Faculty of Mathematics and Computer Science, Babes-Bolyai University,
Cluj-Napoca

The study of the dynamic graphs

PhD Thesis Summary

PhD student: Pătcaş Csaba
Supervisor: Prof. Dr. Kása Zoltán

2011

Abstract

Dynamic graphs are graphs that can change in time, by undergoing a number of local updates. In a dynamic graph problem some property of the graph must be maintained during these updates and queries must be answered as efficiently as possible, without recomputing everything from scratch using a classical static graph algorithm. Dynamic graph problems have a wide range of applications and can also be used to speed-up existing static algorithms.

Our dissertation is a comprehensive study of data structures, techniques and algorithms used in solving dynamic graph problems. We emphasize the practical aspect of these solutions, by giving an overview of the experimental studies conducted for each of the most important dynamic graph problems. In every case we summarize the current state of the art and give recommendations for algorithms to be used in practical applications, based on the expected structure of the graph and operation sequence. The explanation of the algorithms is facilitated by pseudocodes and figures.

Our most original contribution is the study of the debts' clearing problem. It is a problem with applications mainly in economics, which can be naturally modeled using graph theory. We prove that the problem is NP-hard, but provide an exact algorithm, that can be useful for reasonable sizes of the input. Reformulating the problem in dynamic graphs can have its own set of applications. We give a data structure to solve the dynamic version of the problem, which can also be used to develop a different solution for the static version. We report the results of an extensive experimental study comparing these algorithms and in the closing of our dissertation we propose a genetic algorithm to solve the problem for large inputs.

Keywords: dynamic graph, debt clearing

Acknowledgements

I wish to thank to my supervisor Kása Zoltán, for all the scientific assistance given during these years. I also have to thank him for his patience, as I am well aware that it was needed when working with me. He was kind and smiling with me, even when I did not expect him to be so. It seems that he somehow knew that being strict is not the best way to motivate me, at least not in the long run.

I am grateful to my former teacher and actual colleague Robu Judit for the motivation given along the way and for arranging my mobility to Linz, providing me the best conditions to work on my dissertation. Without her help, I could not have finished in time.

I wish to thank to my family for supporting me financially, emotionally and otherwise, when the times were tough – and they were on some occasions during the last five years. When I was in elementary school, my mother used to help me on many occasions with the homework; sometimes, when I was still not satisfied with the result even after working hours and hours, she used to tell me: “Let it go son, you are not writing your doctoral dissertation!”[1]. Well, now I am.

Last but not least I am thankful to my friends, colleagues and all the persons around me. Some were part of my life at one point and had a more significant influence than others. My closest friends organized a great surprise farewell party for me before I left to Linz and supported me in all kind of matters before and during my stay. Ionescu Klára, former teacher and now a friend, shared a great amount of her experience with me when we prepared together problems for programming contests and in our private conversations. Csató Lehel made some useful comments on the preliminary version of this dissertation. I am thankful to Bartha Attila for implementing the genetic algorithm solving the debts’ clearing problem.

All of these persons contributed to my growth on the journey to be the person I am today.

[1] The exact Hungarian phrase was: „Ó, jó lesz az már fiam, nem a doktori disszertációdat írod!”

Contents

Table of Notations

Bold font	New term
CAPITAL FONT	Abstract method name, problem name
`Typewriter font`	Abbreviation of an algorithm, concrete implementation of a method
Italic font	Variable names, mathematical notations
n	Number of nodes in a graph
m	Number of edges or arcs in a graph
nr_q	Number of queries in a dynamic graph algorithm
nr_u	Number of updates in a dynamic graph algorithm
nr	Total number of operations in a dynamic graph algorithm
t_u	Worst-case update time complexity
t_q	Worst-case query time complexity
$\overline{t_u}$	Amortized update time complexity
$\overline{t_q}$	Amortized query time complexity
t_a	Amortized time complexity per operation
t_p	Preprocessing time complexity per operation
t_t	Total expected worst-case running time complexity
$u \cdots v$	Path from node u to node v

List of Acronyms

`Abd97`	Abdeddaïm's algorithm, described in [Abd97]
`Abd00`	Abdeddaïm's algorithm, described in [Abd00]
`DSF`	Disjoint set forest, described in Section 2.1
`Debt`	Our algorithm for fully dynamic debt clearing, described in Section 4.2
`ES`	Decremental connectivity algorithm by Even and Shiloach
`FredI-85`	Topological partition by Frederickson
`FredI-91`	Restricted partition by Frederickson
`FredI-Mod`	Light partition by Amato et. al
`FredII-85`	Topology tree based on topological partitions by Frederickson
`FredII-91`	Topology tree based on restricted partitions by Frederickson
`FredIII-85`	2-dimensional topology tree based on `FredII-85`
`FredIII-91`	2-dimensional topology tree based on `FredII-91`
`HK`	Fully dynamic connectivity algorithm by Henzinger and King
`HT`	Refinement of `HK` by Henzinger and Thorup
`HDT`	Fully dynamic connectivity algorithm by Holm, De Lichtenberg and Thorup
`HDTMST`	Fully dynamic minimum spanning tree algorithm by Holm, De Lichtenberg and Thorup
`Ital`	Partially dynamic transitive closure algorithms by Italiano
`Ital-Gen`	Generalization of `Ital` by Frigioni et. al
`KUF`	k-UF tree by Blum, described in Section 2.2
`Spars(X)`	Sparsification described in Section 2.6 on top of algorithm `X`
`ThoDec`	Decremental connectivity algorithm by Thorup

1 Introduction

1.1 Motivation

Graph theory is an established area of research in combinatorial mathematics. It is also one of the most active areas of mathematics that has found a large number of applications in diverse areas including not only computer science, but also chemistry, physics, biology, anthropology, psychology, geography, history, economics, and many branches of engineering. Graph theory has been especially useful in computer science, since any data structure can be represented by a graph. Furthermore, there are applications in networking, in the design of computer architectures, and generally, in virtually any branch of computer science ([HarGup97]).

Traditional graph algorithms operate on static graphs. They deal with the development of an algorithm, that, given a fixed graph as input, solves a particular problem on it, for example: "is the graph connected?".

Dynamic graphs are not fixed in time, but can evolve through local changes of the graph. The problem has to be resolved quickly after each modification. The challenge for an algorithm dealing with a dynamic graph is to maintain in an environment of dynamic local changes, the desired graph property efficiently, that is, without recomputing everything from scratch after each dynamic change. Dynamic graphs model many graphs occurring in real-life applications much more closely, because no large system is truly static ([AlbEtAl98, Zar02]).

1.2 The structure of this work

In Section 1.1 we give the main motivational factors of this dissertation. A list of our publications and original results can be found in Section 1.3. In Section 1.4 we lay down the theoretical base of the main concepts used throughout this work along with our conventions for notations. In the theoretical analysis of dynamic graph problems various computational models are used, which are shortly described in Section 1.5.

In Chapter 2 a collection of advanced data structures used in dynamic graph algorithms can be found. In Chapters 3 and 4 the main dynamic problems in undirected, respectively directed graphs are considered.

1.3 Own contribution

Our original contributions are listed below. Most of them were published in [Pat09], [Pat11], [Pat11b] and [PatBar11]. Some related work can be also found in [PatIon08] and [IonPat08].

- A comprehensive presentation of problems on directed and undirected dynamic graphs, with comparisons of novel algorithms not only from a theoretical but also a practical point of view.
- To the best of our knowledge the first detailed pseudocode for the algorithm by Even and Shiloach and implementation details for other algorithms.
- Mathematically rigorous proofs for NP-hardness, NP-hardness in the strong sense and NP-easiness of the debts' clearing problem in the general case and in the case restricted to a single path.
- Algorithms and data structures developed to solve the debts' clearing problem in the static and also the dynamic version, tested in a set of experiments.
- New recombination and mutation operators used in the genetic algorithm.

1.4 Definitions and notations

In this section we state some well-known graph theoretical definitions, followed by the definition of the dynamic graph and a short categorization of dynamic graph problems.

Definition 1.1 We call $G = (V, E)$ a **graph**, where V is the set of **nodes** or **vertices**, and $E \subseteq V \times V$ is the set of **edges** or **arcs**. $\square$

Definition 1.2 If $(i, j) \in E \Leftrightarrow (j, i) \in E$ the graph is **undirected**, and E is the set of edges, otherwise the graph is **directed** (sometimes called a **digraph**) and we call E the set of arcs (sometimes noted by A). $\square$

Definition 1.3 If G is directed and contains no cycles, then it is called a **directed acyclic graph**, commonly abbreviated as **DAG**. $\square$

Definition 1.4 In a **weighted graph** we have a weight associated to each edge or arc, $w : E \to \mathbb{R}$. $\square$

Definition 1.5 A **dynamic graph** is a graph that changes in time by undergoing a sequence of updates. An update is an operation that inserts or deletes edges or nodes of the graph, or changes attributes associated to edges or nodes. $\square$

Definition 1.6 A dynamic graph problem is said to be **incremental** if only insertions are allowed. $\square$

Definition 1.7 A dynamic graph problem is said to be **decremental** if only deletions are allowed. $\square$

Definition 1.8 A dynamic graph problem is said to be **partially dynamic** if it is either incremental or decremental. $\square$

Definition 1.9 A dynamic graph problem is said to be **fully dynamic** if there is no restriction regarding the type of updates. $\square$

1.5 Computational models

Several computational models have been developed to facilitate the theoretical analysis of dynamic graph algorithms. A short description of the most important ones is given in the corresponding sections of our dissertation.

2 Data structures

2.1 Disjoint set forest

Introduction This data structure is used to represent disjoint sets of items and is the main building block in incremental connectivity algorithms.

Supported operations Each set has a **name**, which in most of the cases is just a member of the set (also called a representative). The supported operations are:

- $\textsc{Make}(x)$: Creates a new set, whose only member is x. Since the sets are disjoint, x must not be in any other set.
- $\textsc{Union}(x, y)$: Unites the sets having x and y as representatives. $x \neq y$ is assumed.
- $\textsc{Find}(x)$: Returns the name of the set, that contains x.

Performance Let nr_u be the number of UNION operations, nr_q the number of FIND operations, n the number of MAKE operations and $nr = nr_u + nr_q + n$. As the sets ar disjoint, it is easy to deduce, that after $n-1$ UNION operations only one set would remain, thus $nr_u \leq n-1 \Rightarrow nr_u = O(n)$. We also assume, that the MAKE operations are the first n operations performed.

MAKE and UNION are supported in constant worst-case time, while FIND is $t_q = O(\log n)$ in the worst case. A whole sequence of operations has $t_t = O(n + nr_q \cdot \alpha(nr_q + n, n))$ total expected time and $\Theta(n)$ memory is needed to store the forest.

α is a very slowly growing function, which does not exceed four in any practical application. It is defined as a functional inverse of Ackermann's function, defined as follows:

$A(1, j) = 2^j$, if $j \geq 1$

$A(i, 1) = A(i-1, 2)$, if $i \geq 2$

$A(i, j) = A(i-1, A(i, j-1))$, if $i, j \geq 2$

$\alpha(m,n) = \min\{i \geq 1 | A(i, \lfloor m/n \rfloor) > \log n\}$

2.2 k-UF tree

Introduction k-UF trees were introduced in [Blu85] to solve the disjoint set union problem, also giving the best worst-case complexity per operation for incremental connectivity.

Supported operations k-UF trees support the same operations as disjoint set forests: Make, Union and Find.

Performance Both Find and Union take $O(\log n / \log\log n)$ time in the worst case, and their running time does not amortize. The memory complexity is $\Theta(n)$.

2.3 Vertex cluster

Introduction Vertex clusters, topology trees and 2-dimensional topology trees were first introduced in [Fre83] to support minimum spanning trees under the operation of updating the cost of an edge in the graph, but they have several other applications, such as efficiently maintaining the minimum spanning tree in planar graphs, connectivity, generating the k smallest spanning trees or 2-edge connectivity under edge and node insertion and deletion ([Fre85, Fre97])).

Vertex clusters work on top of a spanning tree of the graph and are based on grouping the nodes of the graph into sets that induce a connected subgraph. All of the four different strategies of clustering we know of are described, each having its own advantages and disadvantages.

Definition 2.1 To avoid confusion, in the rest of this work we refer to the original tree as **underlying tree**, to differentiate it from the tree built upon it. The underlying tree is sometimes a spanning tree of the original input graph, which is called **underlying graph**. □

Definition 2.2 The edges from the underlying graph, that are not in the underlying spanning forest, are called **nontree edges**. □

Supported operations The following operations are supported:

- SWITCH(u, v, x, y): replaces tree edge (x, y) with (u, v). It is assumed, that (x, y) is on the path connecting u and v in the tree.
- REMOVE(u, v): deletes edge (u, v) from the spanning tree and returns a replacement edge, if it exists. It is assumed, that (u, v) is in the spanning tree.

Performance Both SWITCH and REMOVE can be supported in $O(m^{2/3})$, using $O(m)$ space and preprocessing time $t_p = O(m)$.

2.4 Topology tree

Introduction A topology tree is a hierarchical representation of clusters, built by recursively applying a partition of the nodes, until one node is left.

Supported operations Topology trees support the same operations as vertex clusters. Additionally to implement these operations one may need to split a topology tree, or merge two topology trees.

- SPLIT(T, u, v): splits the topology tree T after the deletion of tree edge (u, v).
- MERGE(T_1, T_2): merges the topology trees T_1 and T_2..
- SWITCH(u, v, x, y): replaces tree edge (x, y) with (u, v). It is assumed, that (x, y) is on the path connecting u and v in the tree.
- REMOVE(u, v): deletes edge (u, v) from the spanning tree and returns a replacement edge, if it exists. It is assumed, that (u, v) is in the spanning tree.

Performance The topology tree can be built in time linear on the number of nodes of the spanning tree. SPLIT and MERGE can be carried out in $O(\log n)$, where n is the number of vertices of the tree. SWITCH and REMOVE are supported in $O(\sqrt{m \log m})$ time, with the preprocessing time being $t_p = O(m)$ and the space complexity also $O(m)$.

2.5 2-dimensional topology tree

Supported operations 2-dimensional topology trees support the same operations as topology trees.

Performance All operations take $O(\sqrt{m})$ time, with preprocessing time $t_p = O(m)$ and $O(m)$ space requirement.

2.6 Sparsification tree

Introduction Sparsification is a general technique, which applies to a wide variety of dynamic graph problems. It can be applied on top of graph algorithms to speed them up and can be used as a black box, that is it does not require knowledge of the internal details of the underlying algorithm. It was introduced in [EppEtAl92] improving time bounds for several dynamic graph problems, such as minimum spanning trees, k smallest spanning trees, connectivity, biconnectivity and 3-edge connectivity. Sparsification also provided the first dynamic algorithm for 4-edge connectivity, k-edge connectivity, 3-vertex connectivity, 4-vertex connected components and bipartiteness. Later the technique was slightly improved in [EppGalIta93], then the results were summarized in [EppEtAl97]. In [AmaCatIta97] the first version was called **simple sparsification**, while the second version is called **improved sparsification**. Both are described in detail in our dissertation.

Supported operations There are three types of sparsification strategies:

- BASIC SPARSIFICATION can be used to dynamize static algorithms.

- STABLE SPARSIFICATION can be used to speed up existing fully dynamic algorithms.
- ASYMMETRIC SPARSIFICATION is useful in applications with more insertion than deletions for which partially dynamic algorithms exists to support insertions.

Performance In order to apply BASIC SPARSIFICATION we need to compute efficiently sparse certificates. If we note the time to find a sparse certificate by $f(n, m)$, the time needed to construct a data structure for testing the property by $g(n, m)$, which can answer queries in $q(n, m)$, then an update can be supported by BASIC SPARSIFICATION in $O(f(n, O(n)) \cdot \log(m/n) + g(n, O(n)))$ with simple sparsification and in $O(f(n, O(n)) + g(n, O(n)))$ with improved sparsification, and a query can be supported in $q(n, O(n))$.

STABLE SPARSIFICATION is useful, when we can maintain efficiently stable sparse certificates. This variant transforms time bounds of the form $O(m^p)$ to those of form $O(n^p)$. More generally, if we note by $f(n, m)$ the time needed to maintain a stable sparse certificate per update, for which there is a data structure to test the property with update time $g(n, m)$ and query time $q(n, m)$, then the same time bounds hold as in the case of BASIC SPARSIFICATION.

If we note by $f(n, m)$ the time needed to find a sparse certificate, by $g(n, m)$ the time needed to construct a partially dynamic data structure for testing the property, which can handle edge insertions in time $p(n, m)$ and answer queries in time $q(n, m)$, then by the means of ASYMMETRIC SPARSIFICATION we can give a fully dynamic data structure which supports edge insertions in $O(\frac{f(n,O(n))+g(n,O(n))}{n} + p(n, O(n)))$, edge deletions in $f(n, O(n)) \cdot O(\log(m/n)) + g(n, O(n))$ and queries in $q(n, O(n))$.

The memory needed to store the sparsification tree is $O(m)$ in case of simple sparsification. For improved sparsification an $O(m \log(n^2/m))$ bound is straightforward and can be improved to $O(m)$ for BASIC SPARSIFICATION and $O(\frac{m}{n} \cdot h(n))$ in case of STABLE SPARSIFICATION, where $h(n)$ is the space needed by a single node of the sparsification tree.

The preprocessing time is $O(m)$ for simple sparsification. For improved sparsification, obtaining $t_p = O(m \log(n^2/m))$ is trivial and can be optimized to $O(\frac{m}{n} \cdot h(n))$ in case of BASIC SPARSIFICATION and STABLE SPARSIFICATION, where $h(n)$ is the time needed processing a single node of the sparsification tree.

2.7 Euler Tour tree

Introduction Euler Tour trees were introduced in [HenKin99] as an ingredient for their fully dynamic connectivity algorithm, which was the first to achieve polylogarithmic bounds.

Supported operations Several operations can be carried out efficiently:

- TREE(u): return the root of the tree containing u.
- NONTREEEDGES(T): return a list of nontree edges incident to tree T. Edges with both endpoints in T are returned twice.
- INSERTTREE(u, v): inserts (u, v) as a new tree edge, connecting the tree containing u with the tree containing v.
- INSERTNONTREE(u, v): inserts nontree edge (u, v).
- DELETETREE(u, v): split the tree containing u and v by removing edge (u, v).
- DELETENONTREE(u, v): remove nontree edge (u, v).
- SAMPLEANDTEST(T): selects randomly a nontree edge incident to T and returns it if it has exactly one endpoint in T. Edges with both endpoints in T have twice as much probability to be selected.

Acronym of algorithm	t_p	t_u	t_q	t_a	Memory complexity
`DSF`	$\Theta(n)$	$O(\log n)$	$O(\log n)$	$O(\alpha(nr_q + n, n)))$	$\Theta(n)$
`KUF`	$\Theta(n)$	$O(\frac{\log n}{\log \log n})$	$O(\frac{\log n}{\log \log n})$	$O(\frac{\log n}{\log \log n})$	$\Theta(n)$

Figure 1: Comparison of incremental connectivity algorithms

Performance The connectivity algorithm form [HenKin99] uses two implementation of Euler Tour trees, the first with binary trees and the second with $\log n$-ary trees. In the first implementation NONTREEEDGES runs in $O(m' \log n)$ time, where m' is the size of the output, while the other operations need $O(\log n)$ time. In the second implementation DELETETREE and INSERTTREE are slowed down to $O(\log^2 n / \log \log n)$, TREE is improved to $O(\log n / \log \log n)$, and the other operations stay the same.

An Euler Tour tree can be stored in $O(n)$ space, with an additional $O(m)$ if nontree edges must be also maintained. Preprocessing time is $t_p = O(m \log n + n)$ ([AlbCatIta97]).

3 Undirected dynamic graph problems

3.1 Incremental connectivity

The incremental connectivity problem can be defined as follows. Given an undirected graph, initially containing n isolated nodes, the following operations must be supported:

- INSERT(u, v): adds an edge between nodes u and v.
- CONNECTED(u, v): returns *true* if nodes u and v are in the same connected component and *false* otherwise.

A comparative table of the algorithms presented in the dissertation is shown in Figure 1. For the meaning of notations and abbreviations see the Table of Notations and List of Acronyms from the beginning of this summary.

3.2 Decremental connectivity

The decremental connectivity problem can be defined as follows. Given an undirected graph $G(V, E)$, the following operations must be supported:

- DELETE(u, v): removes the edge between nodes u and v. It is assumed, that $(u, v) \in E$.
- CONNECTED(u, v): returns *true* if nodes u and v are in the same connected component and *false* otherwise.

For the algorithm by Even and Shiloach (`ES`) $t_p = \Theta(n + m), t_u = O(m), t_q = O(1), t_a = O(n)$ and the space usage is $\Theta(n + m)$. In the case of Thorup's algorithm (`ThoDec`) such bounds are much harder to give beside for t_a, which was proved in [Tho99], as they depend on the level of recursion the algorithm reaches and also on the underlying fully dynamic algorithm. Using the technique presented in [Tho00] the space complexity for each level of the recursion can be reduced to $O(m)$. Even then, the hidden constant is quite large, as several instances of graphs are stored on each level.

3.3 Fully dynamic connectivity

In fully dynamic connectivity the following operations must be supported:

- INSERT(u, v): adds an edge between nodes u and v. It is assumed, that $(u, v) \notin E$.
- DELETE(u, v): removes the edge between nodes u and v. It is assumed, that $(u, v) \in E$.
- CONNECTED(u, v): returns *true* if nodes u and v are in the same connected component and *false* otherwise.

In Figure 2 various running times are shown, where known.

In Figure 3 the preprocessing time and memory usage of the algorithms is compared. For `HDT` the memory usage refers to the one described for the original paper, which can be improved to $O(m)$ as described in [Tho00].

Acronym of algorithm	t_u	t_q	t_a	Average running time
`FredI-85`, `FredI-91`	$O(m^{2/3})$	$O(1)$	$O(m^{2/3})$	$O(\frac{n}{m^{1/3}} + \log n)$
`FredII-85`, `FredII-91`	$O(\sqrt{m \log m})$	$O(1)$	$O(\sqrt{m \log m})$	$O(\frac{n \cdot \sqrt{\log m}}{\sqrt{m}} + \log n)$
`FredIII-85`, `FredIII-91`	$O(\sqrt{m})$	$O(1)$	$O(\sqrt{m})$	$O(\frac{n}{\sqrt{m}} + \log n)$
`Spars(FredIII)`	$O(\sqrt{n})$	$O(1)$	$O(\sqrt{n})$	$O(\sqrt{n})$
`HK`	$O(m \log n)$	$O(\frac{\log n}{\log \log n})$	$O(\log^3 n)$	
`HT`, `HDT`	$O(m \log n)$	$O(\frac{\log n}{\log \log n})$	$O(\log^2 n)$	

Figure 2: Comparison of update and query times of fully dynamic connectivity algorithms

Acronym of algorithm	t_p	Memory complexity
`FredI-85`, `FredI-91`	$O(m)$	$O(m)$
`FredII-85`, `FredII-91`	$O(m)$	$O(m)$
`FredIII-85`, `FredIII-91`	$O(m)$	$O(m)$
`Spars(FredIII)`	$O(m)$	$O(m)$
`HK`	$O(m + n \log n)$	$O(m + n \log n)$
`HT`, `HDT`	$O(m + n \log n)$	$O(m + n \log n)$

Figure 3: Comparison of preprocessing times and memory usage of fully dynamic connectivity algorithms

3.4 Fully dynamic minimum spanning tree

We do not address specifically the incremental and decremental versions of the minimum spanning tree problem for the following reasons. The incremental problem can be easily solved in $O(\log n)$ per update using link-cut trees. On the other hand, solving the decremental version of the problem is one of the ingredients of the algorithm by Holm et. al described in the dissertation.

In the fully dynamic problem, given a weighted, undirected graph $G(V, E, W)$, we would like to support:

- INSERT(u, v, w): inserts an edge (u, v) in the graph with weight w. $(u, v) \notin E$ is assumed before the operation.

- REMOVE(u, v): removes edge (u, v) from the graph. We assume $(u, v) \in E$.

Acronym of algorithm	t_a	Memory complexity
`FredI-85`, `FredI-91`	$O(m^{2/3})$	$O(m)$
`FredII-85`, `FredII-91`	$O(\sqrt{m \log m})$	$O(m)$
`FredIII-85`, `FredIII-91`	$O(\sqrt{m})$	$O(m)$
`Spars(FredIII)`	$O(\sqrt{n})$	$O(m)$
`HDTMST`	$O(\log^4 n)$	$O(m \log n)$

Figure 4: Comparison of fully dynamic minimum spanning tree algorithms

- Change(u, v, w): changes the weight of edge (u, v) to w. We assume $(u, v) \in E$.
- Mst(): returns the cost of the minimum spanning tree of the current graph, and the edges it contains, if necessary. We use the term "tree" without loss of generality, even if it is actually a forest, if the graph is not connected.

We note, that Change(u, v, w) is not crucial, as it can be carried out with a sequence of Remove(u, v) and Insert(u, v, w).

In Figure 4 amortized running times per operation and memory usage of different fully dynamic minimum spanning tree algorithms are listed.

4 Directed dynamic graph problems

4.1 Dynamic transitive closure

We do not consider the partially dynamic and fully dynamic versions of the problem separately, as experiments have conclusively shown ([KroZar08]), that the currently best known theoretical fully dynamic algorithms are clearly inferior to simple-minded approaches and to hybridizations of partially dynamic algorithms. Thus, we present only the incremental and decremental algorithms with practical significance.

Given a directed graph $G(V, A)$, we give the following definitions.

Acronym of algorithm	t_p	t_t	Memory complexity
`Abd97`	$O(n \cdot m)$	$O(k^2 \cdot (m + nr_u) + (m + nr_u)^*)$	$O(k \cdot n)$
`Abd00`	$O(n \cdot m)$	$O(k \cdot (m + nr_u)^*)$	$O(k \cdot n)$
`Ital`	$O(n^2 + n \cdot m)$	$O(n \cdot (m + nr_u))$	$O(n^2)$
`Ital-Gen`	$O(n^2 + n \cdot m)$	$O(m^2)$	$O(n^2)$
`RZ`	$O(n^2 + n \cdot m)$	$O(n \cdot m)$	$O(n^2)$

Figure 5: Comparison of dynamic transitive closure algorithms

Definition 4.1 A node v is **reachable** by node u if and only if there is a directed path from u to v in G. $\square$

Definition 4.2 The digraph $G(V, A^*)$, that has the same node set with G but has an arc $(u, v) \in A^*$ if and only if v is reachable by u in G is called **transitive closure** of G. We shall denote $|A^*|$ by m^*. $\square$

Definition 4.3 If v is reachable from u (in G), then we call v a **descendant** or **successor** of u and u an **ancestor** or **predecessor** of v. $\square$

The operations to be supported are:

- Insert(u, v): adds the arc (u, v) into the graph.
- Remove(u, v): deletes the arc (u, v) from the graph.
- Reachable(u, v): returns *true* if there is a directed path from node u to node v and *false* otherwise.
- SearchPath(u, v): Returns a path from u to v, or $\emptyset$ if there is none.

In Figure 5 various complexities of dynamic transitive closure algorithms are shown. `Abd97` and `Abd00` are incremental only, and k is the number of node-disjoint paths the original graph is decomposed in. `Ital` is either incremental or decremental, the time bounds do not hold for a mixed sequence. The total expected time for `Ital-Gen` is for the decremental part, the incremental part having the same complexity as `Ital`. `RZ` is decremental only.

List of borrowings:

Borrower	Lender	Amount of money
1	2	10
2	3	5
3	1	5
1	4	5
4	5	10

Solution:

Sender	Reciever	Amount of money
1	5	10
4	2	5

Figure 6: Example for the debts' clearing problem

4.2 Fully dynamic debt clearing

Introduction In this section we discuss an original problem proposed in 2008 by us at the qualification contest of the Romanian national team of informatics for the Central European Olympiad of Informatics and Balkan Olympiad of Informatics.

The problem statement is the following:

Let us consider a number of n entities (eg. persons, companies), and a list of m borrowings among these entities. A borrowing can be described by three parameters: the index of the borrower entity, the index of the lender entity and the amount of money that was lent. The task is to find a minimal list of money transactions that clears the debts formed among these n entities as a result of the m borrowings made.

In [Pat09] we model this problem using graph theory:

Definition 4.4 Let $G(V, A, W)$ be a directed, weighted multigraph without loops, $|V| = n$, $|A| = m$, $W : A \to \mathbb{Z}$, where V is the set of vertices, A is the set of arcs and W is the weight function. G represents the borrowings made, so we will call it the **borrowing graph**. □

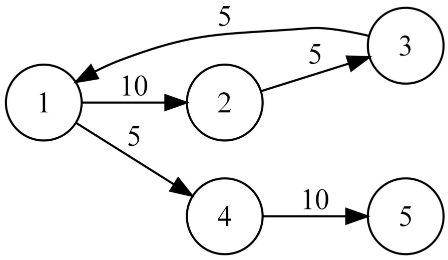

Figure 7: The borrowing graph associated with the given example. An arc from node i to node j with weight w means, that entity i must pay w amount of money to entity j.

The borrowing graph corresponding to the example from Figure 6 is depicted in Figure 7.

Definition 4.5 Let us define for each vertex $v \in V$ the **absolute amount of debt** over the graph G: $D_G(v) = \sum_{\substack{v' \in V \\ (v,v') \in A}} W(v,v') - \sum_{\substack{v'' \in V \\ (v'',v) \in A}} W(v'',v)$

Sometimes for simplicity we will refer to the absolute amount of debt of a node as D **value**. □

Definition 4.6 Let $G'(V, A', W')$ be a directed, weighted multigraph without loops, with each arc (i, j) representing a transaction of $W'(i, j)$ amount of money from entity i to entity j. We will call this graph a **transaction graph**. These transactions clear the debts formed by the borrowings modeled by graph $G(V, A, W)$ if and only if:

$D_G(v_i) = D_{G'}(v_i), \forall i = \overline{1, n}$, where $V = \{v_1, v_2, \ldots, v_n\}$

We denote this by: $G \sim G'$. □

See Figure 8 for a transaction graph with minimal number of arcs corresponding to the example from Figure 6.

Using the terms defined above, the debt's clearing problem can be reformulated as follows:

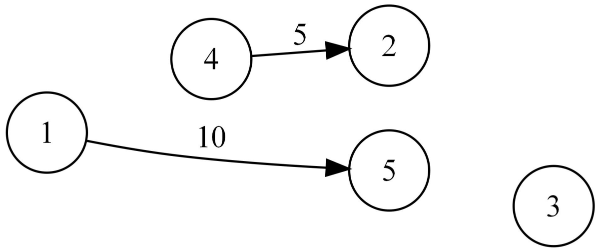

Figure 8: The respective minimum transaction graph. An arc from node i to node j with weight w means, that entity i pays w amount of money to entity j.

Given a borrowing graph $G(V, A, W)$ we are looking for a minimal transaction graph $G_{min}(V, A_{min}, W_{min})$, so that $G \sim G_{min}$ and $\forall G'(V, A', W') : G \sim G', |A_{min}| \leq |A'|$ holds.

The problem's relation to complexity classes Let us denote the optimization problem described in the introduction as DEBT. We will call the corresponding decision problem DEBT-DECISION, defined as follows:

Given a borrowing graph $G(V, A, W)$ and a natural number $M \leq |A|$, is there a transaction graph $G'(V, A', W'), G \sim G'$, so that $|A'| \leq M$?

Lemma 4.7 DEBT-DECISION *is NP.* □

Lemma 4.8 SUBSET SUM *is reducible to* DEBT-DECISION. □

Theorem 4.9 DEBT-DECISION *is NP-complete.* □

Corollary 4.10 DEBT *is NP-hard.* □

Lemma 4.11 3-PARTITION *is pseudo-polynomially transformable in* DEBT-DECISION. □

Theorem 4.12 DEBT-DECISION *is NP-complete in the strong sense.* □

Corollary 4.13 DEBT *is NP-hard in the strong sense.* □

Let us define the problem DEBT-DECISION-PARTIAL as follows:

Given a borrowing graph $G(V, A, W)$, a "partial graph" $G^p(V, A^p, W^p)$ and a natural number $M \leq |A|$, can G^p "completed" to a transaction graph with at most M arcs? More formally is there a transaction graph $G'(V, A', W'), G \sim G'$, so that $|A'| \leq M$ and $A^p \subset A, W^p(a) = W'(a), \forall a \in A^p$?

Lemma 4.14 DEBT-DECISION-PARTIAL *is NP.* □

Lemma 4.15 ([Pat11b]) DEBT *is Turing reducible to* DEBT-DECISION-PARTIAL. □

Theorem 4.16 ([Pat11b]) DEBT *is NP-easy.* □

Corollary 4.17 DEBT *is NP-equivalent.* □

A restricted version Let us define the problem DEBT-PATH as follows:

Given a borrowing graph $G(V, A, W)$, whose arcs form a path, find the minimum transaction graph $G'(V, A', W'), G \sim G'$. More formally $A = \bigcup_{i=1}^{n-1} \{(v_{p_i}, v_{p_{i+1}})\}$, $v_{p_i} = v_{p_j} \Rightarrow i = j, \forall i, j = \overline{1, n}$.

Theorem 4.18 ([Pat11b]) DEBT-PATH *is NP-hard.* □

Theorem 4.19 ([Pat11b]) DEBT-PATH *is NP hard in the strong sense.* □

Theorem 4.20 DEBT-PATH *is NP-easy.* □

Corollary 4.21 DEBT-PATH *is NP-equivalent.* □

A solution based on dynamic programming We give a solution using the dynamic programming method. It uses similar techniques to the algorithm discovered independently by Bellman ([Bel62]), respectively Held and Karp ([HelKar62]) for solving the Traveling Salesman Problem.

The following observation is crucial in our solutions.

Theorem 4.22 ([PatBar11]) *Any instance of the debt clearing problem can be solved trivially by at most $n - 1$ transactions.* □

Let us denote by V_{left} the set of nodes having positive D values and by V_{right} the set of nodes having negative D values, formally $V_{left} = \{u|D(u) > 0\}$, $V_{right} = \{u|D(u) < 0\}$. Let $n_1 = |V_{left}|$, $n_2 = |V_{right}|$ and $V_{left} = \{left_1, \ldots, left_{n_1}\}$, $V_{right} = \{right_1, \ldots, right_{n_2}\}$. Let us define the subproblems of the dynamic programming problem with two parameters i and j, where i is a binary representation of n_1 bits, and j is a binary representation of n_2 bits ($i = \overline{0, 2^{n_1} - 1}, j = \overline{0, 2^{n_2} - 1}$). A subproblem will have the following meaning:

$dp_{i,j}$ = the number of arcs in the minimal transaction graph containing only the nodes from V_{left} determined by the bits of i and the nodes from V_{right} determined by the bits of j.

The recursive formula to determine the values of the subproblems is the following[2]:

$dp_{i,j} = \min(dp_{i \texttt{ XOR } i', j \texttt{ XOR } j'} + bitcount(i') + bitcount(j') - 1)$, where

1. $i \texttt{ AND } i' = i'$
2. $j \texttt{ AND } j' = j'$
3. $\sum\limits_{i' \texttt{ AND } 2^k \neq 0} D(left_k) = - \sum\limits_{j' \texttt{ AND } 2^k \neq 0} D(right_k)$
4. $bitcount(x)$ returns the number of bits of x equal to 1.

Let us analyze the performance of the proposed algorithm. The number of subproblems is $2^{n_1} \cdot 2^{n_2} = 2^{n_1+n_2}$, which in the worst case is 2^n. Thus the space complexity of our algorithm is $\Theta(2^n)$. To solve a subproblem (i, j) we need all the pairs (i', j'), such that i' is a subset of i and j' is a subset of j. We can codify any pair (i, i') with a sequence of length n_1 of ternary digits. A digit will be 0, if the respective node is not in i, 1 if it is in i but not in i' and 2 if it is in i' (and thus also in i). The same codification can be done for any (j, j') pair. Thus the number of steps performed by our algorithm is proportional to $3^{n_1} \cdot 3^{n_2} = 3^n$

[2]We note by AND the bitwise and operation and by XOR the bitwise exclusive or operation

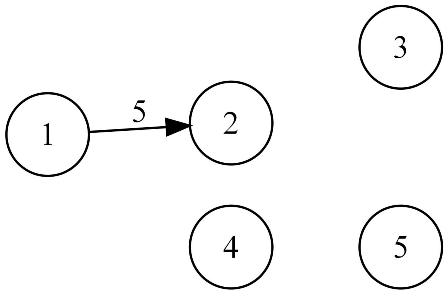

Figure 9: Result of the QUERY operation called after the third arc was added

The debts' clearing problem in dynamic graphs In the dynamic debts' clearing problem ([Pat11]) we want to support the following operations:

- INSERTNODE(u) - adds a new node u to the borrowing graph.
- REMOVENODE(u) - removes node u from the borrowing graph. In order for a node to be removed, all of its debts must be cleared first. In order to affect the other nodes as little as possible, the debts of u will be cleared in a way that affects the least number of nodes, without compromising the optimal solution for the whole graph.
- INSERTARC(u, v, x) - insert an arc in the borrowing graph. That is, u must pay x amount of money to v.
- REMOVEARC(u, v) - removes the debt between u and v.
- QUERY() - returns a minimal transaction graph.

For instance calling the QUERY operation after adding the third arc in the borrowing graph corresponding to Figure 6 would result in the minimal transaction graph from Figure 9.

A data structure for solving dynamic debts' clearing As the static version of the problem is NP-hard, it is not possible to support all these operations in polynomial time (unless P = NP). Otherwise we could just build up the whole graph one arc at a time, by m calls of InsertArc, then construct a minimal transaction graph by a call of Query, which would lead to a polynomial algorithm for the static problem.

Our data structure used to support these operations is based on maintaining the subset of nodes, that have non-zero absolute amount of debt $V^* = \{u|D(u) \neq 0\}$. The sum of D values for all the $2^{|V^*|}$ subsets of V^* is also stored in a hash table called *sums*.

InsertNode As for our data structure only nodes having non-zero D values are important, and a new node always starts with no debts, it means that nothing has to be done when calling InsertNode.

InsertArc When InsertArc is called, the D values of the two nodes change, so V^* can also change. When a node leaves V^*, we use a lazy updating scheme for the subsets it is contained in, because when a new node enters V^* we have to calculate the sum of all of the subsets it is contained in anyway.

If both u and v were in V^* and remained in it after changing the D values, then we simply add x to the sum of all subsets containing u, but not v, and subtract x from those containing v but not u. The sum of the subsets containing both nodes does not change.

If one of the nodes was just added to V^* ($D[u] = x$, or $D[v] = -x$), then all the sums of the subsets containing it must be recalculated. This recalculation can be done in $O(1)$ for each subset, taking advantage of sums already calculated for smaller subsets.

Query To carry out Query we observe, that finding a minimal transaction graph is equivalent to partitioning V^* in a maximal number of disjoint zero-sum subsets, more formally $V^* = P_1 \cup \ldots \cup P_{max}, sums[P_i] = 0, \forall i = \overline{1, max}$ and $P_i \cap P_j = \emptyset, \forall i, j = \overline{1, max}, i \neq j$. The reason for this is, that all the debts in a zero-sum subset P_i can be cleared by $|P_i| - 1$ transactions (by Theorem 4.22, also see [Pat09, Pat11b, Ver04]), thus to clear all the debts, $|V^*| - max$ transactions are necessary.

Let S^0 be the set of all subsets of V^*, having zero sum: $S^0 = \{S | S \subset V^*, sums[S] = 0\}$. Then, to find the maximal partition, we use dynamic programming.

Let $dp[S]$ be the maximal number of zero-sum sets, $S \subset V^*$ can be partitioned in.

$$dp[S] = \begin{cases} \text{not defined}, & \text{if } sums[S] \neq 0 \\ 0, & \text{if } S = \emptyset \\ \max\{dp[S \setminus S'] + 1 | S' \subset S, S' \in S^0\}, & \text{otherwise} \end{cases}$$

Building dp takes at most $2^{|V^*|} \cdot |S^0|$ steps.

As the speed at which Query can be carried out depends greatly on the size of S^0, we can use two heuristics to reduce its size, without compromising the optimal solution. To facilitate the running time of these heuristics, S^0 can be implemented as a linked list.

Clear pairs Choosing sets containing exactly two elements in the partition will never lead to a suboptimal solution, if the remaining elements are partitioned correctly ([Ver04]). Thus, before building dp, sets having two elements can be removed from S^0, along with all the sets, that contain those two elements (because we already added them to the solution, so there is no need to consider sets that contain them in the dynamic programming): $S^0 := S^0 \setminus (\{u, v\} \cup \{S' | u \in S' \text{ or } v \in S'\})$. The running time of this heuristic is $\Theta(|S^0|)$.

Clear non-atomic sets If a set $S_i \in S^0$ is contained in another set $S_j \in S^0$, then S_j can be safely discarded, because $S_j \setminus S_i$ will also be part of S^0, and combining S_i with $S_j \setminus S_i$ always leads to a better solution, than using S_j alone: $S^0 := S^0 \setminus \{S_j | \exists S_i \in S^0 : S_i \subset S_j\}$. This heuristic can be carried out in $\Theta(|S^0|^2)$.

RemoveNode To delete a node u with the conditions listed in the introduction is equivalent to finding a set P of minimal cardinality containing u, that can still be part of an optimal partition, that is $dp[V^*] = dp[V^* \setminus P] + 1$.

This algorithm can not be used together with the **Clear pairs** heuristic, because clearing pairs may compromise the optimal removal of u. The running time is the same as for QUERY, because dp must be built.

RemoveArc Because clearing an arc between two nodes is the same as adding an arc in the opposite direction, this can be easily implemented using INSERTARC. If the D values of the two nodes have the same sign, it means, that no arc could appear in a minimal transaction graph between the two nodes, so nothing has to be done.

A new algorithm for the static problem We can observe, that the QUERY operation needs only the set S^0 to be built, and in order to build S^0 the sum of all subsets of V^* needs to be calculated. Thus, after processing all the arcs in $\Theta(m)$ time and finding the D values, the *sums* hash table can be built in $\Theta(2^|V^*|)$ by dynamic programming:

$$sums[S = \{s_1, \dots s_k\}] = \begin{cases} 0, & \text{if } S = \emptyset \\ D[s_1], & \text{if } |S| = 1 \\ sums[\{s_2, \dots s_k\}] + D[s_1], & \text{otherwise} \end{cases}$$

After *sums* is built, we can construct S^0 by simply iterating once again over all the subsets of V^* and adding zero-sum subsets to S^0. Then we clear pairs and non-atomic sets, call QUERY and we are done. This yields to a total complexity of $\Theta(m + 2^{|V^*|} + |S^0|^2 + 2^{|V^*|} \cdot |S^0|)$.

Practical behavior As it can be seen from the time complexities of the operations, the behavior of the presented algorithms depends on the cardinalities of V^* and S^0 and their running times may vary from case to case.

We have made some experiments to compare our new algorithms and the static algorithm presented in [Pat09]. We used the same 15 test cases which were used, when the problem was proposed in 2008 at the qualification contest of the Romanian national team. Figure 10 contains the structure of the graphs used for each test case.

In our first experiment we compared three algorithms: the old static

Test	n	m	$\|A_{min}\|$	Short description
1	20	19	1	A path with the same weight on each arc
2	20	20	0	A cycle with the same weight on each arc
3	8	7	7	Minimal transaction graph equals to borrowing graph
4	20	19	19	Two connected stars
5	20	15	15	Yields to $D[i] = 2, \forall i = \overline{1,10}$, $D[i] = -1, \forall i = \overline{11,19}$ and $D[20] = -11$, maximizing the number of triples (zero-sets with cardinality three)
6	20	10	10	Yields to $D[i] = 99, \forall i = \overline{1,10}$, $D[i] = -99, \forall i = \overline{11,20}$, maximizing the number of pairs
7	20	19	12	A path with random weights having close values (50 ± 10)
8	20	20	10	A cycle with random weights having close values (50 ± 10)
9	10	100	7	Random graph with weights ≤ 10
10	12	100	9	Random graph with weights ≤ 10
11	15	100	11	Random graph with weights ≤ 10
12	20	100	14	Random graph with weights ≤ 10
13	20	19	15	A path with consecutive weights
14	20	30	15	Ten pairs, a path, a star and triples put together
15	20	100	15	Dense graph with weights ≤ 3

Figure 10: The structure of the test cases

algorithm based on dynamic programming, our new static algorithm and the dynamic graph algorithm. For the third algorithm we called InsertArc for each arc, then Query once in the end, after all arcs were added.

In the second experiment we used the same methodology to compare the old static algorithm and our new dynamic algorithm. For the first algorithm the solution was recomputed from scratch each time an arc was read from the input file, and for the second after each InsertArc a Query was also executed. Detailed results can be found in our dissertation.

To better understand these results, we performed additional experiments. First, we compared the running time of the two static algorithms on randomly generated graphs having $n = 16$ nodes and $m = 20$ arcs having costs from the $[1, MAXVALUE)$ interval. For every even $MAXVALUE \in [2, 80]$ we

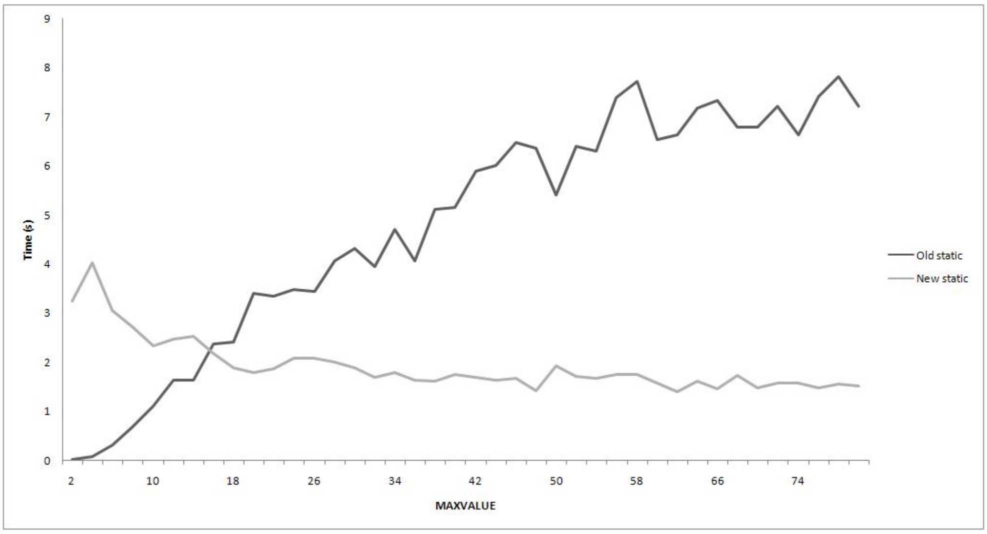

Figure 11: Total running times of the two static algorithms over 1000 random instances of graphs having $n = 16$ nodes, $m = 20$ arcs and arc costs less than $MAXVALUE$.

generated 1000 different graphs and executed both algorithms on them. As it can be seen in Figure 11 for small values of $MAXVALUE$ the old static algorithm is faster, but from $MAXVALUE = 16$ it becomes slower and slower. We also note that the new algorithm is more robust, as its running time does not fluctuate as wildly as in the case of the old algorithm.

To better understand the inner details of the algorithms, for instance why the old algorithm gets slower as $MAXVALUE$ increases, we measured separately the time spent in each phase of the algorithms.

In this experiment we generated 10000 random graphs having $n = 16$ nodes and $m = 20$ arcs and calculated the average running time of each phase for both algorithms. We investigated two cases, one for which the old algorithm is faster (Figure 12) and the other in which the new algorithm is faster (Figure 13).

The running time of the old static algorithm is dominated by the preprocessing time in both cases. A further investigation reveals, that the memory

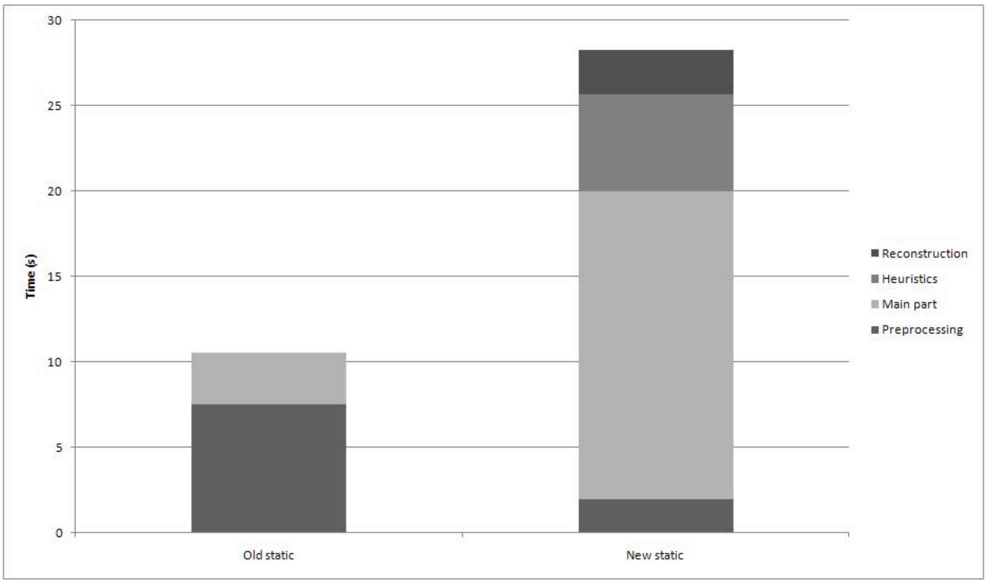

Figure 12: Total running times of various phases of the algorithms over 10000 random instances of graphs having $n = 16$ nodes, $m = 20$ arcs and arc costs less than 10.

allocation part is the bottleneck of this algorithm. Even if both algorithms have to allocate $\Theta(2^{|V^*|})$ memory, it seems like allocating matrices of this size is much more time consuming than allocating vectors of the same total size. This could explain the fact, that the new algorithm has much smaller preprocessing time.

When $MAXVALUE = 10$, there is a bigger probability that pairs can be found in the preprocessing phase of the old algorithm, thus reducing the total amount of memory that needs to be allocated, explaining the increased running time for $MAXVALUE = 50$. This was confirmed by our experiments, the average size of V^* after removing pairs being 9.8 and respectively 13.3.

The new algorithm behaves as expected, spending significantly more time in the main part and the heuristics phase for $MAXVALUE = 10$. The reason is the bigger cardinality of S^0 on average, which was about four times greater compared to $MAXVALUE = 50$ (615.5 and 153.2 respectively on

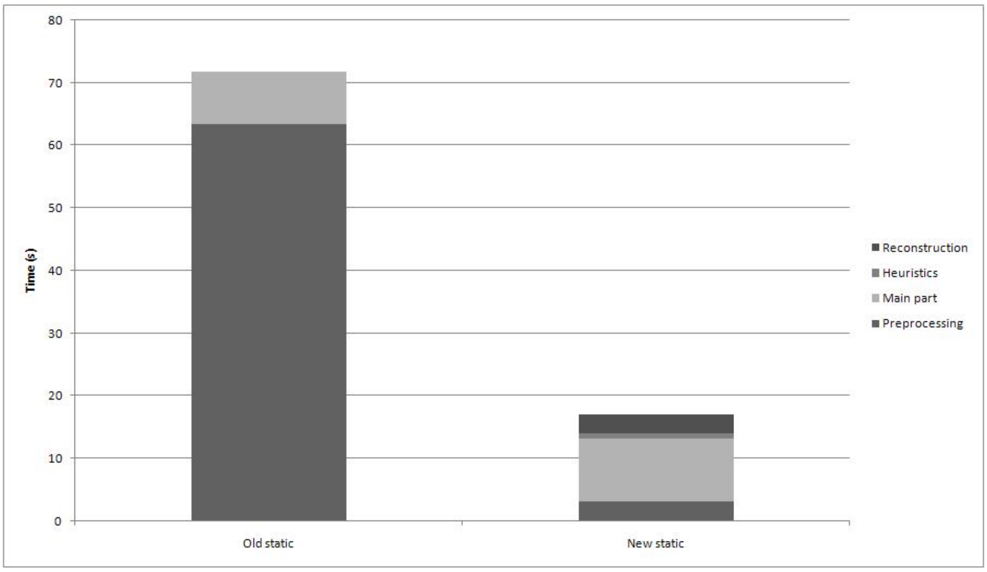

Figure 13: Total running times of various phases of the algorithms over 10000 random instances of graphs having $n = 16$ nodes, $m = 20$ arcs and arc costs less than 50.

average).

Our experiments are not meant to be an exact comparison among the algorithms, as the running time can greatly depend on the details of the implementation. Their purpose was just to get a general overview of the behavior of the various algorithms for different kind of graphs.

Managing large instances Let us denote with n^* the number of nodes having a non zero D value, formally $n^* = |V^*| = |\{u|D(u) \neq 0\}|$. The algorithms presented so far can find an optimal solution in reasonable time only for instances of the magnitude of 20 - 30 for n^*. It could be also desirable to find "sufficiently good" solutions for larger inputs.

As for many intractable problems, techniques from the field of artificial intelligence are useful to obtain good solutions. We propose a genetic algorithm ([Hol75]) to solve the debts' clearing problem ([PatBar11]).

We use the reformulation of the problem of partitioning V^* into disjoint zero-sum sets.

Representation A solution of the problem is represented by a permutation of the D values of V^*, the set of nodes. Thus a candidate solution is a vector $C = (c_1, c_2, \ldots, c_n)$, such that $c_i = D(u), \forall i \in \overline{1, n}$ for some unique $u \in V^*$.

The idea of permutation representations is used intensively in solutions of the Traveling Salesman Problem ([GolLin85, OliSmiHol87, WhiStaFuq89]).

Fitness assignment To evaluate the fitness of a chromosome, we iterate over the genes of the chromosome in increasing order and maintain the partial sum obtained so far, that is $s_i = \sum\limits_{j=1}^{i} c_j$. For every $s_i = 0$, we have found a new zero-sum subset of the partition (starting after the last encountered partial sum equal to zero and ending at i), so we can add one to the fitness of the chromosome.

Recombination Various operators for permutation representations are discussed in [Dav85, GolLin85, Gor90, OliSmiHol87, Sys91, WhiStaFuq89]. We propose new recombination operators ([PatBar11]).

Operator 1 Let C_1 and C_2 be the two chromosomes, and $k \in [1, n]$ a random index. Then, the first descendant C_1' can be obtained by copying the first k genes from C_1 and appending to it the elements of the permutation not used so far in the same order as they appear in C_2. The second descendant C_2' is obtained symmetrically.

Operator 2 The problem with Operator 1 is, that the first descendant inherits most of its properties from C_1 and very little from C_2. Symmetrically C_2' inherits most of its properties from C_2 and very little from C_1. This is undesirable, as both C_1 and C_2 can contain subsets from the optimal partition.

A better recombination operator may be the following. First, determine the partitions codified by C_1 and C_2, as described at the evaluation of the fitness function. Let those be $C_1 = P_{1,1} \cup P_{1,2} \cup \ldots$ and $C_2 = P_{2,1} \cup P_{2,2} \cup \ldots$. Initialize $C_1' := C_1$ and $C_2' := C_2$.

Then, iterate over every $P_{1,i}$. If some $P_{1,i}$ is contained in some $P_{2,j}$, that is $P_{1,i} \subset P_{2,j}$, replace $P_{2,j}$ in the second descendant with $P_{1,i} \cup (P_{2,j} \setminus P_{1,i})$. Repeat the same procedure for C_2 symmetrically.

Mutation Four new mutation operators are proposed (Operators 3–6), having the property, that the fitness of the chromosome does not decrease ([PatBar11]).

Operator 1 The inversion operator described by Holland ([Hol75]) can be used without modification, on the sequence between the i^{th} and j^{th} elements.

Operator 2 A simplified version of Operator 1 can be easily carried out, by swapping the place of genes i and j in the chromosome.

Operators 3 and 4 Operators 1 and 2 can be used on the partition $C = P_1 \cup P_2 \cup \ldots$ instead of the permutation representation. This method guarantees that the fitness of the chromosome does not decrease.

Operators 5 and 6 Operators 1 and 2 can also be used inside some P_k without decreasing the fitness.

Because of the strongly NP-hardness of the problem, it is challenging to generate large test cases for which information about the optimal solution is known. In our dissertation we describe four methods to generate large test cases.

Forbidden transactions, interest rates, discounts It is natural to assume, that transactions are not possible between any pair of entities because of personal, practical, economical or other reasons.

An instance of such a problem can be described by two graphs, the borrowing graph G and the permission graph G_P defined below.

Definition 4.23 A **permission graph** $G_P(V, A_P)$ is a directed unweighted graph, which has an arc $(u, v) \in A_P$ if a transaction from u to v is allowed. $\square$

The original version of the problem corresponds to a permission graph equal to the complete graph. It can be easily seen, that by introducing the permission graph we generalized the original problem, thus this version is also NP-hard in the strong sense. Furthermore, the algorithms described above cannot be easily adjusted to solve this more general version and it seems that finding such algorithms is a difficult task.

To make our model even more realistic we can make the permission graph weighted and impose that any amount of money paid by u to v will be multiplied by the weight of the corresponding arc (u, v) in the permission graph. Thus a weight bigger than one would mean a discount given to u by v, and a weight smaller than one an interest rate. In this version of the problem we can ask to minimize the sum of the money paid by all entities.

Bibliography (selection)

[Abd97] Saïd Abdeddaïm. *On incremental computation of transitive closure and greedy alignment.* In *Combinatorial Pattern Matching, 8th Annual Symposium*, volume 1264 of *Lecture Notes in Computer Science*, 167–179. 1997.

[Abd00] Saïd Abdeddaïm. *Algorithms and experiments on transitive closure, path cover, and multiple sequence alignment.* In *Proceedings of the 2nd Algorithm Engineering and Experiments*, 157–169. 2000.

[AlbCatIta97] David Alberts, Giuseppe Cattaneo and Giuseppe F. Italiano. *An empirical study of dynamic graph algorithms.* ACM Journal of Experimental Algorithmics, volume 2, 1997.

[AlbEtAl98] David Alberts, Giuseppe Cattaneo, Giuseppe F. Italiano, Umberto Nanni and Christos D. Zaroliagis. *A software library of dynamic graph algorithms.* In *Proceedings of Algorithms and Experiments*, 129–136. 1998.

[AlbHen98] David Alberts and Monika R. Henzinger. *Average-case analysis of dynamic graph algorithms.* Algorithmica, volume 20(1):31–60, 1998.

[AlsBenRau99] Stephen Alstrup, Amir M. Ben-Amram and Theis Rauhe. *Worst-case and amortised optimality in union-find.* In *Proceedings of the 31st Annual ACM Symposium on Theory of Computing*, 499–506. ACM Press, 1999.

[AmaCatIta97] Giuseppe Amato, Giuseppe Cattaneo and Giuseppe F. Italiano. *Experimental analysis of dynamic minimum spanning tree algorithms.* In *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms*, 314–323. 1997.

[Bel62] Richard Bellman. *Dynamic programming treatment of the travelling salesman problem.* Journal of the ACM, volume 9(1):61–63, 1962.

[Blu85] Norbert Blum. *On the single-operation worst-case time complexity on the disjoint set union problem.* In *2nd Symposium of Theoretical Aspects of Computer Science*, volume 182 of *lncs*, 32–38. Springer, 1985.

[CorLeiRiv90] Thomas H. Cormen, Charles E. Leiserson and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[CorEtAl01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. *Introduction to Algorithms, Second Edition.* The MIT Press and McGraw-Hill Book Company, 2001.

[Dav85] Lawrence Davis. *Applying adaptive algorithms to epistatic domains.* In *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, 162–164. Morgan Kaufmann, 1985.

[DemFinIta04] Camil Demetrescu, Irene Finocchi and Giuseppe F. Italiano. *Dynamic graph algorithms.* In *Handbook of Graph Theory.* CRC Press, 2004.

[EppGalIta93] David Eppstein, Zvi Galil and Giuseppe F. Italiano. *Improved sparsification.* Technical Report 93-20, Department of Information and Computer Science, University of California at Irvine, 1993.

[EppEtAl92] David Eppstein, Zvi Galil, Giuseppe F. Italiano and Amnon Nissenzweig. *Sparsification - A technique for speeding up dynamic graph algorithms (extended abstract).* In

Proceedings of the 33rd Annual Symposium on Foundations of Computer Science, 60–69. IEEE Computer Society Press, 1992.

[EppEtAl97] David Eppstein, Zvi Galil, Giuseppe F. Italiano and Amnon Nissenzweig. *Sparsification - a technique for speeding up dynamic graph algorithms.* Journal of the ACM, volume 44(5):669–696, 1997.

[Fis72] Michael J. Fischer. *Efficiency of equivalence algorithms.* In *Proceedings of the Workshop on Complexity of Computations*, 153–167. Plenum Press, 1972.

[Fre83] Greg N. Frederickson. *Data structures for on-line updating of minimum spanning trees.* In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing*, 252–257. ACM Press, 1983.

[Fre85] Greg N. Frederickson. *Data structures for on-line updating of minimum spanning trees, with applications.* SIAM Journal on Computing, volume 14(4):781–798, 1985.

[Fre97] Greg N. Frederickson. *Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees.* SIAM Journal on Computing, volume 26(2):484–538, 1997.

[GarJoh79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, 1979.

[GolLin85] David E. Goldberg and Robert Lingle, Jr. *Alleles, loci, and the traveling salesman problem.* In *Proceedings of the 1st International Conference on Genetic Algorithms and their Applications*, 154–159. Lawrence Erlbaum Associates, 1985.

[Gor90] MARTINA GORGES-SCHLEUTER. *Genetic algorithms and population structure - A massively parallel algorithm.* Ph.D. thesis, University of Dortmund, 1990.

[Har69] FRANK HARARY. *Graph Theory.* Addison-Wesley, 1969.

[HarGup97] FRANK HARARY and GOPAL GUPTA. *Dynamic graph models.* Mathematical and Computer Modelling, volume 25(7):79–87, 1997.

[HelKar62] MICHAEL HELD and RICHARD M. KARP. *A dynamic programming approach to sequencing problems.* Journal of the Society for Industrial and Applied Mathematics, volume 10(1):196–210, 1962.

[HenKin99] MONIKA R. HENZINGER and VALERIE KING. *Randomized fully dynamic graph algorithms with polylogarithmic time per operation.* Journal of the ACM, volume 46(4):502–516, 1999.

[Hol75] JOHN H. HOLLAND. *Adaptation in Natural and Artificial Systems.* The University of Michigan Press, 1975.

[IonPat08] KLÁRA IONESCU and CSABA G. PĂTCAŞ. *Improving the performance of algorithms using the principles of binary search.* Technical Review, volume 43(3):7–14, 2008.

[KroZar08] IOANNIS KROMMIDAS and CHRISTOS D. ZAROLIAGIS. *An experimental study of algorithms for fully dynamic transitive closure.* ACM Journal of Experimental Algorithmics, volume 12, 2008.

[Kru56] JOSEPH B. KRUSKAL. *On the shortest spanning subtree of a graph and the traveling salesman problem.* Proceedings of the American Mathematical Society, volume 7:48–50, 1956.

[OliSmiHol87] I. Oliver, D. Smith and J. Holland. *A study of permutation crossover operators on the traveling salesman problem.* In *Proceedings of the Second International Conference on Genetic Algorithms*, 224–230. Lawrence Erlbaum Associates, 1987.

[Pat09] Csaba G. Pătcaş. *On the debts' clearing problem.* Studia Universitatis Babeş-Bolyai Series Informatica, volume 54(2):109–120, 2009.

[Pat11] Csaba G. Pătcaş. *The debts' clearing problem: a new approach.* Acta Universitatis Sapientiae Informatica (accepted), 2011.

[Pat11b] Csaba G. Pătcaş. *The debts' clearing problem's relation with complexity classes.* Studia Universitatis Babeş-Bolyai Series Informatica (accepted), 2011.

[PatBar11] Csaba G. Pătcaş and Attila Bartha. *Evolutionary solving of the debts' clearing problem.* (submitted), 2011.

[PatIon08] Csaba G. Pătcaş and Klára Ionescu. *Algorithmics of the knapsack type tasks.* Teaching Mathematics and Computer Science, volume 6(INFODIDACT):37–71, 2008.

[Sys91] Gilbert Syswerda. *Schedule optimization using genetic algorithms.* In *Handbook of Genetic Algorithms*, 332–349. Van Nostrand Reingold, 1991.

[Tar75] Robert E. Tarjan. *Efficiency of a good but not linear set union algorithm.* Journal of the ACM, volume 22(2):448–501, 1975.

[TarVis85] Robert E. Tarjan and Uzi Vishkin. *An efficient parallel biconnectivity algorithm.* SIAM Journal on Computing, volume 14(4):862–874, 1985.

[Tho99] MIKKEL THORUP. *Decremental dynamic connectivity.* Journal of Algorithms, volume 33(2):229–243, 1999.

[Tho00] MIKKEL THORUP. *Near-optimal fully-dynamic graph connectivity.* In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing*, 343–350. ACM Press, 2000.

[Ver04] TOM VERHOEFF. *Settling multiple debts efficiently: An invitation to computing science.* Informatics in Education, volume 3(1):105–126, 2004.

[WhiStaFuq89] DARRELL WHITLEY, TIMOTHY STARKWATER and D'ANN FUQUAY. *Scheduling problems and traveling salesmen: The genetic edge recombination operator.* In *Proceedings of the Third International Conference on Genetic Algorithms*, 133–140. Morgan Kaufmann Publishers, 1989.

[Zar02] CHRISTOS ZAROLIAGIS. *Implementations and experimental studies of dynamic graph algorithms.* In *Experimental Algorithmics*, volume 2547, 229–278. 2002.