynamic graph algorithm maintains actual information on a given property P of a graph subject to dynamic change of the graph. Possible changes include edge insertion, edge deletion or edge weight update. Any dynamic graph algorithm for property P should process updates and after each update, it should be able to answer queries on P. In case of dynamic graph algorithms, we say that an algorithm is *incremental* if it supports insertions only, and *decremental* if it supports deletions only. We maintain a graph $G = (V, E)$ under intermixed sequence of **insert**(e) and **delete**(e) operations. Updates of this type are called *edge* updates throughout the thesis. We also consider dynamic algorithms for generalized updates called *v-centered* updates. Let E_v denote an arbitrary set of edges incident with a common vertex v. The operation v-**insert**(E_v) inserts E_v into the edge set E and v-**delete**(E_v) deletes E_v from the edge set E.

In the case of undirected graphs, we show algorithms supporting the following types of queries:

- **test-matching**(e): test if an edge e is contained in any perfect matching, i.e., test if e it is *allowed*, assuming that a graph has a perfect matching;
- **spanning-trees**: compute the number of spanning trees.

To the best of my knowledge, no dynamic solutions to the above problems have been known until now.

In the case of directed graphs, we consider the following problems:

- **paths-DAG**(v,w): return the number of distinct paths from v to w, assuming that the graph is acyclic;
- **reachability**(v,w): test whether there is a path from v to w;
- **distance**(v,w): return the distance between v and w.

Any algorithm for dynamic counting the number of paths in DAG has worst-case complexity $\Omega(n)$ per operation. It follows from the fact that counting paths in DAGs can be used to multiply matrices (for details see [1]). This is a strong evidence that the lower bound of $\Omega(n)$ for the dynamic transitive problem can also hold.

Savege [47] in his result about hard matrices assumed only that addition is associative. Thus, his result can be also used to obtain a lower bounds for boolean matrix-vector multiplication. This implies the following observation.

Corollary 2.2.1. *A dynamic straight line program for transitive closure, requires in the worst case $\Omega(\frac{n^2}{\log n})$ arithmetic operations to support column operations.*

Similarly to Lemma 2.1.2, this does not rule out the possibility of devising asymptotically faster amortized algorithms.

2.2.1 Dynamic Transitive Closure

The problem of dynamic transitive closure has been studied by many researchers. The first papers on this subject described only incremental and decremental algorithms [25, 26, 30, 54, 22]. In the case of incremental updates, the fastest algorithms were presented by Italiano [26], La Poutré and van Leeuwen [30]. Their algorithms need $O(n)$ amortized time per insertion and $O(1)$ time per query. The case of decremental algorithms seems to be inherently harder. An $O(n)$ amortized time algorithm is known only in the case of acyclic graphs. In the case of general graphs only an $O(n^2)$ time algorithm [25] was known till the paper of Henzinger and King who presented the randomized algorithm with query time $O(n/\log n)$ and amortized update time $O(n \log^2 n)$.

The first fully dynamic transitive closure algorithm was given by Henzinger and King [22]. They devised a Monte Carlo algorithm with one-sided error, $\tilde{O}(nm^{0.58})$ amortized time per update, and $\Theta(n/\log n)$ time per query. Khanna, Motwani and Wilson [27] showed that when a lookahead of $\Theta(n^{0.18})$ updates is permitted, there exists a deterministic algorithm with update time $\Theta(n^{2.18})$. Next, King and Sagert [29] devised an algorithm for general directed graphs supporting queries in $O(1)$ time and updates in $O(n^{2.26})$ time. They also showed an algorithm for acyclic graphs with $O(n^2)$ update time. These bounds were improved by King [28], who presented a deterministic algorithm with $O(n^2 \log n)$ amortized update time and $O(1)$ query time. In [9], Demetrescu and Italiano improved these bounds further by presenting an algorithm with $O(n^2)$ amortized update time and $O(1)$ query time for general digraphs. Similar result was later presented in [42]. In [9], the authors also gave the first algorithm with subquadratic update time for directed acyclic graphs. This algorithm can answer queries in $O(n^\epsilon)$ time and perform updates in $O(n^{\omega(1,\epsilon,1)-\epsilon} + n^{1+\epsilon})$ time, for any $\epsilon \in [0, 1]$, where $\omega(1, \epsilon, 1)$ is the exponent of the matrix multiplication of an $n \times n^\epsilon$ matrix by an $n^\epsilon \times n$ matrix. The current best bounds on $\omega(1, \epsilon, 1)$ [24] imply an $O(n^{0.575})$ query time and $O(n^{1.575})$ update time. This subquadratic algorithm is randomized with one-sided error. More recently, Roditty and Zwick [43] presented an algorithm with $O(m\sqrt{n})$ update time and $O(\sqrt{n})$ query time, and another [44] with $O(m + n \log n)$ update time and $O(n)$ query time.

Notice that an update may change $\Omega(n^2)$ entries of the transitive closure matrix, and assuming that queries are answered in $O(1)$ time, the bound of $O(n^2)$ worst-case time seems to be the best we can hope for. Also, the Corollary 2.2.1 gives the evidence that this bound might hold. However, no such algorithm has been presented so far, in particular the algorithms in [9, 42] have only amortized time bounds. In dynamic setup, we are often interested in answering each query in a small worst-case time, therefore devising such algorithms might be very useful.

Our Results

Most of the algorithms mentioned above contain (except [28, 43, 44]) a fast matrix multiplication as a subroutine. However, only the algorithm of Demetrescu and Italiano [9] explores the equivalence between matrix multiplication and transitive closure. The algorithm uses explicitly the formula for transitive closure (for definition see Section 3.3)

$$A^c = \sum_{i=0}^{i=n} A^i.$$

Using algorithms for dynamic evaluation of polynomials over matrices for this formula, the authors obtained dynamic algorithms for transitive closure. In the paper [46] we present a novel technique. We show that the transitive closure can be computed by inverting an appropriately defined adjacency matrix of the graph. Thus, algorithms for dynamic matrix inverse can be directly applied to the problem of dynamic transitive closure. Edge updates of the graph are translated into simple updates of the matrix and v-centered updates into row and column updates. Using the thick updates we can also show how to deal faster with the case when updates can change edges incident to a subset of vertices. Our reduction is randomized with one-sided bounded error. The size of the arithmetic is $O(\log n)$ and hence the bounds on arithmetic operations for dynamic matrix inverse are translated into time bounds for dynamic transitive closure. This gives the first algorithm with $O(n^2)$ worst-case update time and $O(1)$ query time. We also show two algorithms breaking the $O(n^2)$ barrier for general directed graphs. The first one achieves $O(n^{1.575})$ update time and $O(n^{0.575})$ query time, and generalizes the result from [9] for DAGs. The second one has even faster $O(n^{1.495})$ update time at the cost of much higher $O(n^{1.495})$ query time.

2.2.2 Dynamic Shortest Paths

The study of algorithms for dynamic maintenance of shortest paths was started more than 35 year ago [31, 36, 41]. Since then many algorithms with times comparable to evaluating everything from scratch have been proposed [14, 18, 19, 38, 39, 45].

The first decrease-only algorithm was proposed by Ausiello *et al.* [2]. The algorithm worked for graphs with integer weights less than some integer C and needed $O(Cn \log n)$ amortized time per edge insertion.

Next, two dynamic algorithms for planar graphs have been developed. Henziger *et al.* [23] developed an $O(n^{\frac{4}{3}} \log(nC))$ time algorithm in the case of integer weights. The second algorithm was proposed by Fakcharoemphol and Rao in [15] who showed how to support queries and updates in $O(n^{\frac{4}{5}} \log^{\frac{13}{5}} n)$ amortized time.

The first fully dynamic algorithm for general graphs in the case of integer weights was given by King [28]. The running time of the algorithm is $O(n^{2.5}\sqrt{C \log n})$

per update. In the case of real edge weights similar results was obtained by Demetrescu and Italiano in [10, 11]. They assumed that there are at most S different real edge weights and obtained an algorithm supporting updates in $O(n^{2.5}\sqrt{S\log^3 n})$ time and queries in constant time. They also presented two families of trade-off algorithms that have smaller update time but at the cost of bigger query time. In the case of unweighted graphs, Baswana, Hariharan and Sen [3] developed simpler deletion only algorithms. The final step in obtaining $\tilde{O}(n^2)$ update time was made by Demetrescu and Italiano in [12].

The shortest paths problem is harder than the transitive closure problem. Similarly to the transitive closure problem, assuming $O(1)$ queries, the bound of $O(n^2)$ on the worst-case time for updates seems to be the best we can hope for. Thus, it is interesting to know if allowing greater query time one can reduce the update time below $O(n^2)$. Until now, no such algorithm has been known.

Our Results

Using an algorithm for dynamic matrix inverse over rings we have developed an algorithm for computing lengths of the shortest paths with at most k edges. This algorithm, combined with the standard technique of path decomposition, gives an algorithm supporting updates in $O(n^{1.932})$ time and queries in $O(n^{1.288})$ time. This result resolves the open question (see e.g. [10, 11, 12]) if there exist algorithms with sub-quadratic update and query times. The problem of dynamic single source shortest distances seems inherently simpler than dynamic all pair shortest distances, but till now the best solution for this problem was evaluating everything again from the scratch, and this takes $O(n^2)$ time. We have also resolved the question whether more efficient algorithms for this problem exist. However, our result is only of theoretical importance, the $\tilde{O}(n^2)$ algorithm from [12] is surely practically more efficient.

2.3 Perfect Matchings and Spanning Trees

To the best of my knowledge, no dynamic algorithms for testing whether a graph has a perfect matching and counting spanning trees in graph have been known until now. In this thesis we present the first dynamic algorithms for both problems.

The algorithms for maintaining matrix inverse dynamically can be directly applied to the problems of counting spanning trees and testing if an edge is allowed in a graph having a perfect matching. In the case of counting spanning trees, we use the Kirchoff theorem, and in the case of the matching problem, we use the result of Rabin and Vazirani [37]. They showed that the inverse matrix of the adjacency matrix encodes allowed edges in the graph. The inverse can be computed with $O(\log n)$ bit arithmetic, what allows us to construct two algorithms:

- the first one supporting queries in $O(n^2)$ time and updates in $O(1)$ time,
- the second one supporting queries in $O(n^{1.575})$ time and updates in $O(n^{0.575})$ time.

The first algorithm can be used to construct a very simple algorithm for computing perfect matchings in graphs in $O(n^3)$ time, whereas the second allows to find perfect matchings in $O(n^{2.575})$ time. This approach has been explored more deeply by the author in the papers with Marcin Mucha [35, 34], where two algorithms were shown: an $O(n^\omega)$ time algorithm for finding maximum matchings in general graphs and an $O(n^{\frac{\omega}{2}})$ time algorithm for finding maximum matchings in planar graphs. These results improve over the fastest previously known algorithms of Micali and Vazirani [33], Blum [4], and Gabow and Tarjan [20], that worked in $O(n^{2.5})$ time.

Chapter 3

Definitions and Preliminaries

In this chapter we introduce some basic definitions and facts from algebra (Section 3.1), linear algebra (Section 3.2), and from graph theory (Section 3.3). The contents of these sections are meant only for reference and for this reason we give them in the form of compact lists. The next section contains the basic facts on matrix algorithms (Section 3.4). In the last section of this chapter we describe a randomization method used in all our algorithms (Section 3.5).

3.1 Algebra

div – the integer division, $a \text{ div } b = \lfloor \frac{a}{b} \rfloor$.

Γ_n – the set of n element permutations.

sgn(p) – the sign of the permutation p,

$$\operatorname{sgn}(p) = \begin{cases} 1, & \text{if } p \text{ is even,} \\ -1, & \text{if } p \text{ is odd.} \end{cases}$$

Z_p – a finite field modulo prime number p.

$R[[u]]$ – the ring of the formal power series over a ring R. The elements of $R[[u]]$ are of the form $a_0 + a_1u + a_2u^2 + \ldots + a_iu^i + \ldots$, where $a_0, a_1, a_2, \ldots, a_i, \ldots \in R$.

$R^n[[u]]$ – the ring of the formal power series modulo u^{n+1}.

uR – the set $\{ux : x \in R\}$, where R is a ring R.

3.2 Linear Algebra

$R^{n\times m}$ – the set of $n \times m$ matrices with entries from R. For a matrix $A \in R^{n\times m}$, the row index set is $\{1, \ldots, n\}$ and the column index set is $\{1, \ldots, m\}$.

$a_{i,j}$ – the element in the i-th row and in the j-th column of matrix A.

$(\cdot)_{i,j}$ – the element in the i-th row and in the j-th column of the matrix resulting from an expression $(\cdot)$.

$\det(A)$ – the determinant of the matrix A,

$$\det(A) = \sum_{\rho \in \Gamma_n} \operatorname{sgn}(\rho) \prod_{k=1}^{n} a_{k,\rho_k}.$$

$A^{i,j}$ – the $(n-1) \times (n-1)$ matrix resulting from the deletion of the j-th row and the i-th column of the $n \times n$ matrix A.

$\operatorname{adj}(A)$ – the adjoint matrix, or shortly adjoint, of the matrix A. Adjoint is the $n \times n$ matrix, where $\operatorname{adj}(A)_{i,j} = (-1)^{i+j} \det(A^{j,i})$.

A^T – the transpose matrix of A, i.e. $(A^T)_{i,j} = a_{j,i}$.

I_n – the $n \times n$ identity matrix from $R^{n \times n}$. If there is no danger of confusion, we will write I instead of I_n.

A^{-1} – the inverse matrix of the matrix A, i.e. the matrix such that $A^{-1}A = I$ and $AA^{-1} = I$.

$(A)^i$ – the i-th column of A.

A_i – the i-th row of A.

e_i – n dimensional zero vector except 1 on the i-th place.

x_i – the i-th element of the vector x.

$A_{R,C}$ – the submatrix of A corresponding to rows from the set R and columns from the set C.

linear system of equations – given by a matrix A and a coefficient vector b. The solution to the system is a vector x, satisfying $Ax = b$.

3.3 Graphs

graph – Graph G is a pair (V, E), where V is a finite set of vertices, and E is a set of edges between the vertices — $E = \{(u,v) | u, v \in V\}$. If the graph is undirected, the adjacency relation defined by the edges is symmetric, or $E = \{\{u,v\} | u, v \in V\}$ (set of pairs of vertices rather than ordered pairs).

bipartite graph – A bipartite graph G is defined as a triple (V, W, E), where V and W are disjoint sets of vertices, and E is a set of edges between vertices from these two sets – $E = \{(v, w)|v \in V, w \in W\}$. If the graph is undirected, the adjacency relation defined by the edges is symmetric or $E = \{\{u, v\}|u, v \in V\}$.

adjacency matrix – Let $G = (V, E)$ be a graph, where $V = \{v_1, \ldots, v_n\}$. The adjacency matrix of G is the $n \times n$ matrix $A = A(G)$ such that,

$$a_{i,j} = \begin{cases} 1, & \text{if } (v_i, v_j) \in E, \\ 0, & \text{otherwise.} \end{cases}$$

$\tilde{A}(G)$ – Let $G = (V, E)$ be a graph and $V = \{v_1, \ldots, v_n\}$. A *symbolic adjacency matrix* of G is the $n \times n$ matrix $\tilde{A}(G)$ such that

$$\tilde{A}(G)_{i,j} = \begin{cases} x_{i,j} & \text{if } (v_i, v_j) \in E \text{ and } i < j, \\ -x_{i,j} & \text{if } (v_i, v_j) \in E \text{ and } i > j, \\ 0 & \text{otherwise,} \end{cases}$$

where $x_{i,j}$ are unique variables corresponding to the edges of G.

$\tilde{B}(G)$ – Let $G = (V, W, E)$ be a bipartite graph, $V = \{v_1, \ldots, v_n\}$, and $W = \{w_1, \ldots, w_n\}$. A *bipartite symbolic adjacency matrix* of G is the $n \times n$ matrix $\tilde{B}(G)$ such that

$$\tilde{B}(G)_{i,j} = \begin{cases} x_{i,j} & \text{if } (v_i, w_j) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

where $x_{i,j}$ are unique variables corresponding to the edges of G.

$G(A)$ – Let A be an $n \times n$ matrix. By $G(A)$ we denote the n-vertex graph (V, E) such that $V = \{v_1, \ldots, v_n\}$ and $(v_i, v_j) \in E$ if and only if $A_{i,j} \neq 0$.

A^c**, transitive closure** – Let $G = (V, E)$ be a directed graph, such that $V = \{v_1, \ldots, v_n\}$. The *transitive closure* A^c of G is the $n \times n$ matrix A such that $A_{i,j} \neq 0$ iff there is a path from v_i to v_j in G.

A^d**, distance matrix** – Let $G = (V, E)$ be a directed graph and $V = \{v_1, \ldots, v_n\}$. The *distance matrix* A^d for G is the $n \times n$ matrix A such that $A_{i,j}$ is the distance from v_i to v_j in G, i.e., the length of the shortest path from v_i to v_j in G.

$\tilde{O}$ – so called Soft-O. We write $f(n) = \tilde{O}(g(n))$ if $f(n) = O(g(n) \log^k g(n))$, for some constant k. Essentially, it is Big-O, ignoring logarithmic factors.

3.4 Matrix Multiplications

Designing our algorithms we use the fast rectangular matrix multiplication. Let $\omega(a, b, c)$ denote the exponent of the multiplication of an $n^a \times n^b$ matrix by an $n^b \times n^c$

matrix. For $a = b = c = 1$ we get the exponent of the square matrix multiplication $\omega = \omega(1,1,1)$.

In his seminal paper Strassen [50] showed that square matrices can be multiplied in $o(n^3)$ time. His bound has been improved many times since then. The best current bound is due to Coppersmith and Winograd [8], $\omega \leqslant 2.376$. Their ideas were latter extended by Coppersmith [7], who showed that it is possible to multiply an $n \times n^{0.294}$ matrix by an $n^{0.294} \times n$ matrix with $\tilde{O}(n^2)$ arithmetic operations.

Let $\alpha = \sup\{a : \omega(1,a,1) = 2 + o(1)\}$. The value of α is not larger then 0.294. By combining bounds on α and ω, Huang and Pan [24] showed the following bound on $\omega(a,b,c)$.

Lemma 3.4.1 (Huang and Pan [24]). *Let* $1 \geqslant a \geqslant b$. *Then*

$$\omega(1,a,b) \leqslant \hat{\omega}(1,a,b) = \begin{cases} 1+a & \text{if } 0 \leqslant b \leqslant \alpha a, \\ 1+\gamma a+\delta b & \text{if } \alpha a \leqslant b \leqslant 1, \end{cases}$$

where $\gamma = \frac{1+\alpha-\alpha\omega}{1-\alpha} \leqslant 0.844$ *and* $\delta = \frac{\omega-2}{1-\alpha} \leqslant 0.533$.

Notice that $\gamma + \delta = \omega - 1$, and so.

Corollary 3.4.2. *For any* $0 \leqslant a \leqslant 1$,

$$\omega(1,a,a) \leqslant a(\omega-1)+1.$$

This bound can also be proved by partitioning the $n \times n^a$ matrix into $n^a \times n^a$ matrices and using Coppersmith and Winograd algorithm to multiply them.

We also use the following lemma due to Bunch and Hopcroft [5].

Lemma 3.4.3 (Bunch and Hopcroft [5]). *The matrix inverse complexity is the same as the square matrix multiplication complexity — $O(n^\omega)$ arithmetic operations are sufficient to compute the matrix inverse.*

3.5 Testing Polynomials

Almost always when we apply algebraic methods to construct graph algorithms, the problem is reduced to testing if some polynomial is non-zero. For example, if we want to check if a graph has a perfect matching, then we check if appropriately defined adjacency matrix is non-singular. In order to verify that the matrix is non-singular, we compute its determinant, which is a polynomial. In this case we cannot use symbolic computation because the polynomial may have exponentially many terms and it would give an exponential time algorithm. The following lemma due to Zippel [55] and Schwartz [49] can be used to overcome this obstacle.

Lemma 3.5.1. *If $p(x_1, \ldots, x_m)$ is a non-zero polynomial of degree d with coefficients in a field and S is a subset of the field, then the probability that p evaluates to 0 on a random element $(s_1, s_2, \ldots, s_m) \in S^m$ is at most $d/|S|$. We call such event false zero.*

Corollary 3.5.2. *If a polynomial of degree n is evaluated on random values modulo prime number p of length $(1+c)\log n$, then the probability of false zero is at most $\frac{1}{n^c}$, for any $c > 0$.*

Note that in the standard computation model with word size $O(\log n)$, the finite field arithmetic modulo p can be realized in constant time. We will use this fact to establish the time bounds for our algorithms for **test-matching** problem and **reachability** problem. You should also notice that our algorithms will be randomized with one-sided error.

Chapter 4

Dynamic Determinant

4.1 Dynamic Matrix Algorithms: Row and Column Updates

4.1.1 Dynamic Matrix Inverse

In this section we present an algorithm for dynamic matrix inverse problem supporting row and columns updates in $O(n^2)$ algebraic operations. The algorithm maintains the inverse matrix explicitly and is able to answer simple queries in constant time.

The matrix A after the update of column i to a vector v is given by $A' = A + (v - (A)^i)e_i^T$, where e_i is the i-th versor. We assume that the matrix remains non-singular during updates, then we can use Procedure 1 or Procedure 2 for updating the inverse and Procedure 3 for querying the inverse.

Procedure 1 recomputes the inverse of A after a column update.

```
COLUMN-UPDATE-INVERSE($v, i$):
{maintained data: $A$, $A^{-1}$ }
    • compute $b = A^{-1}(v - (A)^i)$,
    • compute $B^{-1} = (I + be_i^T)^{-1}$,
    • update $A^{-1} := B^{-1}A^{-1}$,
    • update $A := A + (v - (A)^i)e_i^T$.
```

Let us verify that the procedure COLUMN-UPDATE-INVERSE works correctly. Similarly we can show correctnes of the ROW-UPDATE-INVERSE procedure.

We have:

$$B^{-1}A^{-1} = (I + be_i^T)^{-1}A^{-1} =$$

$$= (A(I + be_i^T))^{-1} = (A + Abe_i^T)^{-1} =$$

Procedure 2 recomputes the inverse of A after a row update.

```
ROW-UPDATE-INVERSE($v, i$):
{maintained data: $A$, $A^{-1}$ }
    • compute $b^T = (v^T - (A)_i)A^{-1}$,
    • compute $B^{-1} = (I + b^T e_i^T)^{-1}$,
    • update $A^{-1} := A^{-1}B^{-1}$,
    • update $A := A + e_i(v^T - (A)_i)$.
```

Procedure 3 returns the element of the inverse of A

```
QUERY-INVERSE($i, j$):
{maintained data: $A$, $A^{-1}$ }
    • return $A^{-1}_{i,j}$.
```

$$= (A + AA^{-1}(v - (A)^i)e_i^T)^{-1} =$$

$$= (A + (v - (A)^i)e_i^T)^{-1} = A'^{-1}.$$

Indeed, after the update, the inverse is computed correctly.

The computation of the vector b in the first step of the procedure takes $O(n^2)$ arithmetic operations. In the second step we compute the inverse of matrix B, which is given by:

$$B^{-1} = (I + be_i^T)^{-1} = \tag{4.1}$$

$$= \begin{bmatrix} 1 & \dots & 0 & -\frac{b_1}{1+b_i} & 0 & \dots & 0 \\ \vdots & \ddots & \vdots & \vdots & \vdots & & \vdots \\ 0 & \dots & 1 & -\frac{b_{i-1}}{1+b_i} & 0 & \dots & 0 \\ 0 & \dots & 0 & (1+b_i)^{-1} & 0 & \dots & 0 \\ 0 & \dots & 0 & -\frac{b_{i+1}}{1+b_i} & 1 & \dots & 0 \\ \vdots & & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & -\frac{b_n}{1+b_i} & 0 & \dots & 1 \end{bmatrix}.$$

Notice that this computation requires $O(n)$ arithmetic operations. Next, we have to perform multiplication $A'^{-1} = B^{-1}A^{-1}$. Because matrix B^{-1} has only $O(n)$ non-zero entries, this multiplication can be done with $O(n^2)$ arithmetic operations.

4.1.2 Dynamic Matrix Determinant

The dynamic matrix inverse algorithm can be extended for computing the determinant. We only have to update the value of the determinant after each update (see Procedure 4 and Procedure 5).

Procedure 4 recomputes the determinant of A after column updates.

```
COLUMN-UPDATE-DETERMINANT($v, i$)
{maintained data: $A$, $A^{-1}$, $\det(A)$ }
    • call COLUMN-UPDATE-INVERSE($v$,$i$),
    • update $\det(A) := \det(A)(1 + b_i)$.
```

Procedure 5 returns the determinant of A.

```
QUERY-DETERMINANT():
{maintained data: $A$, $A^{-1}$, $\det(A)$ }
    • return $\det(A)$.
```

We have $A' = A \cdot B$ and so $\det(A') = \det(A)\det(B)$. Since $B = I + be_i^T$, $\det(B) = 1 + b_i$. Thus, the determinant is recomputed correctly.

4.1.3 Dynamic Matrix Adjoint

When the matrix is non-singular, the adjoint can be easily computed with the use of the Cramer's rules $\text{adj}(A)_{i,j} = \det(A)(A^{-1})_{i,j}$. Thus, the procedures for maintaining the matrix inverse can be used to maintain the adjoint. We have only to change the query procedure in the following way (see Procedure 6).

Procedure 6 returns the element of the adjoint of A.

```
QUERY-ADJOINT($i, j$):
{maintained data: $A$, $A^{-1}$, $\det(A)$ }
    • return $\det(A)A^{-1}_{i,j}$.
```

4.1.4 Dynamic Linear System of Equations

In the case of linear system of equations, we use:

- Procedure 7 for updating the matrix,
- Procedure 8 for updating the coefficient vector,
- Procedure 9 for querying the solution vector.

Notice that the multiplication $A^{-1}b$ can be realized in $O(n^2)$ arithmetic operations and the whole update cost remains quadratic.

Procedure 7 recomputes the solution of the linear system of equations after a column in A was changed.

```
COLUMN-UPDATE-LINEAR-SYSTEM($v, i$)
{maintained data: $A$, $A^{-1}$, $b$, $x$}
    • call COLUMN-UPDATE-INVERSE($v$,$i$),
    • update $x := A^{-1}b$.
```

Procedure 8 recomputes the solution of the linear system of equations after the coefficient vector b was changed.

```
VECTOR-UPDATE-LINEAR-SYSTEM($b'$)
{maintained data: $A$, $A^{-1}$, $b$, $x$}
    • set $b := b'$,
    • update $x := A^{-1}b$.
```

4.1.5 Summary

The complexities of the above algorithms are summarized in the following theorem.

Theorem 4.1.1. *The problems of dynamic determinant, matrix inverse, matrix adjoint and solving linear system of equations, with non-singular row and column updates, can be solved with the following costs:*

- **initialization** $O(n^\omega)$ *arithmetic operations,*
- **update** $O(n^2)$ *arithmetic operations (worst-case),*
- **query** $O(1)$ *arithmetic operations (worst-case).*

4.2 Dynamic Matrix Inverse: Thick Updates

In this section we generalize the algorithms from the previous section to allow updates changing a set of columns of the matrix. Without loss of generality we can assume that we update a block of the first n^ε columns, because otherwise we can reorder the columns of A and rows of A^{-1} appropriately. Then, after the update, we can restore the previous order. This reordering costs at most $O(n^2)$ operations and will not be the dominating cost. In the case of thick update we have only to change Procedure 1 for updating the matrix inverse. The other procedures remain without changes. The new algorithm for handling updates is presented as Procedure 10. It changes the first $k = O(n^\varepsilon)$ columns to vectors $v_1, \dots, v_k$.

Let us check if the updated matrix A' is computed correctly. We have:

$$B^{-1}A^{-1} =$$

Procedure 9 returns the solution to the linear system of equations.

```
QUERY-LINEAR-SYSTEM($i$):
{maintained data: $A$, $A^{-1}$, $b$, $x$ }
    • return $x_i$.
```

Procedure 10 recomputes the inverse of A after a thick update.

```
THICK-UPDATE-INVERSE($v_1, \dots, v_k$):
{maintained data: $A$, $A^{-1}$ }
    • compute $B = I + A^{-1}\Big(v_1 - (A)^1 \Big| v_2 - (A)^2 \Big| \dots \Big| v_k - (A)^k \Big| \underbrace{0 \Big| \dots \Big| 0}_{n-k \text{ times}}\Big)$,
    • compute $B^{-1}$,
    • update $A^{-1} := B^{-1}A^{-1}$,
    • update $A := A + \Big(v_1 - (A)^1 \Big| v_2 - (A)^2 \Big| \dots \Big| v_k - (A)^k \Big| \underbrace{0 \Big| \dots \Big| 0}_{n-k \text{ times}}\Big)$.
```

$$= \left(I + A^{-1}\Big(v_1 - (A)^1 \Big| v_2 - (A)^2 \Big| \dots \Big| v_k - (A)^k \Big| \underbrace{0 \Big| \dots \Big| 0}_{n-k \text{ times}}\Big)\right)^{-1} A^{-1} =$$
$$= \left(A\Big(I + A^{-1}\Big(v_1 - (A)^1 \Big| v_2 - (A)^2 \Big| \dots \Big| v_k - (A)^k \Big| \underbrace{0 \Big| \dots \Big| 0}_{n-k \text{ times}}\Big)\Big)\right)^{-1} =$$
$$= \left(A + \Big(v_1 - (A)^1 \Big| v_2 - (A)^2 \Big| \dots \Big| v_k - (A)^k \Big| \underbrace{0 \Big| \dots \Big| 0}_{n-k \text{ times}}\Big)\right)^{-1} =$$
$$= A'^{-1}.$$

We will now look in more detail at the complexity of this procedure. We have to compute two matrix products and one matrix inverse. The multiplication

$$A^{-1}\Big(v_1 - (A)^1 \Big| v_2 - (A)^2 \Big| \dots \Big| v_k - (A)^k \Big| \underbrace{0 \Big| \dots \Big| 0}_{n-k \text{ times}}\Big),$$

requires $n^{\omega(1,\varepsilon,1)}$ arithmetic operations. In order to see how fast the inverse B^{-1} can be computed, we notice that B is a block matrix of the form:

$$B = \left[\begin{array}{c|c} B_{11} & 0 \\ \hline B_{21} & I \end{array}\right], \tag{4.2}$$

where B_{11} is an $n^\varepsilon \times n^\varepsilon$ matrix and B_{21} is an $(n - n^\varepsilon) \times n^\varepsilon$ matrix. The matrix B^{-1} is given by:

$$B^{-1} = \left[\begin{array}{c|c} B_{11}^{-1} & 0 \\ \hline -B_{21}B_{11}^{-1} & I \end{array}\right]. \tag{4.3}$$

Thus, the inverse B^{-1} can be computed in $O(n^{\omega(1,\varepsilon,\varepsilon)})$ arithmetic operations. The cost is dominated by the multiplication $B_{21}B_{11}^{-1}$.

The final multiplication $A'^{-1} = B^{-1}A^{-1}$ can be written as follows:

$$A'^{-1} = B^{-1}A^{-1} = \left[\begin{array}{c|c} B_{11}^{-1} & 0 \\ \hline -B_{21}B_{11}^{-1} & I \end{array}\right] A^{-1} = \left[\begin{array}{c|c} B_{11}^{-1} - I & 0 \\ \hline -B_{21}B_{11}^{-1} & 0 \end{array}\right] A^{-1} + A^{-1}.$$

Only the first n^{ε} columns play role in the multiplication, so the multiplication can be realized in $O(n^{\omega(1,\varepsilon,1)})$ arithmetic operations.

4.2.1 Summary

In order to solve the „thick" matrix problems, we can apply the same methods as in the previous section and get the following theorem.

Theorem 4.2.1. *The problems of dynamic determinant, matrix inverse, matrix adjoint and solving linear system of equations, with non-singular thick updates, i.e., updates of n^{ε} rows or columns, can be solved with the following costs:*

- **initialization** $O(n^{\omega})$ *arithmetic operations,*
- **update** $O(n^{\omega(1,\varepsilon,1)})$ *arithmetic operations (worst-case),*
- **query** $O(1)$ *arithmetic operations (worst-case).*

4.3 Dynamic Matrix Inverse: Simple Updates

In this section we present how to speed up the dynamic matrix algorithm by keeping the inverse in a lazy form. However, in order to support updates faster, we have to restrict ourselves only to simple operations. The lazy form of the matrix is maintained with Procedure 11.

The inverse matrix in the algorithm is kept in the following form:

$$A^{-1} = (N + I)M.$$

We first show that the update $(*)$ maintains this lazy form. Let us define:

$$N' := B^{-1}(N + I) - I.$$

It is easy to check that

$$(N' + I)M = (B^{-1}(N + I) - I + I)M =$$
$$= B^{-1}(N + I)M = B^{-1}A^{-1} = A'^{-1},$$

because $A' = AB$.

Before analyzing the cost of the above procedure we need the following observation (Lemma 4.3.1).

Procedure 11 recomutes the lazy form of the inverse matrix after a simple update.

```
UPDATE-LAZY-INVERSE($i, j, x$):
{maintained data: $M$, $N$, $A$, $k$ }
  • compute $b = (N + I)M(x - (A)_{i,j})e_i$,
  • compute $B^{-1} = (I + be_j^T)^{-1}$,
  • update $N := B^{-1}(N + I) - I$ (*),
  • update $A := A + (x - (A)_{i,j})e_i e_j^T$,
  • update $k := k + 1$,
  • if $k = n^\varepsilon$ then
      – recompute $M$ with the last $k$ updates using THICK-UPDATE-INVERSE
        procedure (**),
      – set $N := 0$,
      – set $k := 0$.
```

Lemma 4.3.1. *The matrix N has no more than k non-zero columns.*

Proof. We have

$$N' = B^{-1}(N + I) - I = B^{-1}N + B^{-1} - I.$$

The matrix $B^{-1}N$ has the same non-zero columns as N, whereas $B^{-1} - I$ has only one non-zero column. Thus, the sum may have only one non-zero column more. □

Let us now calculate the cost of n^ε updates. In order to process these updates, we have to:

- recompute the lazy form with the THICK-UPDATE-INVERSE procedure — $O(n^{\omega(1,\varepsilon,1)})$ arithmetic operations,
- compute the vector b and recompute the matrix N — this can be done in $O(n^{1+\varepsilon})$ arithmetic operations with use of the form of N from Lemma 4.3.1

Therfore, the amortized cost of one update is $O(n^{\omega(1,\varepsilon,1)-\varepsilon} + n^{1+\varepsilon})$ arithmetic operations.

The query for the matrix inverse can be realized with Procedure 12.

Procedure 12 returns the element of the inverse matrix.

```
QUERY-LAZY-INVERSE($i, j$):
{maintained data: $M$, $N$, $A$, $k$ }
  • return $e_i^T(N + I)Me_j$.
```

We need only $O(n^\varepsilon)$ arithmetic operations for the query in the worst case. The procedures for querying adjoint and determinant are similar to ones presented in

Section 4.1. However, the procedures for dynamic linear system of equations have to be modified — see Procedure 13, Procedure 14, and Procedure 15.

4.3.1 Dynamic Linear System of Equations

Procedure 13 recomputes the solution of the linear system of equations after the matrix A was changed.

```
MATRIX-UPDATE-LAZY-LINEAR-SYSTEM($i, j, a$)
{maintained data: $M$, $N$, $A^{-1}$, $b$, $x$}
    • call UPDATE-LAZY-INVERSE($i$,$j$,$a$),
    • update $x := B^{-1}x$.
```

Procedure 14 recomputes the solution of the linear system of equations after the coefficient vector b was changed.

```
VECTOR-UPDATE-LAZY-LINEAR-SYSTEM($i$,$a$)
{maintained data: $M$, $N$, $A^{-1}$, $b$, $x$}
    • set $b := b + (a - b_i)e_i$,
    • update $x := x + (N + I)M(a - b_i)e_i$.
```

Procedure 15 returns the i-th element of the solution vector.

```
QUERY-LAZY-LINEAR-SYSTEM($i$):
{maintained data: $M$, $N$, $A^{-1}$, $b$, $x$ }
    • return $x_i$.
```

4.3.2 Summary

We summarize the above results in the following theorem.

Theorem 4.3.2. *The problems of dynamic determinant, matrix inverse, matrix adjoint and solving linear system of equations, with non-singular simple updates, can be solved with the following costs:*

- **initialization** $O(n^{\omega})$ *arithmetic operations,*
- **update** $O(n^{\omega(1,\varepsilon,1)-\varepsilon} + n^{1+\varepsilon})$ *arithmetic operations (worst-case),*
- **query for inverse and adjoint** $O(n^{\varepsilon})$ *arithmetic operations (worst-case),*
- **query for determinant and linear system of equations** $O(1)$ *arithmetic operations (worst-case).*

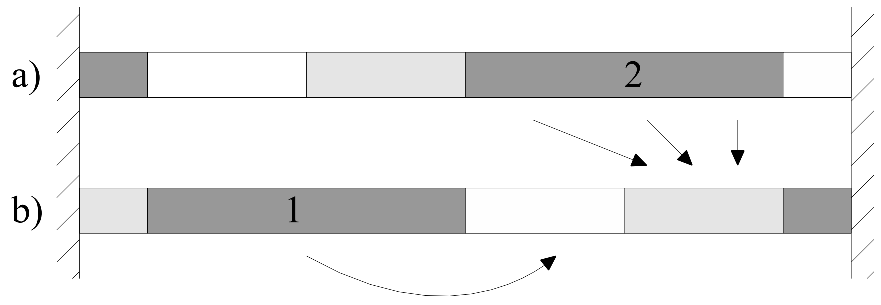


Figure 4.1: Structures a) and b) are used interchangeably. Structure a) is used to answer queries in the block 1 of operations, and structure b) answers the querries in block 2. When structure a) answers queries, structure b) is updated with the updates from the block 1, and then with the updates from block 2.

Proof. The amortized bound on update can be made worst-case by the standard technique. We keep two copies of the data structures: one is used for queries and the other is updated in the background. The updating scheme is presented in Figure 4.1. $\square$

The costs of the particular operations mentioned in the above theorem are illustrated in Figure 4.2.

In order to get the fastest possible update complexity, we balance the two terms in the bound and get $\omega(1,\varepsilon,1) = 1 + 2\varepsilon$. Applying Lemma 3.4.1 we obtain $\varepsilon < 0.575$. Thus, the update can be realized in $O(n^{1.575})$ arithmetic operations and query in $O(n^{0.575})$ arithmetic operations.

4.3.3 Maintaining a Part of the Inverse

The lazy matrix form has an interesting property which will be used in Chapter 8 in the construction of dynamic algorithms for the distance problem. It is possible to maintain in subquadratic time a part of the inverse matrix given by a subset X of the rows and a subset Y of the columns, i.e., when we are only allowed to query the matrix $(A^{-1})_{X,Y}$. If $|X| = O(n^\alpha)$ and $|Y| = O(n^\beta)$, such a dynamic matrix problem is called *α,β-restricted.*

Theorem 4.3.3. *The α,β-restricted problems of dynamic matrix inverse and matrix adjoint, with non-singular simple updates, can be solved with the following costs:*

- **initialization** $O(n^\omega)$ *arithmetic operations,*
- **update** $O(n^{\omega(1,\varepsilon,1)-\varepsilon} + n^{1+\varepsilon} + n^{\alpha+\beta})$ *arithmetic operations (worst-case),*
- **query for inverse and adjoint** $O(1)$ *arithmetic operations (worst-case).*

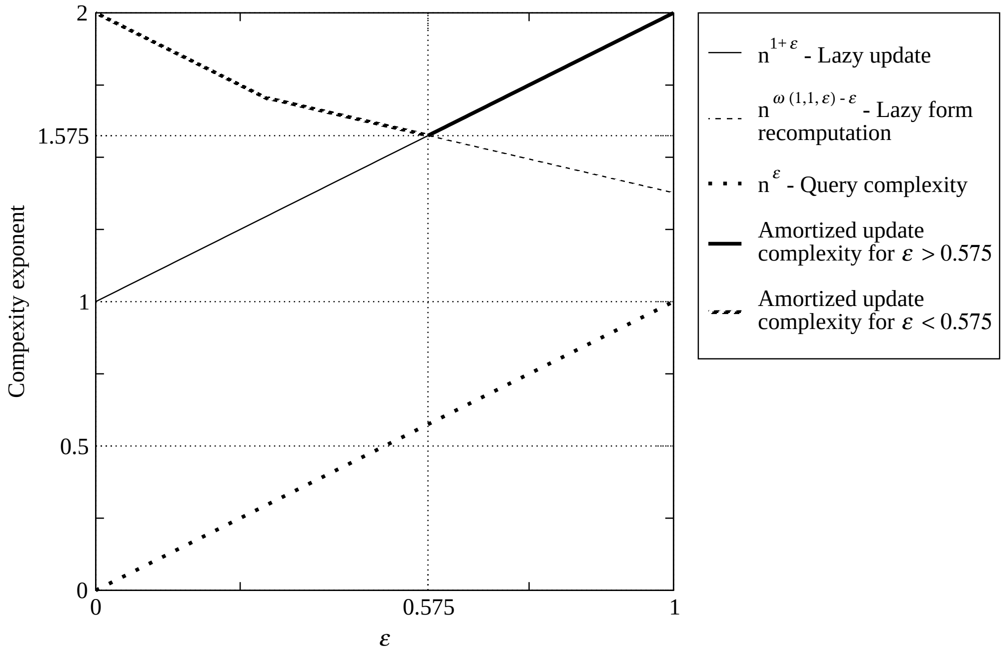


Figure 4.2: The update and query complexities prooved in Theorem 4.3.2.

Proof. It is sufficient to show how to recompute $(A^{-1})_{X,Y}$ after each update. The matrix A'^{-1} is given by

$$A'^{-1} = B^{-1}A^{-1} = (I + B^{-1} - I)A^{-1} = A^{-1} + (B^{-1} - I)A^{-1}.$$

Notice that due to a special form of B^{-1} (see (4.1)) only the i-th row of A^{-1} is used in this multiplication. In other words, we have

$$A'^{-1} = A^{-1} + ((B^{-1} - I))^{i}(A^{-1})_{i}.$$

This allows to recompute only a part of A^{-1} in the way given in Procedure 16.

Procedure 16 Recomputes the part of the inverse matrix after a simple update.

```
UPDATE-INVERSE-PART($v, i$):
{maintained data: $A$, $A^{-1}$ }
    • call UPDATE-LAZY-INVERSE($i, j, x$),
    • query out the $i$-th row of $A^{-1}$,
    • update $(A^{-1})_{X,Y} := (A^{-1})_{X,Y} + ((B^{-1} - I))_{X,\{i\}} A^{-1}_{\{i\},Y}$.
```

Querying out $(A^{-1})^{i}$ takes $O(n^{1+\epsilon})$ arithmetic operations and does not increase update cost as compared to UPDATE-LAZY-INVERSE procedure. The recomputation of the part of the inverse requires only a vector-vector multiplication what costs $O(|X||Y|) = O(n^{\alpha+\beta})$ arithmetic operations. □

4.4 Dynamic Matrix Inverse: Simple Updates II

In this section we present a different update-query time trade-off. It allows us to show algorithms with faster update time – $O(n^{1.495})$, but much higher query time – $O(n^{1.495})$.

Let $A^{(k)}$ denote the matrix after k updates. We keep matrices $N^{(l)}, B^{(l)}$ and vectors $b^{(l)}$, for all $l \leqslant k$, and a matrix M such that:

$$\begin{array}{rcl} (A^{(l)})^{-1} & = & (I + N^{l})M, \\ N^{(l+1)} & = & (B^{(l+1)})^{-1}(N^{(l)} + I) - I, \end{array} \tag{4.4}$$

In the updates and queries we compute only a part of the entries of $N^{(l)}$,$B^{(l)}$ and $b^{(l)}$, for $l \leqslant k$. The other entries are computed only when they are needed. Consider a sequence of k updates that have modified columns $j^1, \ldots, j^k$ of A. The elements in rows $j^1, \ldots, j^k$ in $N^{(l)}$, $B^{(l)}$ and $b^{(l)}$ will be computed explicitly for all $l \leqslant k$. We use Procedure 17 to recompute the matrices after each update.

Let us first check if the information we maintain is sufficient to recompute the matrix. The procedure UPDATE-LAZY-ROW-II(j, l) computes the j-th rows of $N^{(l+1)}$, $(B^{(l+1)})^{-1}$ and $b^{(l+1)}$, and needs the following prerequisites:

multiplication (1): the j-th row of $N^{(l)}$,

inversion (2): the j-th and $j^{(l+1)}$-th rows of $b^{(l+1)}$ (see the form (4.1) of B),

multiplication (3): the j-th row of $(B^{(l+1)})^{-1}$, there are only two non-zero elements in j-th row of $(B^{(l+1)})^{-1}$, so we need only the j-th and $j^{(l+1)}$-th rows of $N^{(l)}$ (again see the form (4.1) of B),

Notice that in the UPDATE-LAZY-INVERSE-II procedure, the calls are organized in such a way that the prerequisites are satisfied.

The UPDATE-LAZY-INVERSE-II procedure calls $O(k)$ times the procedure UPDATE-LAZY-ROW-II. Each call needs $O(k)$ arithmetic operations. The total cost of updates is $O(k^2) = O(n^{2\varepsilon})$. The same cost has the procedure QUERY-LAZY-INVERSE-II (Procedure 18) used to query the inverse matrix.

The amortized cost of one update is $O(n^{\omega(1,\varepsilon,1)-\varepsilon} + n^{2\varepsilon})$ arithmetic operations and the cost of query is $O(n^{2\varepsilon})$.

Theorem 4.4.1. *The problems of dynamic determinant, matrix inverse, matrix adjoint and solving linear system of equations, with non-singular simple updates, can be solved with the following costs:*

- **initialization** $O(n^{\omega})$ *arithmetic operations,*

Procedure 17 recomputes the lazy form of the inverse matrix after a simple update.

```
UPDATE-LAZY-INVERSE-II($i, j, x$):
{maintained data: $M$, $N$, $A$, $k$ }
  • $i^{(k+1)} = i$,
  • $j^{(k+1)} = j$,
  • $x^{(k+1)} = x$,
  • for $l := 1$ to $k$ do
      – call UPDATE-LAZY-ROW-II($j^{(k)}, l$),
  • for $l := 1$ to $k$ do
      – call UPDATE-LAZY-ROW-II($j^{(l)}, k$),
  • update $k := k + 1$,
  • if $k = n^{\varepsilon}$ then
      – recompute $M$ with the last $k$ updates using THICK-UPDATE-INVERSE
        procedure,
      – clear $N^{(l)}$, $B^{(l)}$, for $1 \leqslant l \leqslant k$,
      – set $k := 0$.
  • update $A_j^{(k+1)} := A_j^{(k)} + (x - (A^k)_{i,j})e_i$,

UPDATE-LAZY-ROW-II($j, l$):
  (1) compute $b_j^{(l+1)} := e_j^T(I + N^{(l)})M((A^{(l+1)})_{j^{(l+1)}} - (A^{(l)})_{j^{(l+1)}})e_{i^{(l+1)}}$,
  (2) update $(B^{(l+1)})_j^{-1} := e_j^T(I + b^{(l+1)}e_{j^{(l+1)}}^T)^{-1}$,
  (3) update $N_j^{(l+1)} := e_j^T(B^{(l+1)})^{-1}(N^{(l)} + I) - I$.
```

- **update** $O(n^{\omega(1,\varepsilon,1)-\varepsilon} + n^{2\varepsilon})$ *arithmetic operations (worst-case),*
- **query for inverse, adjoint and linear system of equations** $O(n^{2\varepsilon})$ *arithmetic operations (worst-case),*
- **query for determinant** $O(1)$ *arithmetic operations (worst-case).*

The complexities of all operations from Theorem 4.4.1 are illustrated in Figure 4.3.

Balancing both terms in the update time we get the fastest updates for $\varepsilon < 0.7471$. This gives updates and queries in $O(n^{1.495})$ arithmetic operations.

Procedure 18 computes the element of the inverse using the lazy form of the matrix.

```
QUERY-LAZY-INVERSE-II($i, j$):
  • for $l := 1$ to $k$ do
      – call UPDATE-LAZY-ROW-II($j, l$),
  • return $e_i^T(I + N^{k+1})Me_j$.
```

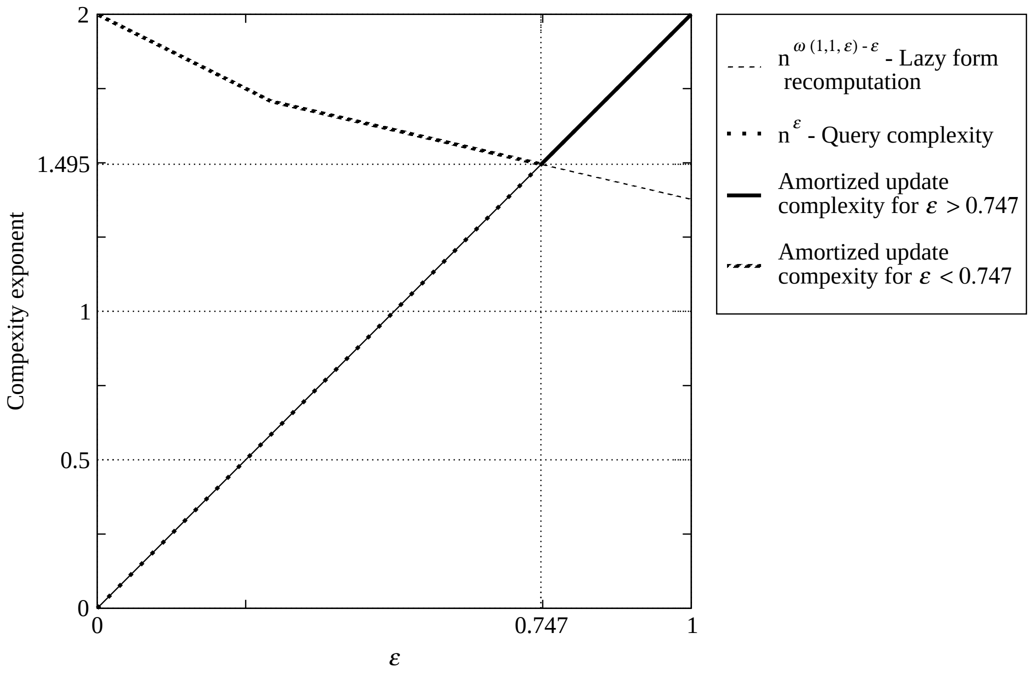


Figure 4.3: The update and query complexity in Theorem 4.4.1.

Chapter 5

Dynamic Determinants Over Rings

In this chapter we discuss the problem of computing matrix determinant and matrix adjoint for matrices over commutative rings. We show that the classic Strassen [51] technique can be extended also to the dynamic case. Strassen has shown how to compute the determinant without divisions in $\tilde{O}(n^{\omega+1})$ ring operations. This gives an $O(n)$ slowdown in comparison to the fast Gaussian elimination [5], that works in $O(n^{\omega})$ field operations. In the dynamic case this approach has the same impact on the complexity – 1 is added to the exponent.

5.1 Matrix Problems: Division-Free Algorithms

The Strassen's idea is to work in $\mathcal{R}[[u]]$ — the ring of the formal power series over a commutative ring $\mathcal{R}$. We will use the fact that in the ring of the formal power series there are some invertible elements, despite the fact that R doesn't have to contain any invertible elements.

Lemma 5.1.1. *The elements of the form $1 - z$, where $z \in u\mathcal{R}[[u]]$, are invertible.*

Proof. The inverse of $1 - z$ is given by

$$\frac{1}{1-z} = 1 + z + z^2 + \ldots = (1+z)(1+z^2)(1+z^4)\ldots.$$

Notice that computing the first n terms of the inversion requires only $O(\log n)$ multiplications in $R[[u]]$. □

Let A be a matrix over a ring $\mathcal{R}$. Define $A(u) = I + u(A - I)$. Then

$$\begin{aligned} A &= A(u)|_{u=1}, \\ \det(A) &= \det(A(u))|_{u=1}, \\ \operatorname{adj}(A) &= \operatorname{adj}(A(u))|_{u=1}. \end{aligned} \tag{5.1}$$

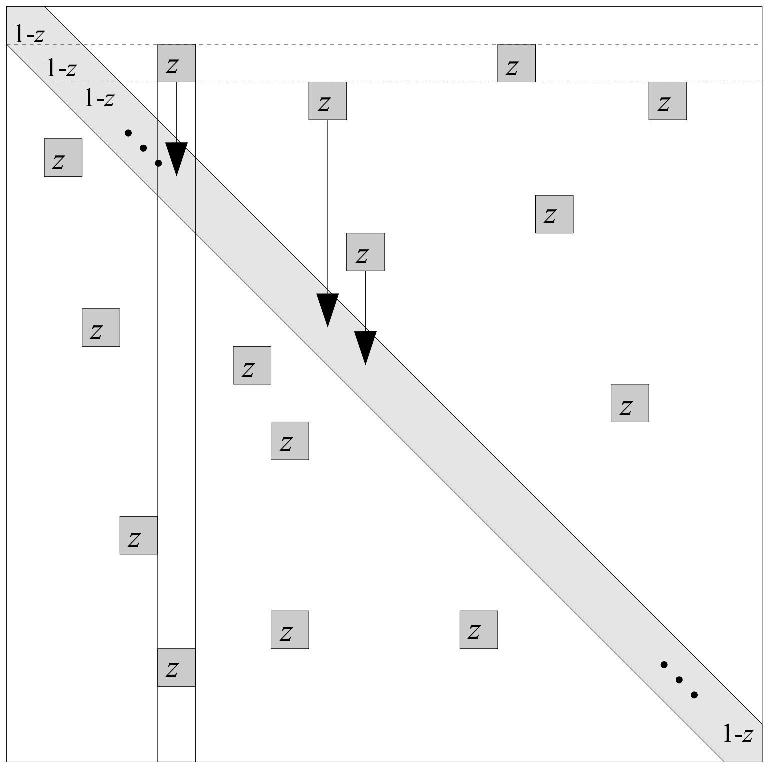


Figure 5.1: During the Gaussian elimination of $A(u)$ only invertible elements appear on the diagonal.

The determinant of the matrix $A(u)$ can be computed using the standard Gaussian elimination because the elements on the diagonal are always invertible. The elimination process is schematically presented in Figure 5.1. At the very beginning, the diagonal elements are of the form $1-z$. They can change during the elimination, but only z-part can be modified.

Matrix $A(u)$ can be used to compute the determinant of A, but the algorithm we get is completely unpractical. The operations in $R[[u]]$ cannot be carried out efficiently. However, all the computations can be carried in $R^n[[u]]$, i.e., in the ring of reminders modulo u^{n+1}. The determinant $\det(A(u))$ is a polynomial of degree n in u, so it will be computed correctly modulo u^{n+1}. Assuming that the elements of $R^n[[u]]$ can be multiplied with $O(n\log(n)\log(\log(n)))$ operations in R [48], we obtain that computing the determinant without divisions can be done in $\tilde{O}(n^{\omega+1})$ operations.

Lemma 5.1.2. *For every $n\times n$ matrix A, the matrix $A(u) = I+u(A-I)$ is invertible. The determinant $\det(A(u))$, the inverse $A(u)^{-1}$ and the adjoint $\operatorname{adj}(A(u))$ can be computed in the ring $R^n[[u]]$ in $O(n^{\omega+1})$ operations in R.*

Proof. The determinant can be computed by the Gaussian elimination as argued above. We could also use it to compute the inverse, but there is an easier way given by the following equation,

$$A(u)^{-1} = \frac{1}{I + u(A - I)} = \sum_{i=0}^{\infty} (-u(A - I))^i = \sum_{i=0}^{n} u^i (-(A - I))^i . \tag{5.2}$$

To verify it multiply both sides by $A(u)$. This gives also an $\tilde{O}(n^{\omega+1})$ algorithm for computing the inverse in $R^n[[u]]$. The adjoint can be computed by taking $\mathrm{adj}(A(u)) = \det(A(u))A(u)^{-1}$. □

Lemma 5.1.2 and the equalities (5.1) imply the following theorem.

Theorem 5.1.3 (Strassen '71)**.** *The determinant and the adjoint of a generic $n \times n$ matrix A over a commutative ring R can be computed in $\tilde{O}(n^{\omega+1})$ ring operations.*

5.2 Dynamic Matrix Problems: Division-Free Algorithms

5.2.1 Dynamic Determinant: Row and Column Updates

In this section we show how the Strasssen technique can be used in the dynamic case. The construction of the matrix $A(u)$ allowed us to use the standard Gaussian elimination for computing matrix determinants over the rings. We have only to prove that whenever the inverse is needed, it exists. We apply here the same approach. We show that all the algorithms from Chapter 4 can be used to maintain dynamically the inverse of $A(u)$. Recall that these procedures only need the inverse of the matrix B.

Lemma 5.2.1. *The matrices B from Procedure 1 and Procedure 2 are invertible, when they are used to maintain the matrix $A(u) = I + u(A - I)$.*

Proof. Let us consider the change of the matrix $A(u)$ after a column update of A. The matrix $A(u)$ is changed in the following way

$$A(u)' = I + u(A + (v - (A)^i)e_i^T - I) = A(u) + (uv - u(A)^i))e_i^T .$$

Now B is given by

$$B = I + be_i^T = I + A(u)^{-1}(uv - u(A)^i) = I + uA(u)^{-1}(v - (A)^i).$$

The inverse B^{-1} can be computed in the same way as in (5.2).

$$B^{-1} = \frac{1}{I + be_i^T} = \sum_{k=0}^{\infty} \left(-be_i^T\right)^k = \sum_{k=0}^{\infty} \left(-uA(u)^{-1}(v - (A)^i)e_i^T\right)^k . \tag{5.3}$$

Note that this inverse can be computed in $O(n^2)$ operations in $\mathcal{R}^n[[u]]$. □

In order to get efficient algorithms, we do all the computations in $\mathcal{R}^n[[u]]$. Each operation in $\mathcal{R}^n[[u]]$ require $\tilde{O}(n)$ operations in $\mathcal{R}$, so we get the following theorem.

Theorem 5.2.2. *The problems of dynamic determinant and matrix adjoint over commutative ring $\mathcal{R}$, with row and column updates, can be solved with the following costs:*

- **initialization** $\tilde{O}(n^{\omega+1})$ *ring operations,*
- **update** $\tilde{O}(n^3)$ *ring operations operations (worst-case),*
- **query** $O(1)$ *ring operations (worst-case).*

5.2.2 Dynamic Determinant: Simple Updates

The trade-off algorithms supporting simple updates presented in Section 4.3 and Section 4.4 perform the same operations as algorithms for row and column updates. The only difference is that the computations are carried out in a lazy manner. We can use them to maintain matrices over a ring in the same way as in the previous section. Hence we get the following two theorems.

Theorem 5.2.3. *The problems of dynamic determinant and matrix adjoint, over a commutative ring $\mathcal{R}$, with simple updates, can be solved with the following costs:*

- **initialization** $\tilde{O}(n^{\omega+1})$ *ring operations,*
- **update** $\tilde{O}(n^{1+\omega(1,\epsilon,1)-\epsilon} + n^{2+\epsilon})$ *ring operations (worst-case),*
- **query for the adjoint** $\tilde{O}(n^{1+\epsilon})$ *ring operations (worst-case),*
- **query for determinant** $O(1)$ *ring operations (worst-case).*

Theorem 5.2.4. *The problems of dynamic determinant and matrix adjoint, over a commutative ring $\mathcal{R}$, with simple updates, can be solved with the following costs:*

- **initialization** $\tilde{O}(n^{\omega+1})$ *arithmetic operations,*
- **update** $\tilde{O}(n^{1+\omega(1,\epsilon,1)-\epsilon} + n^{1+2\epsilon})$ *ring operations (worst-case),*
- **query for adjoint** $\tilde{O}(n^{1+2\epsilon})$ *ring operations (worst-case),*
- **query for determinant** $O(1)$ *ring operations (worst-case).*

5.3 Dynamic Matrix Problems: Euclidean rings

In the previous section we assumed that no inversions are possible, but in some cases, e.g., in the case of the ring of polynomials, there is some notion of divisibility, but in general elements might not be invertible. Such rings are called Euclidean rings. The name comes from the fact that the extended Euclidean algorithm can be carried out in any Euclidean ring.

Let us recall some basic definitions.

Definition 5.3.1. ***Integral domain*** *is a ring that is commutative under multiplication, has an identity element* $(0 \neq 1)$*, and has no divisors of* 0.

Definition 5.3.2. ***Euclidean ring*** *is an integral domain* $\mathcal{E}$ *for which a function* ν *can be defined, mapping non-zero elements of* E *to non-negative integers such that*

- *for all nonzero* $a, b \in \mathcal{E}$, $\nu(ab) \geqslant \nu(a)$,
- *if* $a, b \in \mathcal{E}$ *and* $b \neq 0$*, then there exist* $q, r \in \mathcal{E}$ *such that* $a = bq + r$ *and either* $r = 0$ *or* $\nu(r) < \nu(b)$ *(*q *is a quotient and* r *is a remainder from division).*

Examples of Euclidean rings include:

- $\mathcal{Z}$, the ring of integers. We define $\nu(n) = |n|$, the absolute value of n.
- $\mathcal{F}[x]$, the ring of polynomials over a field $\mathcal{F}$. For each nonzero polynomial p we define $\nu(p)$ to be the degree of p.
- $\mathcal{F}[[x]]$, the ring of the formal power series over a field F. For each nonzero power series f we define $\nu(f)$ as the degree of the smallest power of x occurring in f.
- Every field. Define $\nu(x) = 1$ for all nonzero x.

In the following sections we assume that in Euclidean rings the procedure that finds quotient and remainder, i.e, division procedure, can be carried out in the same time as multiplication.

5.3.1 Dynamic Determinant: Row and Column Updates

In this section we present an algorithm supporting column updates in $O(n^2)$ ring operations and queries in $O(1)$ ring operations. In the algorithm we maintain the matrix X which is equal to the adjoint of the matrix A multiplied by a scalar value $x - X = \mathrm{adj}(A)x$. The value of x is also recomputed in each update. For all these recomputations we also need the value of the determinant (see Procedure 19).

Let us consider an update of the i'th column of A to the vector v. Let X' be the matrix X after the update and let x' be the new value of x. We have

$$X' = X\,\mathrm{adj}(\det(A)I + \mathrm{adj}(A)(v - (A)_i)e_i^T) = \tag{5.4}$$

$$= x\,\mathrm{adj}(A)\,\mathrm{adj}(\det(A)I + \mathrm{adj}(A)(v - (A)_i)e_i^T) =$$

$$= x\,\mathrm{adj}(A\det(A) + A\,\mathrm{adj}(A)(v - (A)_i)e_i^T) =$$

$$= x\,\mathrm{adj}(\det(A)A + \det(A)I(v - (A)_i)e_i^T) =$$

$$= x\det(A)\,\mathrm{adj}(A + (v - (A)_i)e_i^T) = x\det(A)\,\mathrm{adj}(A') =$$

Procedure 19 recomputes the inverse of the matrix A over an Euclidean Ring after a column update.

```
COLUMN-UPDATE-ADJOINT-EUCLIDEAN-RING($v, i$):
{maintained data: $A$, $X$, $x$ }
  • compute $b = \mathrm{adj}(A)(v - (A)_i)$,
  • update $X := X\,\mathrm{adj}(\det(A)I + be_i^T)$,
  • update $x := x\det(A)$,
  • compute $y = \det(A)\det(\det(A)I + be_i^T)$,
  • update $\det(A) := y \ \mathrm{div}\ \det(A)$.
```

$$= x'X',$$

by the equality $A\,\mathrm{adj}(A) = \det(A)I$. In the recomputation of X we use the value of $\det(A)$, which is also recomputed. Let us verify if the new value is correct. We have

$$y = \det(A)\det(\det(A)I + \mathrm{adj}(A)(v - (A)_i)e_i^T) =$$
$$= \det(A\det(A) + \det(A)(v - (A)_i)e_i^T) =$$
$$= \det(A)\det(A').$$

We see that $\det(A')$ is the quotient from division of y by $\det(A)$. The elements of $\mathrm{adj}(A)$ can be determined from the equation $X_{ij} = x\,\mathrm{adj}(A)_{i,j}$ in the same way.

To answer the query we divide the element in X by x (see Procedure 20).

Procedure 20 returns element of the adjoint of A, where A is defined over an Euclidean ring.

```
QUERY-ADJOINT-EUCLIDEAN-RING($i, j$):
{maintained data: $A$, $X$, $x$ }
  • return $X_{i,j}$ div $x$.
```

The calculations in (5.4) take $O(n^2)$ ring operations. We need $O(n^2)$ operation to calculate the vector $b = \mathrm{adj}(A)(v-(A)_i)$. The matrix $B = \mathrm{adj}(\det(A)I+\mathrm{adj}(A)(v-(A)_i)e_i^T)$ can be calculated in the same cost because

$$B = \mathrm{adj}(\det(A)I + be_i^T) =$$

$$= \begin{bmatrix} \det(A)+b_i & \dots & 0 & -b_1 & 0 & \dots & 0 \\ \vdots & \ddots & \vdots & \vdots & \vdots & & \vdots \\ 0 & \dots & \det(A)+b_i & -b_{i-1} & 0 & \dots & 0 \\ 0 & \dots & 0 & 1 & 0 & \dots & 0 \\ 0 & \dots & 0 & -b_{i+1} & \det(A)+b_i & \dots & 0 \\ \vdots & & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & -b_n & 0 & \dots & \det(A)+b_i \end{bmatrix}.$$

The final multiplication takes also $O(n^2)$ ring operations because B has only $O(n)$ non-zero entries.

Theorem 5.3.3. *The problems of dynamic determinant and matrix adjoint over Euclidean ring, with non-singular row and column updates, can be solved with the following costs:*

- **initialization** $O(n^{\omega})$ *ring operations,*
- **update** $O(n^2)$ *ring operations operations (worst-case),*
- **query** $O(1)$ *ring operations (worst-case).*

Proof. The theorem follows directly from the discussion above. $\square$

Remark 1. Notice that over the Euclidean rings it is possible to use the standard Gaussian elimination algorithm, and thus the determinant and the matrix inverse can be computed in $O(n^3)$ ring operations. The idea is to multiply when we have to divide and to divide the result at the end when we have the certainty that the reminder is zero. The Bunch-Hopcroft algorithm (see Lemma 3.4.3) can also be implemented in the same manner.

5.3.2 Dynamic Determinant: Simple Updates

One might wonder why in the previous subsection we have not computed the whole matrix $\mathrm{adj}(A)$ after the update but have kept it as the matrix $X = x\,\mathrm{adj}(A)$. This is needed to keep exactly the same computation scheme as in algorithms working over fields. This allows us to use the lazy computation schemes from Chapter 4 with the algorithm from Theorem 5.3.3. As a consequence, we get the following.

Theorem 5.3.4. *The problems of dynamic determinant and matrix adjoint, over an Euclidean ring $\mathcal{E}$, with non-singular simple updates, can be solved with the following costs:*

- **initialization** $O(n^{\omega})$ *ring operations,*
- **update** $O(n^{\omega(1,\epsilon,1)-\epsilon} + n^{1+\epsilon})$ *ring operations (worst-case),*
- **query for the adjoint** $O(n^{\epsilon})$ *ring operations (worst-case),*
- **query for determinant** $O(1)$ *ring operations (worst-case).*

Theorem 5.3.5. *The problems of dynamic determinant and matrix adjoint, over an Euclidean ring $\mathcal{E}$, with non-singular simple updates, can be solved with the following costs:*

- **initialization** $O(n^{\omega})$ *arithmetic operations,*
- **update** $O(n^{\omega(1,\epsilon,1)-\epsilon} + n^{2\epsilon})$ *ring operations (worst-case),*
- **query for adjoint** $O(n^{2\epsilon})$ *ring operations (worst-case),*
- **query for determinant** $O(1)$ *ring operations (worst-case).*

Chapter 6

Dynamic Counting Spanning Trees

6.1 Spanning Trees

One of possible applications of the algorithms for dynamic determinant is the problem of dynamic computing the number of spanning trees in a graph. This problem can be solved directly by the Kirchoff's matrix-tree theorem (see ,e.g., [13]). For a given a graph G let $D(G)$ denote a diagonal matrix, where the i-th diagonal entry is the degree of v_i. The combinatorial Laplacian of G is defined as

$$L(G) := D(G) - A(G).$$

Theorem 6.1.1 (Matrix-tree theorem). *If $L(G)$ is the combinatorial Laplacian of a graph G then the number of spanning trees in G is equal to $\det(L(G)^{1,1})$.*

For a dynamic graph G we can dynamically maintain the determinant of the matrix $L(G)^{1,1}$. Using the algorithm with the fastest update time we obtain the following result.

Corollary 6.1.2. *There exists an algorithm for the problem of dynamic counting spanning-trees in a graph with the update costs*

- *$O(n^{1.495})$ arithmetic operations (worst-case) for v-centered updates,*
- *$O(n^2)$ arithmetic operations (worst-case) for edge updates.*

Counting the number of spanning trees is a way of testing the reliability of a network, see [6]. Thus, our algorithms can be used for testing faster the reliability of dynamically changing networks.

Chapter 7

Dynamic Transitive Closure

7.1 Counting Paths in DAGs

Consider a directed acyclic graph with n vertices, given by the $n \times n$ adjacency matrix A. The following lemma shows how to compute the transitive closure of the directed acyclic graph (DAG) by using the matrix inverse.

Lemma 7.1.1. *If there exists k such that $A^k = 0$ then*

$$(I - A)^{-1} = \sum_{i=0}^{k-1} A^i. \tag{7.1}$$

Proof. It is sufficient to multiply both sides of the equation by $(I - A)$. □

The element $k_{i,j}$ of the matrix $K = \sum_{i=0}^{k-1} A^i$ gives the number of paths from vertex i to vertex j of lengths less than k. In n-vertex DAG there are no paths of lengths greater than $n - 1$, so $A^n = 0$. In order to compute the number of paths in a dynamic DAG G with the adjacency matrix A, we have only to keep the inverse of the matrix $I - A$ dynamically.

Theorem 7.1.2. *If there exists an algorithm for the dynamic matrix inverse in non-singular case, with the update cost of $O(n^\alpha)$ arithmetic operations, and the query cost of $O(n^\beta)$ arithmetic operations, then there exist an algorithm for dynamic computing the number of paths in n-vertex DAGs with the same update and query costs.*

Using the above theorem, combined with Theorem 4.1.1 and Theorem 4.3.2, we get similar algorithms for reachability in DAGs to those in the papers [29, 9], i.e., we obtain algorithms supporting queries in $O(n^2)$ or $O(n^{1.575})$ time. However, our approach is different — we use the dynamic matrix inverse. Moreover, the use of the dynamic matrix inverse allows to see how to move further and construct similar algorithms for general digraphs easier.

7.2 Transitive Closure of General Digraphs

In the case of general digraphs, we of course cannot use Lemma 7.1.1, but surprisingly also in this case we can use matrix the $I - A$.

Theorem 7.2.1. *Let $\tilde{A}$ be the symbolic adjacency matrix of a digraph G. There exists a path in G from i to j iff* $\operatorname{adj}(I - \tilde{A})_{i,j}$ *is not equal to zero. Moreover,* $\operatorname{adj}(I - \tilde{A})_{i,j}$ *is non-zero over a finite field $\mathcal{Z}_p$.*

Proof. We have $\operatorname{adj}(I - \tilde{A})_{i,j} = (-1)^{i+j} \det((I - \tilde{A})^{j,i})$. Equivalently, if we take $\tilde{Z}$ to be the matrix obtained from $I - \tilde{A}$ by zeroing entries of the j'th row and the i'th column and setting the entry (j, i) to one, then $\operatorname{adj}(I - \tilde{A})_{i,j} = \det(\tilde{Z})$. Then

$$\operatorname{adj}(I - \tilde{A})_{i,j} = \sum_{\rho \in \Gamma_n} \operatorname{sgn}(\rho) \prod_{k=1}^{n} z_{k,\rho_k}. \tag{7.2}$$

The sum is a non-zero polynomial if there exists a permutation ρ such that

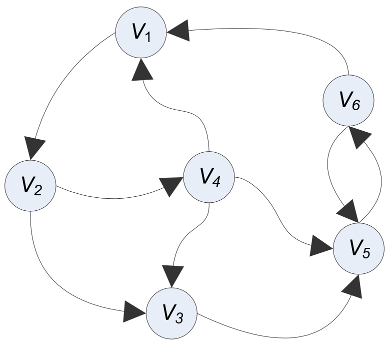
$$\prod_{k=1}^{n} z_{k,\rho_k} \neq 0.$$

The permutation can be viewed as a set of cycles. Notice that the product is non-zero if $\rho_j = i$, so in ρ there exists a cycle containing (j, i). The rest of the cycle forms a path from i to j.

On the other hand, if we have a path from i to j in G, we can add edge (j, i) to form a cycle of ρ. For vertices v not belonging to the path we take $\rho_v = v$. This permutation gives non-zero contribution to the determinant.

Notice that each monomial in (7.2) has coefficient ± 1, so $\operatorname{adj}(I - \tilde{A})_{ij}$ is non-zero over $\mathcal{Z}_p$. ∎

Example 1. Let us consider a graph presented in Figure 7.1.



$$I - \tilde{A} =$$

$$\begin{bmatrix} 1 & -x_{1,2} & 0 & 0 & 0 & 0 \\ 0 & 1 & -x_{2,3} & -x_{2,4} & 0 & 0 \\ 0 & 0 & 1 & 0 & -x_{3,5} & 0 \\ -x_{4,1} & 0 & -x_{4,3} & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -x_{5,6} \\ -x_{6,1} & 0 & 0 & 0 & -x_{6,5} & 1 \end{bmatrix}$$

Figure 7.1: An example graph and the corresponding matrix $I - \tilde{A}$.

Let us use this theorem to check whether v_3 is reachable from v_1. The matrix $\tilde{Z}$ has form

$$\tilde{Z} = \begin{bmatrix} 0 & -x_{1,2} & 0 & 0 & 0 & 0 \\ 0 & 1 & -x_{2,3} & -x_{2,4} & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -x_{4,3} & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -x_{5,6} \\ 0 & 0 & 0 & 0 & -x_{6,5} & 1 \end{bmatrix}.$$

The determinant of this matrix is equal to

$$\det(\tilde{Z}) = x_{1,2}x_{2,3} - x_{1,2}x_{2,4}x_{4,3} - x_{1,2}x_{2,3}x_{5,6}x_{6,5} + x_{1,2}x_{2,4}x_{4,3}x_{5,6}x_{6,5}.$$

The first two elements of this expression correspond to the simple paths $p = v_1 \rightarrow v_2 \rightarrow v_3$ and $q = v_1 \rightarrow v_2 \rightarrow v_4 \rightarrow v_3$. The last two correspond to p and q plus a cycle $v_5 \rightarrow v_6 \rightarrow v_5$.

Theorem 7.2.2. *If there exists an algorithm for the dynamic matrix inverse. in the non-singular case, with the update cost of $O(n^\alpha)$ arithmetic operations, and the query cost of $O(n^\beta)$ arithmetic operations, then there exists an algorithm for dynamic transitive closure with updates in $O(n^\alpha)$ time and queries in $O(n^\beta)$ time. The algorithm is randomized with one-sided error.*

Proof. The algorithm works as follows. We take a prime number p of length $\Theta(\log n)$. We substitute the non-zero entries in A for random numbers in range $1, \ldots, p-1$. Let us denote the resulting matrix B. In the algorithm we keep the inverse of the matrix $(I - B)$. We answer queries by computing $\operatorname{adj}(I - B)$ as stated in Theorem 7.2.1. Lemma 3.5.1 guarantees that the answers are correct with high probability.

In order to be able to use algorithms supporting only non-singular updates, we show that independently of the updates of A, the updates of B are non-singular with high probability. Consider the polynomial given by $\det(I - A)$ with variables being the entries of A. This polynomial is symbolically non-zero over $\mathcal{Z}_p$, due to I term. With the use of Lemma 3.5.1 we can show that during the updates the determinant $\det(I - B)$ is non-zero and matrix $(I - B)$ is invertible with high probability. If it becomes singular during updates, the algorithm can answer anything to queries. In such case, all answers may be wrong, but this happens with a small probability. In order to guarantee that each query is correctly answered, we have to refresh the random choices from time to time. The algorithm for the new choices can be initialized in parts during the updates. It follows that the algorithms for the dynamic matrix adjoint in the non-singular case can be used. □

Combining the above theorem with any of the theorems form Chapter 4, we improve the known results for dynamic transitive closure in the following ways:

Theorem 4.1.1 - gives the first algorithm that needs only $O(n^2)$ time per update in the worst-case.

Theorem 4.2.1 - gives the first algorithm supporting thick updates, i.e. we can support updates changing n^α columns in $\tilde{O}(n^2)$ time.

Theorem 4.3.2 and Theorem 4.4.1 - gives the first two algorithms supporting both updates and queries in $o(n^2)$ time.

Any improvement to dynamic matrix algorithms will directly give faster algorithms for the dynamic transitive closure.

Chapter 8

Dynamic Shortest Paths

8.1 Dynamic Shortest Paths

We are now ready to introduce an algorithm for dynamic computing the shortest path lengths in unweighted graphs.

Theorem 8.1.1. *Let $\tilde{A}$ be the symbolic adjacency matrix of a graph G. Consider* $\mathrm{adj}(I - \tilde{A}u)_{ij}$ *as the polynomial of u. The length of the shortest path in G from i to j is equal to the degree of the smallest degree term in* $\mathrm{adj}(I - \tilde{A}u)_{ij}$*. Moreover, all non-zero terms in* $\mathrm{adj}(I + \tilde{A}u)_{ij}$ *are also non-zero over a finite field $\mathcal{Z}_p$.*

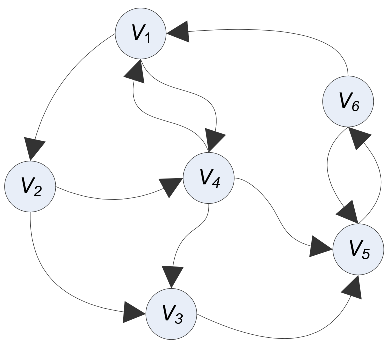
Proof. We have $\mathrm{adj}(I - u\tilde{A})_{i,j} = (-1)^{i+j} \det((I - u\tilde{A})^{j,i})$. Equivalently, if we take $\tilde{Z}$ to be the matrix obtained from $I - u\tilde{A}$ by zeroing entries of the j'th row and the i'th column and setting the entry (j, i) to one, then $\mathrm{adj}(I - u\tilde{A})_{i,j} = \det(\tilde{Z})$, and

$$\mathrm{adj}(I - u\tilde{A})_{i,j} = \sum_{p \in \Gamma_n} \mathrm{sgn}(p) \prod_{k=1}^{n} z_{k,p_k}. \tag{8.1}$$

Take a permutation p such that $\prod_{k=1}^{n} z_{k,p_k} \neq 0$. The permutation p can be viewed as a set of cycles. Notice that the above product is non-zero if $p_j = i$ for some j, so there exists a cycle c containing (j, i) in p. The rest of c forms a path from i to j. There may exist many permutations containing c that give a non-zero contribution to the sum. However, the smallest degree term is introduced by the permutation that is identity on vertices v not belonging to c, i.e., $p_v = v$. The degree of this term is the length of the path from i to j in c. Because we sum over all permutations, the smallest degree term in the sum corresponds to the shortest path.

Notice that each monomial in (8.1) has coefficient 1, so each non-zero term in $\mathrm{adj}(I - u\tilde{A})_{i,j}$ is also non-zero over $\mathcal{Z}_p$. □

Example 2. Consider a graph presented in Figure 8.1.



$$I - u\tilde{A} = \begin{bmatrix} 1 & -ux_{1,2} & 0 & 0 & 0 & 0 \\ 0 & 1 & -ux_{2,3} & -ux_{2,4} & 0 & 0 \\ 0 & 0 & 1 & 0 & -ux_{3,5} & 0 \\ -ux_{4,1} & 0 & -ux_{4,3} & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -ux_{5,6} \\ -ux_{6,1} & 0 & 0 & 0 & -ux_{6,5} & 1 \end{bmatrix}$$

Figure 8.1: An example graph and the corresponding matrix $I - u\tilde{A}$.

We use Theorem 8.1.1 to compute the distance from v_1 to v_3. The matrix $\tilde{Z}$ is of the form

$$\tilde{Z} = \begin{bmatrix} 0 & -ux_{1,2} & 0 & -ux_{1,4} & 0 & 0 \\ 0 & 1 & -ux_{2,3} & -ux_{2,4} & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -ux_{4,3} & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -ux_{5,6} \\ 0 & 0 & 0 & 0 & -ux_{6,5} & 1 \end{bmatrix}.$$

The determinant of this matrix is

$$\det(\tilde{Z}) = u^2x_{1,2}x_{2,3} - u^3x_{1,2}x_{2,4}x_{4,3} + u^2x_{1,4}x_{4,3}+$$

$$+u^4x_{1,2}x_{2,3}x_{5,6}x_{6,5} - u^5x_{1,2}x_{2,4}x_{4,3}x_{5,6}x_{6,5} + u^4x_{1,4}x_{4,3}x_{5,6}x_{6,5}.$$

Two smallest degree terms $u^2x_{1,2}x_{2,3}$ and $u^2x_{1,4}x_{4,3}$ correspond to two shortest paths from v_1 to v_3, the first one goes through vertex v_2 and the second one through vertex v_4.

The above theorem gives a connection between the matrix adjoint and the distances in the graph. In order to construct efficient algorithms, we again use the Zippel-Schwartz lemma (Lemma 3.5.1).

In order to compute distances in G dynamically, we can proceed as follows. We generate a random adjacency matrix A from the symbolic adjacency matrix $\tilde{A}$ of G by substituting each nonzero entry in $\tilde{A}$ with a random number in the range $1, \ldots, p-1$. Then we can maintain dynamically the adjoint of the matrix $I - uA$ over polynomials of degree n, and answer queries by finding the smallest degree term in $\text{adj}(I - uA)_{i,j}$. Using Theorem 5.3.4 we get an algorithm that supports updates in $\tilde{O}(n^{2.575})$ time and queries in $O(n^{0.575})$ time. In order to break the $O(n^2)$-barrier, we will use the construction presented by Demetrescu and Italiano [11] but with a better black box. The black box we need is a dynamic algorithm for computing shortest paths of length $\leqslant k$, for a given k. The above result shows that we are on the right track but we cannot use polynomials of degree k, because the intermediate results are of higher degrees. The solution would be to use the ring of the formal power series modulo a polynomial of degree $k+1$. Unfortunately, this ring is not Euclidean – it has divisors of zero, and hence we cannot use the algorithm from Theorem 5.3.4. However, we can apply the algorithms developed in Section 5.2 to maintain the inverse of the matrix $I - u\tilde{A}$ over the formal power series. One can show (as in (5.2)) that this matrix is invertible. This gives the following theorem.

Theorem 8.1.2. *There exists an algorithm for maintaining dynamic shortest distances $\leqslant k$ in unweighted graphs with updates in $\tilde{O}(k(n^{\omega(1,\epsilon,1)-\epsilon} + n^{1+\epsilon}))$ time and queries in $O(kn^{\epsilon})$ time. The algorithm is randomized and with small probability may return wrong, larger distances.*

Proof. To obtain the above bound we use the lazy computation schemes from Theorem 5.2.3. The update time and query time follow from the fact that the arithmetic operations over the formal power series modulo u^{k+1} can be carried out in time $\tilde{O}(k)$.

In the queries we have to compute the degree of the smallest degree term in an element from $R^k[[u]]$ and this takes $O(k)$ time. $\square$

Remark 2. Theorem 8.1.2 can be also used to maintain the part of the distance matrix, similarly as it was stated in Theorem 4.3.3. The update cost for α, β-restricted version of this problem is $\tilde{O}(k(n^{\omega(1,\epsilon,1)-\epsilon} + n^{1+\epsilon} + n^{\alpha+\beta}))$.

8.2 Path Decomposition Technique

The algorithms of Demetrescu and Italiano [11] for the dynamic distance problem are based on the path decomposition technique introduced in [21], and latter used in [10, 11, 22, 28, 53, 56]. Consider two vertices u and v and any path p between them. In order to compute the length of this path, we can perform searches from some selected nodes until the searches link up. That is, we explore the graph from each of the selected nodes at least as far as the next selected node is reached, see Fig. 8.2. Then we can sum up distances between the selected vertices. However, we do not know the path in advance but we can choose vertices for short searches in such a way that many long paths will go through them.

The way of choosing the nodes for the short searches is given by the following theorem.

Theorem 8.2.1 (Greene and Knuth '82)**.** *If we choose a set H nodes at random from n-node graph, then the probability that a simple given path has a sequence of more than $(cn \log n)/|H|$ nodes, none of which are from H, is bounded above by $2^{-\alpha c}$ for some positive α and sufficiently large n.*

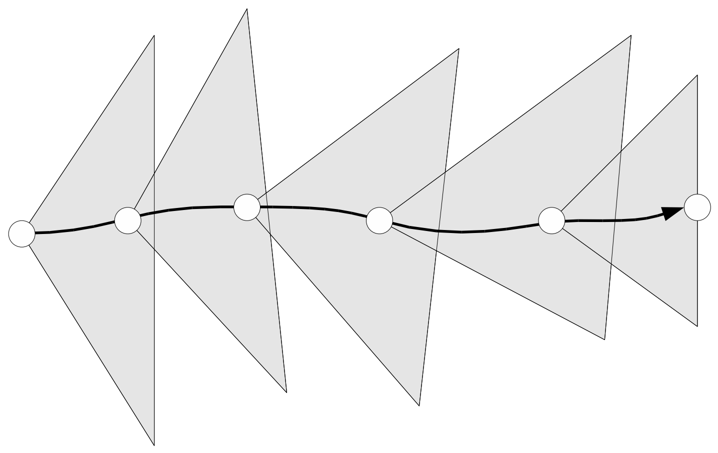


Figure 8.2: Path decomposition technique — we find a path by performing short searches.

This idea was used by Ullman and Yannakakis to develop a parallel algorithm for the shortest distance problem. The framework they developed needs three steps.

1. Select a set of vertices H uniformly at random.
2. Compute the distances $\leqslant k = (cn \log n)/|H|$ in G. Let D be the corresponding distance matrix.
3. Set $B := D_{H,H}$ and compute B^d.
4. The distances in G are with high probability given by

$$A^*_{i,j} = \min\left\{D_{i,j},\ \min_{p,q}\left\{D_{i,p} + B^*_{p,q} + D_{q,j}\right\}\right\}. \tag{8.2}$$

Equation 8.2 follows from Theorem 8.2.1. Consider the shortest path p from i to j. It is obviously a simple path and the theorem can be applied. We obtain that there is at least one vertex from H in sequence of k vertices in p with high probability. The path can be decomposed into shorter paths of lengths at most k starting and ending in H. The lengths of these subpaths are computed in Step 2. In Step 3 we stitch the subpaths together. The vertices i and j might not belong to H, so we need to add in (8.2) the distances from i to H and from H to j.

8.3 Sub-quadratic Dynamic Distances

We are now ready to prove the main theorem of this section.

Theorem 8.3.1. *There exists a randomized algorithm for the dynamic shortest distance problem in unweighted n-vertex graphs, supporting edge updates in $O(n^{1.932})$ time and queries in $O(n^{1.288})$ time.*

Proof. Let G be a graph under considerations. The algorithm is based on the idea presented in the previous section. In the algorithm we maintain:

- A set $H \subseteq V$ of vertices chosen uniformly at random, and such that $|H| = \frac{cn}{n^\mu}\log n = O(n^{1-\mu})$, for any constant $c > 0$, where $c\log n \leqslant n^\mu \leqslant n$.
- An $n \times n$ matrix D such that D_{ij} is the length of the shortest path from i to j in G, that uses at most n^μ edges.
- An $|H| \times |H|$ matrix B obtained from D by choosing only columns and rows corresponding to H.
- The distance matrix B^d of $G(B)$,i.e., the distance matrix obtained from B.

In each update:

- We update the matrix D using the algorithm from Theorem 8.1.2. This takes $\tilde{O}(n^\mu(n^{1+\epsilon} + n^{\omega(1,\epsilon,1)-\epsilon}))$ time.
- The cost of maintaining the matrix B is $\tilde{O}(n^\mu n^{2-2\mu)}$ as stated in Remark 2.
- We recompute the matrix B^* from scratch in $O(n^{3-3\mu})$ time.

The query on the distance from i to j can be answered by computing

$$\min\left\{D_{ij}, \min_{p,q\in H}\left\{D_{ip} + B^*_{pq} + D_{qj}\right\}\right\},$$

where we have to query out a row and a column from D, and then compute the minimum over p, q, what takes $O(n^{1-\mu+\epsilon} + n^{2-2\mu})$ operations.

We get the following bound on time for the edge update:

$$\tilde{O}(n^\mu(n^{1+\epsilon} + n^{\omega(1,\epsilon,1)-\epsilon}) + n^{2-2\mu}n^\mu + O(n^{3-3\mu})).$$

In order to get the fastest update time, we take $\epsilon = 0.575$. This balances the first two terms. By balancing the first and the last term we get

$$1 + \epsilon + \mu = 3 - 3\mu,$$

$$4\mu = 2 - \epsilon,$$

$$\mu = 0.357.$$

The first term is now equal to $\tilde{O}(n^{1.932})$, the third term is smaller and equal to $\tilde{O}(n^{1.644})$, whereas the query time is $O(n^{1.288})$. □

This result resolves the open question (see e.g. [10, 11, 12]) if there exist algorithms with sub-quadratic update and query times. The update cost of the above algorithm can be a little bit improved by applying the algorithms from [56]. However, we do not consider here these modification, because our result is only of a theoretical importance. The $\tilde{O}(n^2)$ algorithm from [12] is surely more efficient in practice.

Chapter 9

Perfect Matchings

9.1 Algebraic Matching Algorithms

Let $G = (V, E)$ be an undirected graph and $V = \{v_1, \ldots, v_n\}$. A *skew symmetric adjacency matrix* of G is an $n \times n$ matrix $\tilde{A}(G)$ such that

$$\tilde{A}(G)_{i,j} = \begin{cases} x_{i,j} & \text{if } \{v_i, v_j\} \in E \text{ and } i < j \\ -x_{i,j} & \text{if } \{v_i, v_j\} \in E \text{ and } i > j \\ 0 & \text{otherwise} \end{cases},$$

where $x_{i,j}$ are unique variables corresponding to the edges of G.

Tutte [52] observed the following

Theorem 9.1.1. *The symbolic determinant* $\det \tilde{A}(G)$ *is non-zero iff* G *has a perfect matching.*

This theorem gives an algorithm for testing if a graph has a perfect matching — we have only to test whether $\det \tilde{A}(G)$ is non-zero. Computing the determinant symbolically gives a deterministic but exponential time algorithm. In order to get a more efficient algorithm, we can apply the Zippel-Schwarz lemma (see Section 3.5). Determinant $\det \tilde{A}$ is a polynomial of degree n and if we compute it over Z_p for $p = O(n^2)$, we get the correct result with high probability.

Lovász showed that this method can be used to compute the size of a maximum matching.

Theorem 9.1.2. *The rank of* $\tilde{A}(G)$ *is equal to twice the maximum matching size.*

These matching algorithms can be implemented to run in time $O(n^\omega)$ using the fast matrix multiplication.

Let G be a graph and let $\tilde{A} = \tilde{A}(G)$ be its symbolic adjacency matrix. Rabin and Vazirani [37] showed that

Theorem 9.1.3. *With high probability,* $\mathrm{adj}(\tilde{A})_{j,i} \neq 0$ *iff the graph* $G - \{v_i, v_j\}$ *has a perfect matching.*

Let us denote by $A(G)$ the matrix obtained from $\tilde{A}(G)$ by substituting variables for random elements in Z_p. In particular $\mathrm{adj}(\tilde{A})_{j,i}$ is a polynomial of degree $n-1$, and hence, if $\{v_i, v_j\}$ is an edge in G, then with high probability $\mathrm{adj}(\tilde{A}_{j,i}) \neq 0$ iff $\{v_i, v_j\}$ is *allowed*, i.e. it is contained in some perfect matching.

Using Theorem 9.1.3 perfect matchings can be found in a straightforward manner: generate a random adjacency matrix $A(G)$, compute $A^{-1}(G)$, find an allowed edge, remove this edge together with its endpoints from G, generate A for the new graph, etc. (see Algorithm 1).

Algorithm 1 computes perfect matchings in general graphs in $O(n^{\omega+1})$ time.

```
FIND-PERFECT-MATCHING:

• $M := \emptyset$,
• **while** $G$ is non-empty **do**
    – compute $A^{-1}(G)$,
    – find an allowed edge $e \in E$,
    – remove $e$ and its endpoints from $G$,
    – add $e$ to $M$,
• return $M$.
```

The algorithm finds in each iteration one edge that belongs to the perfect matching. In each iteration we have to compute the inverse A^{-1} after some changes of A. This takes $O(n^{\omega})$ time. Thus, the whole algorithms runs in $O(n^{\omega+1})$ time. In the next section we show how the algorithms for the dynamic matrix inverse can be used to improve the running time of this method. The following theorem of Rabin and Vazirani shows that randomization is needed only in the first iteration. This is not vital for our algorithm but simplifies our analysis.

Theorem 9.1.4. *If* $A = A(G)$ *is non-singular, then for every vertex* v_i *there exists a vertex* v_j *such that* $A_{i,j} \neq 0$ *and* $(A^{-1})_{j,i} \neq 0$*, i.e.,* (v_i, v_j) *is an allowed edge. Moreover, the matrix* A *with rows* i*,* j *and columns* i*,* j *removed is also non-singular.*

The above theorem states that the same substitution can be used for finding a whole perfect matching.

All of the above results apply also to bipartite adjacency matrices. Let $G = (U, V, E)$ be a bipartite graph, where $|U| = |V| = n$, $U = \{u_1, \ldots, u_n\}$, $V = \{v_1, \ldots, v_n\}$. Let $\tilde{B}(G)_{i,j} = x_{i,j}$ if $(u_i, v_j) \in E$, otherwise let $\tilde{B}(G)_{i,j} = 0$. Any result for bipartite adjacency matrix can be derived from the general case using the following two equalities.

$$\tilde{A}(G) = \begin{bmatrix} 0 & \tilde{B}(G) \\ -\tilde{B}(G) & 0 \end{bmatrix}, \qquad \tilde{A}(G)^{-1} = \begin{bmatrix} 0 & -\tilde{B}(G)^{-1} \\ \tilde{B}(G)^{-1} & 0 \end{bmatrix}.$$

Hence the symbolic determinant of $\tilde{B}(G)$ is non-zero iff the graph has a perfect matching and the rank of the matrix is equal to the maximum matching size.

9.2 Perfect Matching via Dynamic Matrix Inverse

The algorithm from Theorem 4.1.1 can be directly applied to obtain a very simple $O(n^3)$ algorithm for the perfect matching problem (see Algorithm 2). This algorithm is a straightforward implementation of Algorithm 1 with only one difference. In Algorithm 1, when we choose an edge $\{u, v\}$ to the matching, the vertices u and v are removed from the graph. Now we do not remove these vertices, instead we remove all the other edges adjacent to u and v.

Algorithm 2 An $O(n^3)$ time algorithm for finding perfect matchings.

```
FIND-PERFECT-MATCHING-2:

• initialize the algorithm for dynamic inverse with the matrix $A(G)$,
• $M := \emptyset$,
• for $i = 1$ to $\frac{n}{2}$ do
    – let $i$ be an unmatched vertex,
    – for $j := 1$ to $n$ do
        ∗ $x_j :=$ query(i,j),
    – let $j$ be a non-zero element in $x$,
    – add $(i, j)$ to $M$,
    – update-column($i, e_j$),
    – update-row($j, e_i$),
    – update-column($j, e_i$),
    – update-row($i, e_j$),
• return $M$.
```

This algorithms is an improvement over the results of Rabin and Vazirani. The question arises whether we can construct in this way an $o(n^3)$ time algorithm. In Algorithm 2 we have to modify whole rows and columns, so at the first sight there is no way of obtaining faster algorithms with the use of the algorithm from Theorem 4.3.2. This algorithm is fast enough but it supports only simple updates. However, there is a solution. We add a new operation **clear(j,i,x)**. This operations clears all elements in the i-th column and then sets $A_{i,j}$ to x. It modifies the whole column, but it can be realized in the same time as simple updates.

Theorem 9.2.1. *The algorithm in Theorem 4.3.2 can be extended to support the* **clear** *operation in $O(n^{\omega(1,\varepsilon,1)-\varepsilon} + n^{1+\varepsilon})$ arithmetic operations (worst-case).*

Proof. The clear operation can be translated to the update of the i-th column to e_j. The only problem is to compute the vector b for this update. The rest of this update can be done as in the UPDATE-LAZY-INVERSE procedure. We have

$$b = (N + I)M(e_j - (A)_i),$$

but $(N + I)M = A^{-1}$, and

$$b = A^{-1}(e_j - (A)_i) = A^{-1}e_j - A^{-1}(A)_i =$$

$$= A^{-1}e_j - e_i = (N + I)Me_j - e_i.$$

The last formula allows to compute b in $O(n^{1+\varepsilon})$ arithmetic operations. □

Using the **clear** operation we can obtain an $O(n^{2.575})$ time algorithm for finding perfect matchings – Algorithm 3.

Algorithm 3 An $O(n^{2.575})$ time algorithm for finding perfect matchings.

```
FIND-PERFECT-MATCHING-3:

• initialize the algorithm for dynamic inverse with the matrix $A(G)$,
• $M := \emptyset$,
• for $i = 1$ to $\frac{n}{2}$ do
    – let $i$ be any unmatched vertex,
    – for $j := 1$ to $n$ do
        ∗ $x_j :=$ query(i,j),
    – let $j$ be a non-zero element in $x$,
    – add $(i, j)$ to $M$,
    – clear$(i, j, 1)$,
    – clear$(j, i, 1)$,
• return $M$.
```

The important difference between this algorithm and Algorithm 2 is that we clear elements only in columns of the matrix. There is no need to clear rows, since

$$\det(A) = \sum_{\rho \in \Gamma} \operatorname{sgn}(\rho) \prod_{k=1}^{n} a_{k,\rho_k}, \tag{9.1}$$

we have

$$A = \begin{bmatrix} a_{1,1} & \dots & a_{1,i-1} & 0 & a_{1,i+1} & \dots & a_{1,n} \\ \vdots & & \vdots & \vdots & \vdots & & \vdots \\ a_{j-1,1} & \dots & a_{j-1,i-1} & 0 & a_{j-1,i+1} & \dots & a_{j-1,n} \\ a_{j,1} & \dots & a_{j,i-1} & 1 & a_{j,i+1} & \dots & a_{j,n} \\ a_{j+1,1} & \dots & a_{j+1,i-1} & 0 & a_{j+1,i+1} & \dots & a_{j+1,n} \\ \vdots & & \vdots & \vdots & \vdots & & \vdots \\ a_{,1n} & \dots & a_{n,i-1} & 0 & a_{n,i+1} & \dots & a_{n,n} \end{bmatrix}$$

Notice that for each permutation ρ in (9.1) that gives a non-zero contribution, the element $\rho_{j,i}$ must be present. Thus, the other elements in the j-th row of A are not important. Hence we have the following theorem.

Theorem 9.2.2. *If there exists an algorithm for the dynamic matrix inverse in the non-singular case with clear cost of $O(n^\alpha)$ arithmetic operations and query cost of $O(n^\beta)$ arithmetic operations, then there exists an algorithm for computing perfect matching in $O(n \cdot n^\alpha + n^2 \cdot n^\beta)$ time. The algorithm is randomized with one-sided error.*

Combining this theorem with Theorem 4.3.2 we obtain an $O(n^{2.575})$ time algorithm for the perfect matching problem.

9.3 Perfect Matching via Gaussian Elimination

In the previous section we showed the algorithm for the perfect matching problem working in $O(n^{2.575})$ time. In this section we show an algorithm for finding perfect matchings in bipartite graphs in $O(n^\omega)$ time. The algorithms was developed together with Marcin Mucha [35]. In order to obtain an $O(n^\omega)$ time algorithm for general graphs, one have to use some structural properties described in [32]. For details see [35]. The approach presented in this section can be also used to find perfect matchings in planar graphs in $O(n^{\frac{\omega}{2}})$ time [34].

The basic idea is to implement the operation **clear** as shown in the following theorem.

Theorem 9.3.1 (Elimination Theorem). *Let*

$$A = \begin{pmatrix} a_{1,1} & v^T \\ u & B \end{pmatrix} \quad A^{-1} = \begin{pmatrix} \hat{a}_{1,1} & \hat{v}^T \\ \hat{u} & \hat{B} \end{pmatrix} \quad A' = \begin{pmatrix} a_{1,1} & 0 \\ 0 & B \end{pmatrix},$$

where $\hat{a}_{1,1} \neq 0$. Then

$$A'^{-1} = \begin{pmatrix} a_{1,1}^{-1} & 0 \\ 0 & \hat{B} - \hat{u}\hat{v}^T/\hat{a}_{1,1} \end{pmatrix}.$$

Proof. Since $AA^{-1} = I$, we have

$$\begin{pmatrix} a_{1,1}\hat{a}_{1,1} + v^T\hat{u} & a_{1,1}\hat{v}^T + v^T\hat{B} \\ u\hat{a}_{1,1} + B\hat{u} & u\hat{v}^T + B\hat{B} \end{pmatrix} = \begin{pmatrix} I_1 & 0 \\ 0 & I_{n-1} \end{pmatrix}.$$

Using these equalities we get

$$B(\hat{B} - \hat{u}\hat{v}^T/\hat{a}_{1,1}) = I_{n-1} - u\hat{v}^T - B\hat{u}\hat{v}^T/\hat{a}_{1,1} =$$
$$I_{n-1} - u\hat{v}^T + u\hat{a}_{1,1}\hat{v}^T/\hat{a}_{1,1} = I_{n-1} - u\hat{v}^T + u\hat{v}^T = I_{n-1}.$$

And further

$$A'\begin{pmatrix} a_{1,1}^{-1} & 0 \\ 0 & \hat{B} - \hat{u}\hat{v}^T/\hat{a}_{1,1} \end{pmatrix} =$$

$$= \begin{pmatrix} a_{1,1}^{-1} & 0 \\ 0 & \hat{B} - \hat{u}\hat{v}^T/\hat{a}_{1,1} \end{pmatrix}\begin{pmatrix} a_{1,1}^{-1} & 0 \\ 0 & \hat{B} - \hat{u}\hat{v}^T/\hat{a}_{1,1} \end{pmatrix} =$$

$$= \begin{pmatrix} a_{1,1}\hat{a}_{1,1}^{-1} & 0 \\ 0 & B(\hat{B} - \hat{u}\hat{v}^T/\hat{a}_{1,1}) \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & I_{n-1} \end{pmatrix} = I_n.$$

□

The modification of $\hat{B}$ described in this theorem is in fact a single step of the well known Gaussian elimination procedure. In this case we eliminate the first variable (column) using the first equation (row). Similarly, we can eliminate from A^{-1} any other variable (column) j using any equation (row) i such that $A^{-1}_{i,j} \neq 0$.

As an immediate consequence of Theorem 9.3.1 we get simple $O(n^3)$ algorithms for finding perfect matchings in bipartite (Algorithm 4) and general (Algorithm 5) graphs.

Algorithm 4 Simple algorithm for perfect matchings in bipartite graphs.

```
SIMPLE-BIPARTITE-MATCHING($G$):
$X = B^{-1}(G)$,
$M = \emptyset$,
for $c = 1$ to $n$ do
    1. find a row $r$, not yet eliminated, and such that $X_{r,c} \neq 0$ and $B(G)_{c,r} \neq 0$ (($u_c, v_r$)
       is an allowed edge in $G - V(M)$),
    2. eliminate the $r$-th row and the $c$-th column of $B$,
    3. add $(u_c, v_r)$ to $M$.
```

Algorithm 5 Simple algorithm for perfect matchings in general graphs.

```
SIMPLE-GENERAL-MATCHING($G$):
$X = A^{-1}(G)$,
$M = \emptyset$,
for $c = 1$ to $n$ do
  if column $c$ is not yet eliminated then
    1. find a row $r$, not yet eliminated, and such that $X_{r,c} \neq 0$ and $A(G)_{c,r} \neq 0$ (($v_c, v_r$)
       is an allowed edge in $G - V(M)$),
    2. eliminate the $r$-th row and the $c$-th column of $B$,
    3. eliminate the $c$-th row and the $r$-th column of $B$,
    4. add $(v_c, v_r)$ to $M$.
```

Let $G = (U, V, E)$ be a bipartite graph having a perfect matching, and let $B = B(G)$ be its bipartite adjacency matrix. Assume that the column order is fixed and the columns are eliminated from the left to the right. To eliminate the i-th column we have to find a row j, not already eliminated, such that $(u_i, v_j) \in E$ and the updated value of $A^{-1}(j, i)$ is non-zero. Then we eliminate the i-th row and the j-th column. However, there is an algorithm that does exactly what we need — the classical LU factorization algorithm of Hopcroft and Bunch (see [5] and [1]). This algorithm looks for a non-zero element in the currently eliminated column. It is possible to modify it in such a way that it only chooses elements corresponding to edges of G. Due to Theorem 9.1.4, we know that such an element exists in each column. The algorithm of Hopcroft and Bunch works in $O(n^\omega)$ time, and thus it is possible to find perfect matchings in bipartite graphs also in $O(n^\omega)$ time. To obtain an $O(n^\omega)$ time algorithm for general graphs we need some additional consideration. For details see [35].

Bibliography

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman Publishing Co., Inc., 1974.

[2] G. Ausiello, G.F. Italiano, A. Marchetti Spaccamela, and U. Nanni. Incremental algorithms for minimal length paths. *J. Algorithms*, 12(4):615–638, 1991.

[3] S. Baswana, R. Hariharan, and S. Sen. Improved decremental algorithms for maintaining transitive closure and all-pairs shortest paths. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of Computing*, pages 117–123. ACM Press, 2002.

[4] N. Blum. A New Approach to Maximum Matching in General Graphs. In *Proc. 17th ICALP*, volume 443 of *Lecture Notes in Computer Science*, pages 586–597. Springer-Verlag, 1990.

[5] J. Bunch and J. Hopcroft. Triangular factorization and inversion by fast matrix multiplication. *Math. Comp.*, 28:231–236, 1974.

[6] C.J. Colbourn. *The Combinatorics of Network Reliability*. Oxford University Press, Inc., 1987.

[7] D. Coppersmith. Rectangular Matrix Multiplication Revisited. *J. Complex.*, 13(1):42–49, 1997.

[8] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *Proceedings of the nineteenth annual ACM symposium on Theory of Computing*, pages 1–6. ACM Press, 1987.

[9] C. Demetrescu and G.F. Italiano. Fully Dynamic Transitive Closure: Breaking Through the $O(n^2)$ Barrier. In *Proceedings of 41th annual IEEE Symposium on Foundations of Computer Science*, pages 381–389, 2000.

[10] C. Demetrescu and G.F. Italiano. Fully Dynamic All Pairs Shortest Paths with Real Edge Weights. In *Proceedings of 42th annual IEEE Symposium on Foundations of Computer Science*, pages 260–267, 2001.

[11] C. Demetrescu and G.F. Italiano. Improved Bounds and New Trade-Offs for Dynamic All Pairs Shortest Paths. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming*, pages 633–643. Springer-Verlag, 2002.

[12] C. Demetrescu and G.F. Italiano. A new approach to dynamic all pairs shortest paths. In *Proceedings of the thirty-fifth annual ACM Symposium on Theory of Computing*, pages 159–166. ACM Press, 2003.

[13] S. Even. *Graph Algorithms.* Computer Science Press, Rockville, 1979.

[14] S. Even and H. Gazit. Updating Distances in Dynamic Graphs. *Methods of Operations Research*, 49:371–387, 1985.

[15] J. Fakcharoenphol. Planar Graphs, Negative Weight Edges, Shortest Paths, and Near Linear Time. In *Proceedings of the 42nd IEEE symposium on Foundations of Computer Science*, pages 232–241. IEEE Computer Society, 2001.

[16] G.S. Frandsen, J.P. Hansen, and P.B. Miltersen. Lower Bounds for Dynamic Algebraic Problems. *Lecture Notes in Computer Science*, 1563:362–372, 1999.

[17] M.L. Fredman. The Complexity of Maintaining an Array and Computing Its Partial Sums. *J. ACM*, 29(1):250–260, 1982.

[18] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Semi-dynamic Algorithms for Maintaining Single Source Shortest Paths Trees. *Algorithmica*, 22(3):250–274, 1998.

[19] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Fully Dynamic Algorithms for Maintaining Shortest Paths Trees. *J. Algorithms*, 34(2):251–281, 2000.

[20] H.N. Gabow and R.E. Tarjan. Faster scaling algorithms for general graph matching problems. *J. ACM*, 38(4):815–853, 1991.

[21] D.H. Greene and D.E. Knuth. *Mathematics for the Analysis of Algorithms.* Birkhäuser, 1982.

[22] M.R. Henzinger and V. King. Fully Dynamic Biconnectivity and Transitive Closure. In *Proceedings 36th annual IEEE Symposiumon Foundations of Computer Science*, pages 664–672, 1995.

[23] M.R. Henzinger, P.Klein, S.Rao, and S.Subramanian. Faster Shortest-path Algorithms for Planar Graphs. *J. Comput. Syst. Sci.*, 55(1):3–23, 1997.

[24] X. Huang and V.Y. Pan. Fast Rectangular Matrix Multiplication and Applications. *Journal of complexity*, 14(2):257–299, 1998.

[25] T. Ibaraki and N. Katoh. On-line Computation of Transitive Closure for Graphs. *Inform. Proc. Lett.*, 16:95–97, 1983.

[26] G.F. Italiano. Amortized Efficiency of a Path Retrieval Data Structure. *Theor. Comput. Sci.*, 48(2-3):273–281, 1986.

[27] S. Khanna, R. Motwani, and R.H. Wilson. On Certificates and Lookahead on Dynamic Graph Problems. In *Proceedings 7th annual ACM-SIAM Symposiumon on Discrete Algorithms*, pages 222–231, 1996.

[28] V. King. Fully Dynamic Algorithms for Maintaining All-Pairs Shortest Paths and Transitive Closure in Digraphs. In *Proceedings of 40th annual IEEE Symposium on Foundations of Computer Science*, pages 81–91, 1999.

[29] V. King and G. Sagert. A Fully Dynamic Algorithm for Maintaining the Transitive Closure. In *Proceedings of the thirty-first annual ACM Symposium on Theory of Computing*, pages 492–498. ACM Press, 1999.

[30] J.A. La Poutré and J. van Leeuwen. Maintenance of Transitive Closure and Transitive Reduction of Graphs. In *Proc. Workshop on Graph-Theoretic Concepts in Computer Science*, pages 106–120. Lecture Notes in Computer Science 314, Springer-Verlag, Berlin, 1988.

[31] P. Loubal. A Network Evaluation Procedure. *Highway Research Record*, 205:96–109, 1967.

[32] L. Lovász and M.D. Plummer. *Matching Theory.* North-Holland, N.Y., 1986.

[33] S. Micali and V.V. Vazirani. An $O(\sqrt{|V|}|E|)$ algorithm for finding maximum matching in general graphs. In *Proceedings of the twenty first annual IEEE Symposium on Foundations of Computer Science*, pages 17–27, 1980.

[34] M. Mucha and P. Sankowski. Maximum Matchings in Planar Graphs via Gaussian Elimination. In *Proceedings of 12th annual European Symposium on Algorithms*, pages 532–543, 2004.

[35] M. Mucha and P. Sankowski. Maximum Matchings via Gaussian Elimination. In *Proceedings of the 45th annual IEEE Symposium on Foundations of Computer Science*, pages 248–255, 2004.

[36] J. Murchland. The Effect of Increasing or Decreasing the Length of a Single Arc on All Shortest Distances in a Graph. *Technical report*, LBS-TNT-26, 1967.

[37] M.O. Rabin and V.V. Vazirani. Maximum Matchings in General Graphs Through Randomization. *Journal of Algorithms*, 10:557–567, 1989.

[38] G. Ramalingam and T. Reps. An Incremental Algorithm for a Generalization of the Shortest-path Problem. *J. Algorithms*, 21(2):267–305, 1996.

[39] G. Ramalingam and T. Reps. On the Computational Complexity of Dynamic Graph Problems. *Theor. Comput. Sci.*, 158(1-2):233–277, 1996.

[40] J.H. Reif and S.R. Tate. On Dynamic Algorithms for Algebraic Problems. *J. Algorithms*, 22(2):347–371, 1997.

[41] V. Rodionov. The Parametric Problem of Shortest Distances. *U.S.S.R. Computational Math. and Math. Phys.*, 8(5):336–343, 1968.

[42] L. Roditty. A Faster and Simpler Fully Dynamic Transitive Closure. In *Proceedings of the fourteenth annual ACM-SIAM Symposium on Discrete Algorithms*, pages 404–412. Society for Industrial and Applied Mathematics, 2003.

[43] L. Roditty and U. Zwick. Improved Dynamic Reachability Algorithms for Directed Graphs. In *Proceedings of the 43rd Symposium on Foundations of Computer Science*, page 679. IEEE Computer Society, 2002.

[44] L. Roditty and U. Zwick. A Fully Dynamic Reachability Algorithm for Directed Graphs with an Almost Linear Update Time. In *Proceeding of the 36th annual ACM Symposium on Theory of Computing*, pages 184–191. ACM Press, 2004.

[45] H. Rohnert. A Dynamization of the All Pairs Least Cost Path Problem. In *Proceedings of the 2nd Symposium of Theoretical Aspects of Computer Science*, pages 279–286. Springer-Verlag, 1985.

[46] P. Sankowski. Dynamic Transitive Closure via Dynamic Matrix Inverse. In *Proceedings of the 45th annual IEEE Symposium on Foundations of Computer Science*, pages 509–517, 2004.

[47] J.E. Savage. An Algorithm for the Computation of Linear Forms. *SIAM J. Comput.*, 3(2):150–158, 1974.

[48] A. Schonhage and V. Strassen. Schnelle Multiplikation grosser Zahlen. *Computing*, 7:281–292, 1971.

[49] J.T. Schwartz. Fast Probabilistic Algorithms for Verification of Polynomial Identities. *J. Algorithms*, 10:701–717, 1980.

[50] V. Strassen. Gaussian Elimination is Not Optimal. *Numerische Mathematik*, 13:354–356, 1969.

[51] V. Strassen. Vermeidung von Divisionen. *J. reine u. angew. Math.*, 264:182–202, 1973.

[52] W.T. Tutte. The Factorization of Linear Graphs. *J. London Math. Soc.*, 22:107–111, 1947.

[53] J. Ullman and M. Yannakakis. High-probability Parallel Transitive Closure Algorithms. In *Proceedings of the 1990 ACM Symposium on Parallel Algorithms and Architectures*, pages 200–209, July 1990.

[54] D.M. Yellin. Speeding up Dynamic Transitive Closure for Bounded Degree Graphs. *Acta Informatica*, 30:369–384, 1993.

[55] R.E. Zippel. Probabilistic Algorithms for Sparse Polynomials. *Proc. EUROSAM 79, Lecture Notes in Computer Science 72*, pages 216–226, 1979.

[56] U. Zwick. All Pairs Shortest Paths in Weighted Directed Graphs Exact and Almost Exact Algorithms. In *Proceedings of the 39th annual IEEE Symposium on Foundations of Computer Science*, pages 310–319, 1998.