University of West Bohemia
Faculty of Applied Science
Department of Mathematics

Diploma Thesis

Spatial Complexity of Graph Problems

Plzeň, 2012

Michal Hanzlík

This page was intentionally left blank for the assignment.

Declaration

I hereby declare that this Diploma Thesis is the result of my own work and that all external sources of information have been duly acknowledged.

In Pilsen 23. May 2012 ....

Michal Hanzlík

Acknowledgments

I would like to thank my supervisor Doc. RNDr. Tomáš Kaiser, Ph.D. for suggesting the topic, his enthusiastic attitude and for several inspiring discussions.

I would also like to thank Doc. RNDr. Daniel Král', Ph.D. for his comments during the initial discussions about the thesis.

Abstract

This diploma thesis is divided into three parts. In the first part we review some basic facts of the computational complexity theory, present a standard model of computation called Turing machine and use it to define some complexity classes.

In the next part we take a closer look at the logarithmic space complexity classes and especially the classes $\mathcal{NL}$ and $\mathcal{UL}$. It is believed that these classes are equal but the problem remains open. At the end of this part we recall some techniques for working with classes using logarithmic space.

In the last part we prove the main result of this thesis: the directed connectivity problem on a restricted subclass of grid graphs belongs to $\mathcal{UL}$.

Keywords

Computational complexity theory, space complexity, logarithmic space, log-space, unambiguous log-space, grid graphs, min-unique graphs, directed connectivity problem

Abstrakt

Tato diplomová práce je rozdělena do tří částí. V první části připomeneme několik základních faktů z teorie výpočetní složitosti, zavedeme standardní výpočetní model zvaný Turingův stroj a pomocí něho definujeme několik složitostních tříd.

V další části se budeme podrobněji zabývat problémy s logaritmickou prostorovou složitostí a obzvlášť třídami $\mathcal{NL}$ a $\mathcal{UL}$. Panuje obecné přesvědčení o tom, že se tyto třídy sobě rovnají, ale problém zůstává otevřeným. Na konci této části připomeneme některé techniky, které se využívají při práci s třídami problémů řešitelných v logaritmickém prostoru.

V poslední části dokážeme hlavní výsledek této práce: tvrzení, že problém souvislosti orientovaného grafu na podmnožině třídy mřížkových grafů patří do $\mathcal{UL}$.

Klíčová slova

Teorie výpočetní složitosti, prostorová složitost, logaritmický prostor, log-space, jednoznačný log-space, mřížkové grafy, min-unique grafy, problém souvislosti orientovaného grafu

Contents

1 Introduction

The topic of this thesis is computational complexity theory. Our main topic of interest will be complexity classes using logarithmic space. In Chapter 2 we provide an overview of some elementary concepts and results in this field. Namely, in Section 2.1.2 we introduce Turing machines as a computational model and use them to define time and space complexity of computations in Section 2.2 and Section 2.3. By considering restrictions on these two computational resources we define several complexity classes in Section 2.4. Basic results regarding these complexity classes are recalled in Section 2.4.1.

In Chapter 3 we take a closer look at the complexity classes using the logarithmic amount of space. First we introduce two types of the composition of Turing machines that preserve logarithmic space complexity. In Section 3.1 and in Section 3.2 we recall some basic facts about log-space reductions. Section 3.3 serves as a overview of classes using logarithmic space. In this section we mention two important computational problems in space complexity theory namely deciding connectivity of a directed and undirected graphs.

Chapter 4 is devoted to the long standing open problem whether $\mathcal{NL} = \mathcal{UL}$. In Section 4.1 we cover a recent result on the directed planar reachability problem belonging to $\mathcal{UL}$. Finally, we mention two possible approaches to this problem in Section 4.2.

Chapter 5 contains a proof of the main result of this thesis, namely that the directed connectivity problem on the class of 3D monotone grid graphs with bounded height belongs to $\mathcal{UL}$. We also mention some further restrictions of the connectivity problem worth investigating.

1.1 Motivation

Non-determinism in the space complexity theory has interesting properties that are in many ways different than in the time complexity setting. It is a common conjecture that $\mathcal{NP} \neq co\text{-}\mathcal{NP}$, but for space complexity classes, Immerman and Szelepcsényi proved that for $S(n)$ being a computable real function lower bounded by $\log_2 n$ it holds that $\mathsf{NSpace}(S(n)) = co\text{-}\mathsf{NSpace}(S(n))$. Another important result that has currently no equivalent in time complexity is Savitch's theorem that states $\mathsf{NSpace}(S(n)) \subseteq \mathsf{DSpace}(S(n)^2)$.

These results bring about a number of questions about other space complexity classes. Our interest was devoted to an open problem about two space complexity classes called $\mathcal{NL}$ and $\mathcal{UL}$. Most of the researchers believe that $\mathcal{NL} = \mathcal{UL}$. There are several results to support this conjecture but a proof is still missing.

In this thesis we did not try to solve this, probably very hard, problem, but rather give an overview of related results and methods that were developed regarding this problem. Particularly interesting is a recently proved result that states that a directed reachability problem remains $\mathcal{NL}$-complete when we restrict to class of 3D monotone grid graphs. Thus to prove the conjecture about $\mathcal{NL} = \mathcal{UL}$ it is enough to show that this problem is in $\mathcal{UL}$.

There are no problems known to be in the class $\mathcal{UL}$ that are complete for $\mathcal{UL}$ under log-space reductions. Furthermore there are not many problems known to be in $\mathcal{UL}$ at all except problems that are known to be in $\mathcal{L}$ because for trivial reasons $\mathcal{L} \subseteq \mathcal{UL}$.

1.2 Definitions and notation

In this section we introduce some basic graph theory terms. Most of them are taken over from a standard graph theory textbook [Die06] or from [Gol08], [BTV07], [Ryj11] and [Ryj09]. The interested reader is referred to these sources for further details.

An undirected graph G is a pair $G = (V, E)$ where V is a finite set and $E \subset \binom{V}{2}$. A directed graph G is a pair $G = (V, E)$ where V is a finite set and $E \subset V \times V$. Elements of $V(G)$ are called the vertices and elements of $E(G)$ are called the edges. For two vertices $x, y \in V(G)$ connected by an edge we will usually denote this edge xy or $\{x, y\}$ for undirected graphs and (x, y) for directed graphs.

We say that a graph H is a subgraph of a graph G if $V(H) \subset V(G)$ and $E(H) \subset E(G)$.

A path in a graph G is a non-empty graph $P = (V, E)$ of the form

$$V = \{x_{0,}x_1, ..., x_k\},\ E = \{x_0x_1, x_1x_2, ..., x_{k-1}x_k\},$$

where the x_i are all distinct. Vertices x_0 and x_k are the end vertices of P. We call $|E(P)|$ the length of P. If $P = x_0...x_{k-1}$ is a path and $k \geq 3$, then the graph $C := P + x_{k-1}x_0$ is called a cycle. A Hamiltonian cycle is a cycle in a graph G that contains all its vertices.

A graph G is connected, if for every two vertices x, y there exist a path from x to y in G. G is disconnected otherwise.

A tree $T = (V, E)$ is a connected graph with $|V(G)| - 1$ edges, or equivalently a connected graph without cycles. A forest is a disjoint union of trees.

A planar graph is a graph that can be embedded in the plane such that none of its edges intersect.

Later in the thesis we will also use an example about graph coloring. A chromatic number of a graph G, denoted by $\chi(G)$, is the minimum number of colors required for a proper coloring of G. For details see [Die06] Chapter 5.

We are mainly interested in the asymptotic behavior of functions considered in the text. For this we use the Bachmann–Landau notation (or Big O notation).

Definition 1.1 ([AB09])**.** If f,g are two functions from $\mathbb{N}$ to $\mathbb{N}$, then we say that

- $f = O(g)$ if there exists a constant c such that $f(n) \leq c \cdot g(n)$ for every sufficiently large n.
- $f = \Omega(g)$ if $g = O(f)$,
- $f = \Theta(g)$ if $f = O(g)$ and $g = O(f)$,
- $f = o(g)$ if for every $\varepsilon > 0$, $f(n) \leq \varepsilon \cdot g(n)$ for every sufficiently large n.

Unless stated otherwise all the logarithms mentioned in the text are binary logarithms, so $\log n = \log_2 n$.

2 Computational Complexity

Computational complexity focuses mainly on two closely related topics. The first of them is the notion of complexity of a “well defined” problem and the second is the ensuing hierarchy of such problems. By “well defined” problem we mean that all the data required for the computation are part of the input. If the input is of such a form we can talk about the complexity of a problem which is a measure of how much resources we need to do the computation.

What do we mean by relationship between problems? An important technique in the computational complexity is a reduction of one problem to another. Such a reduction establish that the first problem is at least as difficult to solve as the second one. Thus, these reductions form a hierarchy of problems. We will cover this technique in Section 2.1.3.

2.1 Introduction

Before we start, we will have to establish all the basic definitions, which are necessary for further chapters.

In this section we will follow the description given in [Gol08], Section 1.2 and [Koz06] Chapters 1, 2, 4, 5 and 6.

2.1.1 Representation

In mathematics one usually works with mathematical objects as with abstract entities. This is not the case of computational complexity. For our purposes we have to define how data will be represented to be able to find a feasible computational model that will handle them.

Definition 2.1. A string is a finite binary sequence. For $n \in \mathbb{N}$, we denote by $\{0,1\}^n$ the set of all strings of length n and call its elements the $n-$ bit strings. The set of all strings is denoted by

$$\{0,1\}^* := \bigcup_{n \in \mathbb{N}} \{0,1\}^n.$$

For $x \in \{0,1\}^*$, we denote the length of x by $|x|$.

Now we are able to define two most important kinds of computational tasks that we deal with. They differ in the type of the solution.

Definition 2.2 (Search problem)**.** Let $R \subseteq \{0,1\}^* \times \{0,1\}^*$ and $R(x) := \{y : (x,y) \in R\}$ denote the set of solutions[1] for the instance x. A function $f : \{0,1\}^* \longrightarrow \{0,1\}^* \cup \{\perp\}$ solves the search problem of R if for every x the following holds: if $R(x) \neq \emptyset$ then $f(x) \in R(x)$ and otherwise $f(x) = \perp$.

It is required, that f never outputs a wrong solution.

A special case of search problems is a class of problems having a unique solution (i.e. $|R(x)| = 1$ for all x). We will be interested in this particular case in Chapter 4 because it is closely related to our main result.

The definition of decision problems follows.

Definition 2.3 (Decision problem)**.** Let $S \subseteq \{0,1\}^*$. A function $g : \{0,1\}^* \longrightarrow \{0,1\}$ solves the decision problem of S if for every x it holds that $g(x) = 1$ if and only if $x \in S$.

From the definition of decision problem it is clear that the function g is actually the characteristic function of set S.

One should further notice, that f solves the search problem of R when the Boolean function $g : \{0,1\}^* \longrightarrow \{0,1\}$ defined by $g(x) = 1 \iff f(x) \neq \perp$ solves the decision problem of $\{x : R(x) \neq \emptyset\}$.

Closely related to a decision problem is a notion of a language.

Definition 2.4. A language is a set (possibly infinite) of strings.

Definition 2.5. Let M be a Turing machine (see Section 2.1.2). The language $L(M)$ is a subset of $\{0,1\}^*$ such that

- for all $x \in L(M)$, M halts in the accept state s.
- for all $x \notin L(M)$, M halts in the reject state r.

It is easy to see that languages and decision problems are essentially interchangeable.

2.1.2 Turing Machine

Before we give a detailed definition it might be useful to take a look at a brief overview. For that we will consider only deterministic Turing machines (TM). Our TM will act as *acceptor*, which means that it will accept, reject or loop on every input. In general TMs may have several read/write work tapes or dedicated input read-only and output write-only tapes but it can be proven that they all have equivalent computational capabilities as one-tape TM.

[1]We can look at $R(x)$ as on a set of strings that are also outputs of function f on input x.

We will define a one-tape deterministic TM as a 9-tuple

$$M = (Q, \Sigma, \Gamma, \vdash, \sqcup, \delta, s, t, r),$$

where

- Q is a finite set of *states*;
- Σ is a finite *input alphabet* (in our case $\Sigma = \{0, 1\}$);
- Γ is a finite *tape alphabet* and $\Sigma \subset \Gamma$;
- $\vdash \in \Gamma - \Sigma$ is the *left end-marker*;
- $\sqcup \in \Gamma - \Sigma$ is the *blank symbol*;
- $\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{-1, 0, 1\}$ is the *transition function*;
- $s \in Q$ is the *start state*;
- $t \in Q$ is the *accept state*;
- $r \in Q$ is the *reject state* ($r \neq t$).

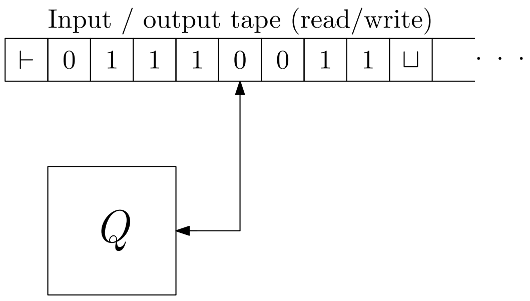

Figure 2.1.1: Schema of a one-tape Turing machine.

The most important of these is the transition function which describes how the TM processes the input. Formally it is defined as follows. For $p, q \in Q$, $a, b \in \Sigma$ and $d \in \{-1, 0, 1\}$ we have $\delta(p, a) = (q, b, d)$. This says that if the state is p and head is scanning symbol a then TM writes b, moves the head in direction d, and enters state q. There are two more criteria we require from the transition function. First the TM should never leave its work tape so for all $p \in Q$ there exist $q \in Q$ such that $\delta(p, \vdash) = (q, \vdash, 1)$. A second requirement is that whenever a TM enters an accept or a reject state it never leaves it so for all $b \in \Gamma$ there exist $c, c' \in \Gamma$ and $d, d' \in \{-1, 0, 1\}$ such that $\delta(t, b) = (t, c, d)$ and $\delta(r, b) = (r, c', d')$. At the beginning of every computation a TM is in state s (a start state). Then it may either enter an infinite loop or end in one of the states t or r that stand for accept and reject, respectively. By entering these two states a TM halts the computation.

We are now going to describe what are deterministic and non-deterministic computational models. The difference between the former and the latter is in the transition function. The description above is of a deterministic TM. For a non-deterministic model the transition function δ is no longer a function but a relation $\Delta \subseteq (Q\backslash\{t, r\} \times \Gamma) \times (Q \times \Gamma \times \{-1, 0, 1\})$. This means that there is not a uniquely determined consecutive state but a finite set of possible next states. These choices create a rooted directed tree structure where root is a start state s (compare with an oriented path structure for a deterministic case). An accepting (resp. rejecting) computation of a non-deterministic TM is a path in the tree structure that starts in the root s and ends in one of the leafs of the tree that corresponds to the state t (resp. r). An input x (a finite string) is accepted by TM if there exists an accepting computation (TM halts when in state t).

As mentioned above a TM M is a 9-tuple consisting of finite objects. This means that M is a finite object and can be encoded into a finite sequence of zeros and ones. It follows that such a finite sequence describing a TM can be used as an input for another TM which leads to the notion of universal TM.

Definition 2.6 (Universal Turing machine)**.** A universal Turing machine is a Turing machine that receives as an input a description of a machine M and an input for M denoted by x_M and returns the value of $M(x_M)$ if M halts on x_M and otherwise does not halt.

Theorem 2.7. *There exists a universal Turing machine.*

This is a fundamental fact underlying the paradigm of general-purpose computers. The universal TM can be viewed as a virtual machine that executes source code representing a specific TM.

2.1.3 Oracle Machine

Another important type of TM is the Oracle Machine (see Figure 2.1.2). The oracle serves as an advice giving tool that is not accounted for time or space usage and the superior TM has control over it.

Oracle machine is a TM defined as in Section 2.1.2 with an additional oracle input tape and oracle output tape. A string written by a TM onto an oracle input tape is called a query for which oracle outputs an answer onto an oracle output tape. This process takes $O(1)$ time and space so the work performed by an oracle is not counted to the time and space complexity of TM.

We now present a brief example of the functionality of an oracle TM. Let M be an oracle deterministic TM working in polynomial time with oracle O that solves problems from $\mathcal{NP}$ (see Section 2.2.1). We are given a graph G and we are supposed to determine the chromatic number $\chi(G)$. This is a well-known optimization problem, which is hard for $\mathcal{NP}$ so we do not expect that there is a deterministic polynomial time TM that solves it. By using an oracle O the situation will change. We can use binary search to find an optimal k and hence $\chi(G)$. Instead of doing all the work, M

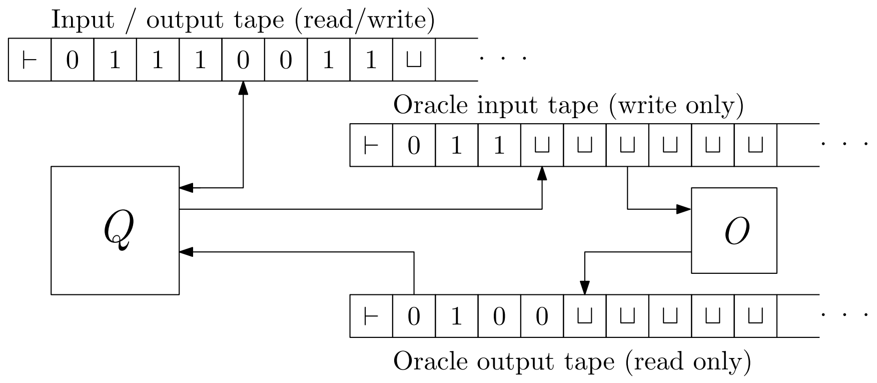
Input / output tape (read/write)
⊢ 0 1 1 1 0 0 1 1 ⊔ . . .
Oracle input tape (write only)
⊢ 0 1 1 ⊔ ⊔ ⊔ ⊔ ⊔ ⊔ . . .
Q
O
⊢ 0 1 0 0 ⊔ ⊔ ⊔ ⊔ ⊔ . . .
Oracle output tape (read only)

Figure 2.1.2: Schema of an oracle one-tape Turing Machine.

will use queries to O asking whether there exists a proper k-coloring for k being the value currently found by binary search. The number of calls of O is upper bounded by $\log|V(G)|$ and because oracle O works in time $O(1)$, our problem now has time complexity $O(\log|V(G)|)$ instead of $O(n!)$ for a brute force approach.

From the foregoing example we see that by using such oracle machines, we are able to separate certain parts (or subroutines) of "efficiently unsolvable" problems and classify their complexity. This approach seems to be very useful when solving a variety of problems in complexity theory. One may imagine this as a kind of reduction that on input x TM M transforms x into x' that is written on oracle query (input) device, oracle answers with y' and M transforms y' into output y which is the solution for input x. In the example above we reduced the problem of finding the chromatic number of a graph to the problem of determining whether there exists a k-coloring of a graph. By a similar technique it can be shown that these tasks are computationally equivalent. By taking an oracle TM M with oracle O that solves a problem of finding the chromatic number of a graph G, it is easy to decide if G has a k-coloring for some $k \geq 2$.

By using oracle machines we can define new complexity classes. Given a class $\mathcal{C}$ of problems accepted by some TM M with access to an oracle O solving problems in a class $\mathcal{D}$ we say that M accepts problems from a class $\mathcal{C}^{\mathcal{D}}$. In the example above, an oracle TM accepted problems from $\mathcal{P}^{\mathcal{NP}}$.

2.1.4 Non-uniform Models

In previous sections we discussed uniform models represented mainly by Turing machines. We saw that a TM is able to handle an input of any (finite) size and either give a corresponding answer or enter an infinite loop. In contrast, non-uniform models depend on the size of an input. Instead of having one TM for solving some problem P we now have a possibly infinite family of models $\mathcal{C} = \cup_{n\in\mathbb{N}} C_n$ for every input size (C_n is a model for handling all inputs of size n). Such classes might not even have a finite description. It is possible that one is required to give an explicit description to every $C_n \in \mathcal{C}$ and since $\mathcal{C}$ is infinite we cannot hope for a finite description of such class.

There are two important non-uniform models of computation: Boolean circuits and "machine that takes advice". For our purposes we will need only Boolean circuits as they play a key role in $\mathcal{P}$-completeness under log-space reductions (Section 3.2).

Definition 2.8 (Boolean circuit)**.** A Boolean circuit is an acyclic directed graph with labeled vertices that are of three kinds[2]: sources, sinks and internal vertices.

- Sources (or input terminals) are vertices with in-degree 0 that are labeled with natural numbers. For input $x = x_1 x_2 ... x_n$, a labeling of input terminals represents an index of input variables. Input terminals do not need to have pairwise distinct labellings.
- Internal vertices (or gates) have in-degree and out-degree at least 1. They are labeled by Boolean operations $\wedge$, $\vee$ and $\neg$, where gates labeled with $\neg$ have in degree exactly 1.
- Sinks (or output terminals) are vertices with in-degree 1 and out-degree 0. Output terminals are uniquely labeled by numbers $1, 2, ..., m$ where m is the number of output terminals. We also say that such a circuit produces output of length m.

Proposition 2.9. *There exist a polynomial-time algorithm that, given a circuit C and a corresponding input x, outputs the value of C on input x.*

To prove Proposition 2.9 consider the following recursive procedure. If there is a non-evaluated gate g with labeling $\wedge$ in circuit C with child vertices $v_1, v_2, ..., v_n$ then value of g is $\wedge_{i=1}^n v_i$. A value of a gate labeled with $\vee$ would be $\vee_{i=1}^n v_i$ and since we allow gates labeled with $\neg$ to have in-degree only 1 such a gate would get a value $\neg v$, where v is its child in circuit C.

For every Boolean function $f : \{0,1\}^* \longrightarrow \{0,1\}$ there exists a family of circuits that compute f . Given an input x of length n, $f(x)$ can be computed by a circuit of size $O(n2^n)$. They are often used to prove lower bounds on complexity of some problems. For further details we refer to [Koz06], Lecture 30.

2.2 Time Complexity

When using Turing machines as a model of computation we are able to measure the number of steps taken by the algorithm on each possible input. Such a function, denote it $t_A : \{0,1\}^* \longrightarrow \mathbb{N}$, is called the time complexity of algorithm A. To make

[2]We do not allow isolated vertices.

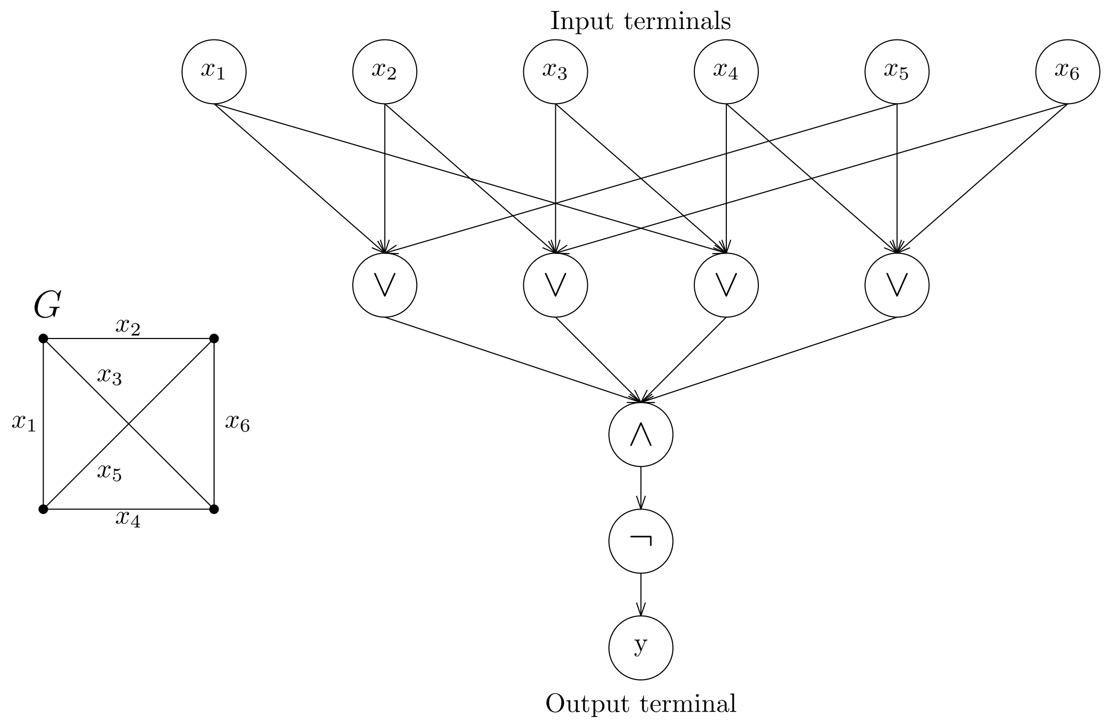

Figure 2.1.3: A Boolean circuit for solving 3-IND (a problem of deciding if a graph contains independent 3 vertices - vertices that are pairwise non-adjacent) on spanning subgraphs G' of the graph G. The variables x_i represent edges of a graph G' ($x_i = 1 \Leftrightarrow x_i \in E(G')$). The first level of gates serves to decide if the edges of G' form a desired subgraph by evaluation of all possibilities. If there are 3 independent vertices then $y = 1$.

this definition reasonable we will focus only on algorithms that halt on every input (i.e. for every input $x \in \{0,1\}^*$, $t_A(x)$ is finite number).

For our purposes we will be mainly interested in a dependence between the size of the input and number of steps of algorithm A. Thus, we will consider $T_A : \mathbb{N} \longrightarrow \mathbb{N}$ defined by $T_A(n) := \max_{x\in\{0,1\}^n} \{t_A(x)\}$. Notice that T_A represents time complexity of worst case input of length $n \in \mathbb{N}$ for algorithm A.

So far we defined the time complexity of an algorithm. The *time complexity of a problem* is time complexity of "the fastest" algorithm that solves it. For two algorithms A and B, the functions T_A and T_B does not have to be comparable so we actually compare $O(T_A)$ and $O(T_B)$. It is obvious that the time complexity of a problem may depend on a model of computation (TM, RAM, ...). The following thesis asserts that the variation is not too significant.

The Cobham-Edmonds Thesis. *A problem has time complexity t in some* "reasonable and general" *model of computation if and only if it has time complexity* poly(t) *in the model of single-tape Turing machines.*

The Cobham-Edmonds Thesis is stronger than the Church-Turing Thesis because it does not only say that class of solvable problems is invariant when one uses a “reasonable and general” model of computation but it also says that the time complexity does not change much (is polynomially related) in either way.

This immediately leads us to the notion of efficient algorithm. We say that an algorithm is efficient when for all inputs of size n it halts after $P(n)$ steps, where $P(n)$ is a polynomial in n. Most of the time we will be interested in an asymptotic behavior of algorithms and not about the exact values of constants. From empirical experience when there is an efficient algorithm for a problem, than there is an efficient algorithm with number of steps bounded by polynomial with a reasonable degree.

2.2.1 $\mathcal{P}$, $\mathcal{NP}$ and completeness

Two well known time complexity classes are $\mathcal{P}$ and $\mathcal{NP}$ where the latter can be defined as a class of decision problems that can be solved in polynomial time by non-deterministic TMs. $\mathcal{P}$ is defined the same way as $\mathcal{NP}$ but one restricts to deterministic TMs only, hence $\mathcal{P} \subseteq \mathcal{NP}$. Whether the opposite inclusion holds is the most famous open question in the complexity theory.

We have already mentioned reductions of problems in Section 2.1.3. We say that a problem P is hard for a class $\mathcal{C}$ if for all problems $Q \in \mathcal{C}$ there exist an oracle machine[3] M_P with oracle solving P such that M_P solves Q. Furthermore we say that P is complete for a class $\mathcal{C}$ if it is hard for $\mathcal{C}$ and $P \in \mathcal{C}$.

The concept of completeness is useful when showing that one class is contained in another. If P is $\mathcal{C}$-complete problem then any problem in $\mathcal{C}$ can be solved using P, thus by showing that $P \in \mathcal{C}'$ implies $\mathcal{C} \subseteq \mathcal{C}'$.

There are several known problems complete for $\mathcal{NP}$ but the one that played a major role in development of complexity theory is the Boolean formula satisfiability (SAT).

Theorem 2.10 ([Coo71]). SAT *is complete for* $\mathcal{NP}$.

It is easy to see that SAT is in $\mathcal{NP}$. What Cook showed was that it is possible to describe polynomially bounded non-deterministic TMs only by using Boolean formulas so any problem that can be solved by such TMs (all problems in $\mathcal{NP}$) are also solvable by using an oracle TM with oracle that solves SAT.

Problems hard for $\mathcal{NP}$ but but not contained in $\mathcal{NP}$ are usually optimization variants of $\mathcal{NP}$-complete problems. A problem worth mentioning is the Travelling salesman problem. It is a problem of finding a Hamiltonian cycle of minimum weight in weighted graph. It can be shown that this problem is $\mathcal{NP}$-hard but not complete because it is not a decision problem. The best known algorithms for this problem

[3]This oracle machine has the same time (resp. space) bounds as problems in a class $\mathcal{C}$.

have at least exponential time complexity in the size of an input so in practice one usually uses approximation or heuristic algorithms. The former algorithms always produce a solution to the problem but it does not have to be the optimal one. The latter algorithms find an optimal solution but only for some instances. The reason why such an approach is used is because both approximation and heuristic algorithms are usually very fast (polynomial in time complexity) and can be used to find a “good” solution even for very large instances.

2.3 Space Complexity

Another measure of efficiency is the use of space (or memory). A natural lower bound for time complexity is a linear function in size of the input (to process each element of the input at least once) but space can be reused during the computation so some of the most interesting space complexity classes are actually those using only a logarithmic amount of work space. Time and space efficiency measure is in conflict so one usually has to sacrifice[4] time when enhancing space complexity and the other way around.

The importance of space complexity is especially in the theoretical realm but we should note that there are several results which show a close connection between both criteria of complexity.

In Section 2.1.2 we defined a TM as a model of computation and used it to measure the time complexity of an algorithm. To measure space complexity we will have to use a slightly more complicated computational model. This is mainly because we want to separate the sizes of input and output data from intermediate storage required by the computation. To do so we will use a 3-tape TM with one input tape which is read only, one output tape which is write only and a work tape which is read/write (see Figure 2.3.1). We define the space complexity of a machine M on input $\{0,1\}^n$, denoted as $s_M(n)$, as a maximum of number of cells on a work tape used during a computation. Similarly the space complexity of an algorithm A will be defined as $S_A(n) := \max_{x\in\{0,1\}^n} \{s_A(n)\}$ and the space complexity of a problem will be again defined as a space complexity of the most space-efficient algorithm that solves it. Notice that as in case of time complexity a function in n which can be even a constant function.

We will further assume, when considering space complexity, that a TM in Figure 2.3.1 never scans the input tape beyond the given input (i.e. there is a special symbol at the end of every input) and it also writes into each output tape cell at most once (this can be assured for a small additive penalty to the space complexity of a TM).

[4]In the sense of substituting one resource for the other.

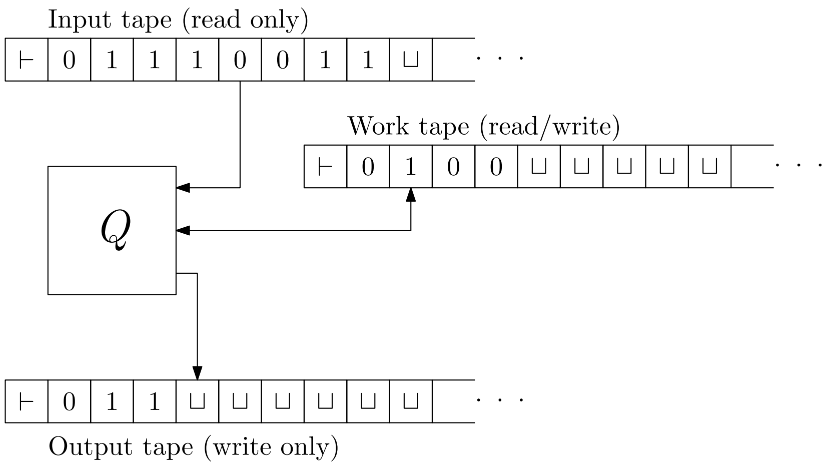

Figure 2.3.1: Schema of a Turing Machine used to measure space complexity.

2.4 Complexity Classes

We have already mentioned complexity classes $\mathcal{P}$ and $\mathcal{NP}$ in Section 2.2.1. In this section we will extend this concept into more general perspective. To do that we introduce complexity classes as a tool for ordering problems due to their actual resource demand. It is obvious that a problem may lie in more than one class.

A complexity class is defined by three main criteria: model of computation (uniform or non-uniform), type of computational problem (decision, search, promise[5], etc.) and resource bound (function of input length). Most of the time we will consider Turing machines as a model of computation and we will work with decision problems. The resource bound criterion will be more interesting.

We recognize four general complexity classes. Denote by $T : \mathbb{N} \longrightarrow \mathbb{N}$ and $S : \mathbb{N} \longrightarrow \mathbb{N}$ two integer functions that we will use as a time and space complexity bound, respectively, and let $L(M)$ be the language accepted by TM M. We can now define

$$\begin{aligned}
\mathsf{DTime}(T(n)) &:= \{L(M) | M \,\text{is a deterministic TM running in time}\, T(n)\}\,, \\
\mathsf{NTime}(T(n)) &:= \{L(M) | M \,\text{is a non-deterministic TM running in time}\, T(n)\}\,, \\
\mathsf{DSpace}(S(n)) &:= \{L(M) | M \,\text{is a deterministic TM running in space}\, S(n)\}\,, \\
\mathsf{NSpace}(S(n)) &:= \{L(M) | M \,\text{is a non-deterministic TM running in space}\, S(n)\}\,.
\end{aligned}$$

For DSpace and NSpace classes, the TM considered is defined as in Section 2.3.

2.4.1 Properties

In this subsection, we discuss the relations between these general complexity classes.

[5]A promise problem is a decision problem where the input is promised to belong to a subset of all possible inputs.

The inclusions $\mathsf{DTime}(T(n)) \subseteq \mathsf{NTime}(T(n))$ and $\mathsf{DSpace}(S(n)) \subseteq \mathsf{NSpace}(S(n))$ are trivial and follow from the fact that a deterministic TM is a special case of a non-deterministic one.

The next property is called linear speedup and it shows that multiplying $T(n)$, resp. $S(n)$ by a constant does not provide any additional computational power.

Theorem 2.11. *Let $T(n) \geq n+1$ and $S(n) \geq \Omega(\log n)$. For any constant $c \geq 1$,*

$$\begin{aligned}
\mathsf{DTime}(cT(n)) &\subseteq \mathsf{DTime}(T(n)),\\
\mathsf{NTime}(cT(n)) &\subseteq \mathsf{NTime}(T(n)),\\
\mathsf{DSpace}(cS(n)) &\subseteq \mathsf{DSpace}(S(n)),\\
\mathsf{NSpace}(cS(n)) &\subseteq \mathsf{NSpace}(S(n)).
\end{aligned}$$

Proof sketch. For proving the last two inclusions, the idea is to expand the working alphabet so that c tape cells can be stored in one tape cell of a new machine and modify the transition function δ. For time bounds we require an extra tape to compress the input. □

The following theorem points out some basic relations between time and space complexity classes.

Theorem 2.12. *Let $T(n) \geq n$ and $S(n) \geq \log n$. Then*

$$\begin{aligned}
\mathsf{DTime}(T(n)) &\subseteq \mathsf{DSpace}(T(n)),\\
\mathsf{NTime}(T(n)) &\subseteq \mathsf{NSpace}(T(n)),\\
\mathsf{DSpace}(S(n)) &\subseteq \mathsf{DTime}(2^{O(S(n))}),\\
\mathsf{NSpace}(S(n)) &\subseteq \mathsf{NTime}(2^{O(S(n))}).
\end{aligned}$$

Proof sketch. The first two inclusions are simple consequences of the fact that in each state a TM can only use one additional work tape cell and hence we get a trivial bound on the space complexity.

The idea behind the two latter inclusions is that there is a finite set of all possible configurations of a TM and each of them can be entered only once (otherwise a TM loops forever). Thus the number of steps is $2^{O(S(n))}$. □

Surprisingly Theorem 2.12 can be strengthened so as to compare a non-deterministic time (resp. space) complexity class with a deterministic space (resp. time) complexity class. It holds that

$$\mathsf{NTime}(T(n)) \subseteq \mathsf{DSpace}(T(n))$$

and

$$\mathsf{NSpace}(S(n)) \subseteq \mathsf{DTime}(2^{O(S(n))});$$

the reasons are similar as in proof of Theorem 2.12 but slightly more technical.

In the following text we will mainly focus on the space complexity classes as they are the main topic of the thesis. One of the most interesting space complexity classes is the class $\mathsf{DSpace}(O(\log n))$, usually denoted as $\mathcal{LOGSPACE}$ (or $\mathcal{L}$). It contains a variety of natural computational problems that we will mention in the following chapter. When considering space complexity, one should also mention class $\mathcal{PSPACE} = \cup_{c\in\mathbb{N}}\mathsf{DSpace}(n^c)$ and its non-deterministic variant is called $\mathcal{NPSPACE}$. From the previously mentioned results follows that $\mathcal{NP} \subseteq \mathcal{PSPACE}$.

In 1970, a breakthrough in the study of space complexity was achieved by Walter Savitch [Sav70].

Theorem 2.13 (Savitch, 1970)**.** *Let $S(n) \geq \log n$. Then*

$$\mathsf{NSpace}(S(n)) \subseteq \mathsf{DSpace}(S(n)^2).$$

Before we sketch the proof we define a problem of deciding the connectivity of a directed graph that we will make use of.

STCONN

Input: oriented graph G and two vertices $s, t \in V(G)$

Question: Is there a path from s to t in G?

Proof sketch. We show the key ideas of the proof for the special case $\mathcal{NL} \subseteq \mathsf{DSpace}(O(\log^2 n))$. Such an inclusion can be proved by solving STCONN in $\mathsf{DSpace}(O(\log^2 n))$. To do that we will use a recursive procedure that has at most logarithmic recursion depth and remembers $O(\log n)$ bits on each level of recursion hence we get desired $O(\log^2 n)$ space complexity.

For a directed graph G and two vertices u and v define a function $\phi_G(u, v, \ell) := 1$ if there exists a path of length ℓ from u to v in G, and $\phi_G(u, v, \ell) := 0$ otherwise. It is obvious that recursion step

$$\phi_G(u, v, 2\ell) = \bigvee_{w\in V(G)} (\phi_G(u, w, \ell) \wedge \phi_G(w, v, \ell))$$

computes ϕ_G. In each recursion level, ℓ is divided by two and thus there are up to $\log \ell$ levels where at the base level, $\phi_G(u', v', 1)$ can be evaluated in log-space just by searching all neighbors of u'. To solve STCONN in $\mathsf{DSpace}(S(n)^2)$ one can use $\phi_G(u, v, |V(G)|)$ and the theorem follows. $\square$

From Savitch's result it immediately follows that $\mathcal{PSPACE} = \mathcal{NPSPACE}$. Proving the similar equivalence for time complexity classes seems to be significantly more difficult. The Clay Mathematics Institute actually listed the $\mathcal{P}$ versus $\mathcal{NP}$ problem as one of the Millennium Prize Problems [Cla12].

Before we continue with the next result we define classes of complement problems.

Definition 2.14. Let $\mathcal{C}$ be a class of decision problems. The complement class denoted by $co\text{-}\mathcal{C}$ is a class of decision problems such that $S \in \mathcal{C}$ if and only if $\{0,1\}^* \backslash S \in co\text{-}\mathcal{C}$.

Another important and more recent result was proven independently by Immerman and Szelepcsényi in 1987 [Imm88, Sze88]. It states the following.

Theorem 2.15 (Immerman, Szelepcsényi, 1987)**.** *For $S(n) \geq \log n$, $\mathsf{NSpace}(S(n)) = co\text{-}\mathsf{NSpace}(S(n))$.*

Immerman and Szelepcsényi actually proved that $\mathcal{NL} = co\text{-}\mathcal{NL}$ but by using a "padding argument"[6] the result can be extended to any other space complexity non-deterministic class above $\mathcal{NL}$.

[6]Padding is a technique for showing that if some complexity classes are equal, then some other classes possessing more computational resource are also equal by extending accepting language with new symbols. For details see [AB09].

3 Logarithmic Space

Complexity classes using only logarithmic work space to process an input became a subject of interest because the space of size $\log n$ is just enough to maintain a counter that may store a number from 0 to n and hence it is possible to remember position of head on input and output tape. We actually allow usage of $O(\log n)$ space which means that the computational model may have a constant number of counters counting from 0 to a polynomial in n. This is enough to solve a variety of natural problems such as adding and multiplying natural numbers. It also gives rise to a widely used reduction called a log-space reduction that is helpful when working with classes below $\mathcal{P}$.

Are there any problems that might be solved using sub-logarithmic amount of work space? A simple problem that requires only constant space is deciding if an integer in binary encoding is even or odd. It is quite surprising that there are problems that require sub-logarithmic space but will not do with constant space [LR97]. The hierarchy of classes is as follows:

$$\mathsf{DSpace}(O(1)) \subsetneq \mathsf{DSpace}(O(\log\log n)) \subsetneq \mathcal{L} \subseteq \mathcal{P}.$$

The last inclusion follows from the fact that from Theorem 2.12 it holds that

$$\mathsf{DSpace}(S(n)) \subseteq \mathsf{DTime}(2^{O(n)})$$

and thus we have

$$\mathcal{L} = \mathsf{DSpace}(O(\log n)) \subseteq \mathsf{DTime}(2^{O(\log n)}) = \mathsf{DTime}(n^{O(1)}) = \mathcal{P}.$$

A similar relation holds also for the non-deterministic case (i.e., $\mathcal{NL} \subseteq \mathcal{NP}$). The question whether $\mathcal{L}$ is a proper subset of $\mathcal{P}$ or the equality holds is one of the most important open questions in complexity theory[1].

In the first two sections of this chapter we adopted the description from [Gol08], Chapter 5. For the following section we used information presented in [AKG12].

3.1 Composition Lemmas

There are two important composition lemmas that show how to compose Turing machines preserving the space bound restriction. For now assume we compose only

[1]The same question for non-deterministic classes (i.e., $\mathcal{NL} \subsetneq \mathcal{NP}$) is also unresolved.

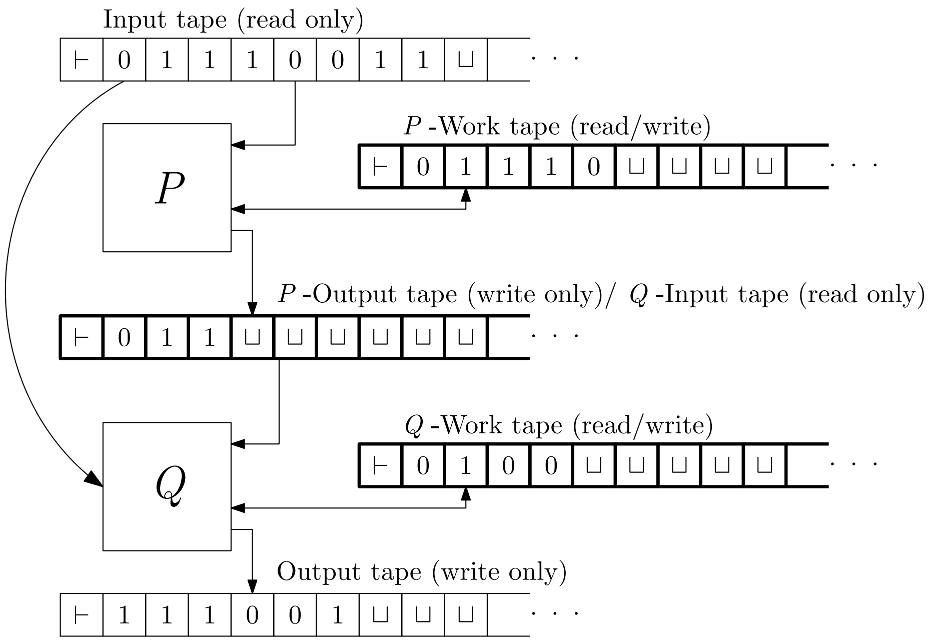

Figure 3.1.1: Naive composition. Tapes shown in bold are relevant for space complexity.

two TMs where the second one uses the original input together with the output of the first TM as its input. An easy case is when output of first TM has length at most $\log n$, because we can emulate its output tape by additional work tape and this will still fulfill the space bound requirement. We call such composition a naive composition.

Lemma 3.1 (Naive composition)**.** *Let $f_1 : \{0,1\}^* \longrightarrow \{0,1\}^*$ and $f_2 : \{0,1\}^* \times \{0,1\}^* \longrightarrow \{0,1\}^*$be computable in space s_1 and s_2, respectively. Then the function f defined by $f(x) := f_2(x, f_1(x))$ is computable in space s such that*

$$s(n) = s_1(n) + s_2(n + l(n)) + l(n), \tag{3.1.1}$$

where $l(n) = \max_{x\in\{0,1\}^n} \{|f_1(x)|\}$.

Lemma 3.1 defines composition where no space optimization is used. The first term in (3.1.1) stands for the space required by the first TM that on input of size n uses $s_1(n)$ work space and produces output of size at most $l(n)$. The second TM has access to the original input as well as to the output produced by the first TM (which is also included into the space complexity of the composition) so the whole input for f_2 has size $n + l(n)$, thus the overall space required to compute f_2 is $s_2(n + l(n))$. One can achieve a simple improvement by reusing the work space of f_1. The space complexity of such a composition would be $s(n) = \max(s_1(n), s_2(n + l(n))) + l(n)$. For our purposes such an improvement would only decrease a constant in (3.1.1) and so we do not distinguish between these two approaches. In Figure 3.1.1 $s_1(n)$

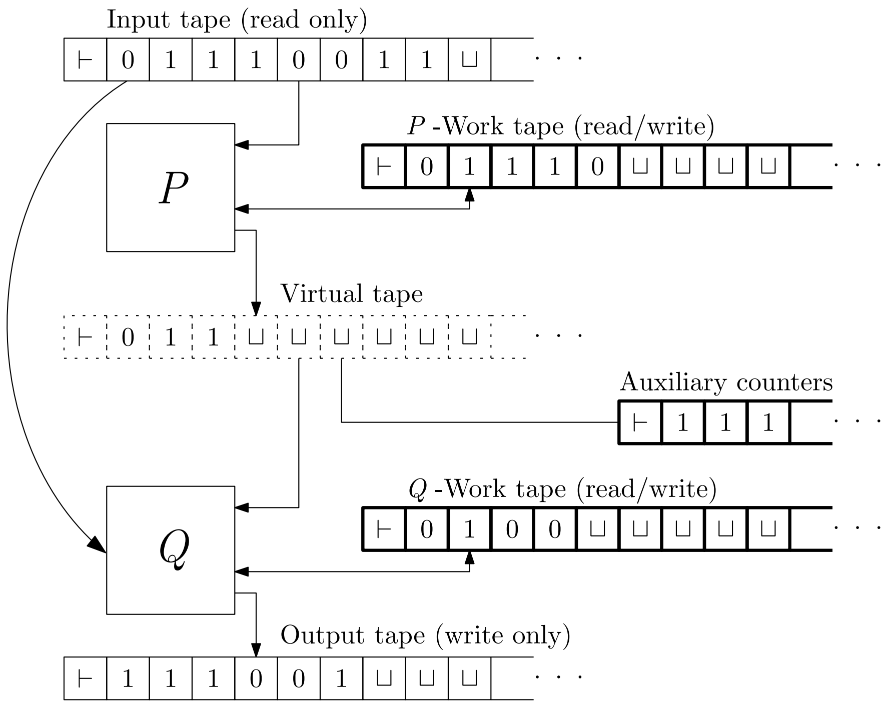

Figure 3.1.2: Emulative composition. Tapes shown in bold are considered when determining the space complexity of the composition. The virtual tape does not store any data, thus it does not occur in (3.1.2).

is the number of cells used on the P-work tape, $s_2(n)$ represents the number of cells used on the Q-work tape and $l(n)$ is the size of the output on the P-output tape.

The second composition lemma is of major importance for us because it is one of the main tools for designing space bounded algorithms.

Lemma 3.2 (Emulative composition)**.** *Let f_1, f_2, s_1, s_2, l and f be as in Lemma 3.1. Then f is computable in space s such that*

$$s(n) = s_1(n) + s_2(n + l(n)) + O(\log(n + l(n))). \tag{3.1.2}$$

As an illustration of emulative composition consider a simple example. Let $f(e)$ be a function that assigns weights to the edges of a graph G that is acyclic with bounded degree and let s, t be vertices of G. Our task is to find a path of minimum weight between s and t. We obviously cannot remember values of function f for each edge of G because that would require space at least linear in the size of the input. So whenever an algorithm for finding a minimal path needs the weight of an edge e, the function $f(e)$ has to be invoked to provide such information. The function f is computed by P in Figure 3.1.2 and the weights of the edges are exactly the information “stored” on the virtual device that together with the original input forms the input for Q.

Lemma 3.2 is described by Figure 3.1.2. The terms $s_1(n)$ and $s_2(n)$ in (3.1.2) have exactly the same meaning as in Lemma 3.1. The last term is more interesting. It is the size of a constant number of auxiliary counters that store positions on the virtual tape. Each of the counters store a number from 0 to $n+l(n)$ so in the binary encoding one requires $\log(n+l(n))$ bits to store such a number and that is where the last term in (3.1.2) comes from. If Q requests a bit in a cell at position $\leq n$, it reads it from original input; otherwise, it asks P to recompute the requested bit. Hence we do not have to remember the output of P but every time Q asks for a part of its output, it is recomputed and we only need an auxiliary counter (of logarithmic size) to remember a position on the output tape of P. Such a virtual tape does not contain any data on its own so it is not included in the space complexity of the composition.

The main difference when applying such an approach is that we first call Q that itself makes calls to P whenever it requires additional information other than that in original input. It is an essence of most algorithms using logarithmic space because it removes the need to remember anything that can be computed from the original input. Even though it provides a very useful tool for log-space computations it has its limits. It is possible to do only a constant number of such compositions. For every emulative composition one has to add a constant number of counters and by performing, say, $\log n$ emulative compositions we end up with space complexity $\log^2 n$.

3.2 Log-space Reductions

Reductions of one problem to another are a useful tool for establishing membership in a complexity class. A widely used reduction is called many-one log-space reduction. It is defined as a reduction of a decision problem to another decision problem solved by an oracle where the oracle call can be used only once at the end of the computation and the result cannot be further modified.

We now define a problem of Boolean circuit evaluation and then we mention a theorem that shows the importance of circuits in computational complexity.

CEVL

Input: Boolean circuit C_n for n input variables and a string α of length n.

Output: The value of C_n on input α.

Theorem 3.3. CEVL *is in $\mathcal{P}$ and every problem in $\mathcal{P}$ is log-space many-one reducible to* CEVL*.*

In other words Theorem 3.3 states that CEVL is $\mathcal{P}$-complete under log-space many-one reductions. This also implies that $\mathcal{L} \neq \mathcal{P}$ if and only if CEVL $\notin \mathcal{L}$. It has a close connection to parallel computations because circuits are used as their model and unless $\mathcal{P} = \mathcal{L}$ no $\mathcal{P}$-complete problem can be solved by the parallel computation [Rei85].

3.3 Classes Using Logarithmic Space

3.3.1 $\mathcal{L}$ (and Undirected connectivity)

In Section 2.4.1 we already defined the complexity class $\mathsf{DSpace}(O(\log n))$, denoted as $\mathcal{LOGSPACE}$ or $\mathcal{L}$. It is the class of decision problems accepted by a deterministic logarithmic space-bounded Turing machine with one read-only input tape, a read-write work tape whose size is bounded by $O(\log n)$, and write only output tape. Since $\mathcal{L}$ is a class of decision problems, an output tape will contain only 0 or 1 when a TM terminates.

There are several natural problems in $\mathcal{L}$. We will first introduce some simple ones.

PATH

Input: Directed graph G.

Question: Is G a directed path?

Lemma 3.4. *PATH $\in \mathcal{L}$.*

Proof. G is a directed path if it is connected, acyclic and has either a single vertex or all but the start and end vertices have in and out degree 1. Denote start vertex as s and end vertex as t. Clearly s has in-degree 0 and t has out degree 0. If $|V(G)| = 1$ then G is a path so let $|V(G)| \geq 2$. We start by finding a vertex s with in-degree 0. If such vertex does not exist then return 0. We set a counter $i := 1$, set $v := s$ and find a neighbor v' of v in G. By repeating this step while incrementing i we either reach t (vertex with out-degree 0) in $|V(G)|$ steps and return 1 or we stop when $i = |V(G)|$ and $v \neq t$ or $v = t$ and $i < |V(G)|$ and return 0. □

CYCLE

Input: Directed graph G.

Question: Is G a directed cycle?

Lemma 3.5. *CYCLE $\in \mathcal{L}$.*

Proof. We will use Lemma 3.4. Choose an arbitrary edge $e = (u, v) \in G$. Set $G' := G \backslash \{e\}$ and solve $\mathsf{PATH}(G')$. If the result is 1 and the vertex with in-degree 0 in G' is v then return 1, else return 0. □

TREE_REACH

Input: Directed tree T and vertices $s, t \in V(T)$.

Question: Is there a directed path from s to t in T?

Lemma 3.6. *TREE_REACH $\in \mathcal{L}$.*

Proof. Denote the root of the tree T by r. Notice that if a directed path from s to t exists s has to be on a directed path from r to t and such path is uniquely determined. It is enough to invert all edges and check if s lies on directed path from t to the r. □

More problems in $\mathcal{L}$ can be found in [CM87].

The problem that will concern us now is about deciding connectivity of undirected graph.

USTCONN

Input: Undirected graph G and vertices $s, t \in V(G)$.

Question: Is there a path from s to t in G?

The most important result in space complexity theory in the past decades is a recent result of Reingold [Rei08] that states that USTCONN $\in \mathcal{L}$.

Theorem 3.7 (Reingold, 2005)**.** *USTCONN $\in \mathcal{L}$.*

In the rest of this section we will show the main idea of the proof.

A key observation is that solving USTCONN in $\mathcal{L}$ is easy if the input graph has constant degree and logarithmic diameter. Let G be a d-regular graph with logarithmic diameter. We can label the edges at any vertex by numbers from $[d]$. For any pair of vertices u, v in one component of G there is a path P from u to v of length $\leq \log |V(G)|$. We can represent P by a sequence s such that for $\phi : P \longrightarrow [d]^{|V(G)|}$, $s = \phi(P)$. Elements of s are the indices of edges used at the visited vertices. The space needed to store such a sequence is d per coordinate times number of coordinates, thus $d \log |V(G)| = O(\log n)$.

First we define expander graphs (for details see [HLWO06]) because such graphs fulfill the previously mentioned requirements and we will use them in the construction later.

Definition 3.8 (Combinatorial definition)**.** A graph $G = (V, E)$ is c-expanding if, for every set $S \subset V(G)$ of cardinality at most $|V(G)|/2$, it holds that

$$\Gamma_G(S) := \{v : \exists u \in S \, \text{s.t.} \, \{u, v\} \in E(G)\}$$

has cardinality at least $(1 + c) \cdot |S|$.

Next we define an expansion parameter as a measure of quality of expansion of G.

Definition 3.9. Let G be a graph. The edge boundary of a set $S \subset V(G)$, denoted ∂S, is $\partial S = E(S, \bar{S})$, where $E(S, \bar{S})$ is the set of outgoing edges from S. The expansion parameter of G, denoted $\bar{\lambda}(G)$, is defined as:

$$\bar{\lambda}(G) = \min_{\{S:|S| \leq \frac{n}{2}\}} \frac{|\partial S|}{|S|}.$$

Reingold showed a method for transforming a graph into a collection of constant degree expanders (with logarithmic diameter) in log-space so by using Lemma 3.2 it is possible to apply the previously mentioned idea. The actual implementation of the method is highly non-trivial so we will only mention some key ideas.

To proceed we have to define a Zig-Zag product which is a product of two graphs defined by Reingold et al. [RVW00]. For simplicity, we assume that the edges in our d-regular graphs are actually partitioned to d perfect matchings (or color classes). For a color $i \in [d]$ and a vertex v let $v[i]$ be the neighbor of v along the edge colored i. Since it is easy into turn a graph G into a 3-regular graph and by adding $d-3$ self loops into d-regular graph, the assumption that the input graph G is d-regular is justifiable.

Definition 3.10 (Zig-Zag product)**.** Let G_1 be a d_1-regular graph on $[n_1]$ and G_2 be a d_2-regular graph on $[d_1]$. Then the Zig-Zag product of G_1 and G_2 denoted by $G_1 ⓩ G_2$ is a d_2^2-regular graph on $[n_1] \times [d_1]$ defined as follows: For all $v \in [n_1]$, $k \in [d_1]$, $i, j \in [d_2]$, the edge (i, j) connects the vertex (v, k) to the vertex $(v[k[i]], (k[i])[j])$.

The vertex $(v[k[i]], k[i][j])$ in the resulting graph can be described as: vertex $k[i]$ is the neighbor of the vertex k along the edge i, thus $v[k[i]]$ is the neighbor of v along the edge $k[i]$. Similarly, $(k[i])[j]$ is the neighbor of $k[i]$ along the edge j.

Now to build a good expander we begin with an input graph G and turn it into a d^2-regular graph G_0 by adding self-loops. Then we find (by exhaustive search) a fixed d-regular graph H with good expansion properties and with d^{2c} vertices where c and d are positive integers. In $t = O(\log |V(G_1)|)$ iterations we build the desired expander letting $G_{i+1} = G_i^c ⓩ H$ for $i = 1, ..., t-1$ where G_i^c is the c-th power of the graph G_i.

Reingold et al. [RVW00] proved that if a suitable graph H is chosen (with $\bar{\lambda}(H) < 1/2$) then in the logarithmic number of iterations we obtain an expander graph because during every iteration the expansion parameter improves, the degree of the graph is preserved (by the definition of Zig-Zag product) and the size increases only by a constant factor.

The question is whether we can perform such transformation in logarithmic space. By using Lemma 3.2 it is possible to show that G_{i+1} can be obtained in log-space when given G_i. The problem is that we actually want to perform a logarithmic number of such transformations. The key to this problem is, as Reingold showed in [Rei08], that to obtain G_{i+1} when given G_i one requires logarithmic space but it can be reused in every iteration. Hence after the logarithmic number of these transformations we end up with the space complexity being $O(\log n)$.

3.3.2 $\mathcal{NL}$ (and Directed connectivity)

$\mathcal{NL}$ is a class of problems accepted by non-deterministic logarithmic space bounded TMs. Non-determinism in space complexity has several features. First, the class $\mathcal{NL}$ is closed under complement as states Theorem 2.15. It is widely believed that no such property holds for time complexity classes (especially $\mathcal{NP}$). Second, it is known that the whole $\mathcal{NL}$ is contained in $\mathsf{DSpace}(O(\log^2(n)))$ by Theorem 2.13. For our further considerations we will be mainly interested in the following open problem.

Conjecture 3.11. *$\mathcal{NL} = \mathcal{UL}$.*

A fundamental problem contained in $\mathcal{NL}$ is the connectivity problem in general directed graphs denoted STCONN (see Section 2.4.1 for definition).

Theorem 3.12. STCONN *is complete for $\mathcal{NL}$ under many-to-one log-space reductions.*

Proof sketch. A computation of a non-deterministic TM can be expressed as a directed graph G with vertices being configurations (states of TM together with work tape content) of TM and edges representing possible transitions between configurations. It is obvious that G is a directed tree with a unique root $r \in V(G)$ that stands for the initial state of the TM. If on input x the TM halts in state t then there is a path in G from r to a leaf of G representing the state t and vice versa. Using emulative composition, G can be created in log-space and the adequate STCONN problem solved on it. $\square$

With this result in hand, to show that $\mathcal{NL}$ is contained in another complexity class, it is enough to show that STCONN can be solved in it. This is obviously true for $\mathcal{P}$ using a backtracking algorithm, hence $\mathcal{NL} \subseteq \mathcal{P}$.

3.3.3 $\mathcal{UL}$

The unambiguous logarithmic space complexity class (denoted $\mathcal{UL}$) is a class of decision problems solvable by a non-deterministic TM M such that

1. if an input $x \in L(M)$, exactly one computation path accepts,
2. if an input $x \notin L(M)$, all computation paths reject.

Unambiguity is a natural restriction of non-deterministic power. It was first introduces in the context of time complexity by Valiant in 1976 [Val76]. The logarithmic space version was introduced in 1991 by Buntrock et al. [BJLR91] and two years later by Àlvarez and Jenner [AJ93].

By the definition, $\mathcal{UL}$ is a restricted version of $\mathcal{NL}$ so a trivial inclusion is $\mathcal{UL} \subseteq \mathcal{NL}$. It is widely believed that whole non-determinism in logarithmic space is captured in $\mathcal{UL}$ and hence $\mathcal{UL} = \mathcal{NL}$. An important result to support the conjecture that $\mathcal{NL} = \mathcal{UL}$ was proved in 1998 by Reinhardt and Allender [RA98].

Definition 3.13. *$\mathcal{UL}$/poly is a class of decision problems solvable by a family of polynomial-size Boolean circuits. The family can be nonuniform.*

Theorem 3.14 (Reinhardt, Allender, 1998)**.** *$\mathcal{NL} \subseteq \mathcal{UL}$/poly.*

Making this relation uniform would prove the conjecture. We will study this problem more closely in Chapter 4.

Another important inclusion is $\mathcal{L} \subseteq \mathcal{UL}$. The reason why it holds is that a deterministic TM has exactly one computational path that either ends state t or r, so the definition of $\mathcal{UL}$ is fulfilled. Hence $\mathcal{UL}$ contains all problems that are in $\mathcal{L}$. The question is whether there are problems that are in $\mathcal{UL}$ but not in $\mathcal{L}$. Some recent results related to this question may be found in [BTV07, LMN09, TW10].

3.3.4 $\mathcal{SL}$

This complexity class denoted as $\mathcal{SL}$ was defined by Lewis and Papadimitriou in [LP82] as a class of problems accepted by a non-deterministic Turing machine M running in logarithmic space such that

1. if an input $x \in L(M)$, one or more computation paths accept,
2. if an input $x \notin L(M)$, all paths reject,
3. if M can make a non-deterministic transition from configuration A to configuration B, than it can also make a non-deterministic transition from B to A.

They showed that $\mathcal{L} \subseteq \mathcal{SL} \subseteq \mathcal{NL}$. A reason for the interest in this class is that it captures the complexity of USTCONN (see Section 3.3.1). Many graph theory problems can be solved by using reduction to USTCONN, which yields a rich set of problems in $\mathcal{SL}$. Some of these may be found in [AG00].

Nisan and Ta-Shma showed in [NTS95] that $\mathcal{SL}$ = co-$\mathcal{SL}$ by reducing USTCONN to its complement.

Theorem 3.15 (Nisan, Ta-Shma, 1995)**.** *$\mathcal{SL}$ = co-$\mathcal{SL}$.*

A major breakthrough was made by Reingold [Rei08] who proved Theorem 3.7 implying $\mathcal{SL} = \mathcal{L}$.

3.3.5 $\mathcal{FL}$

Function log-space ($\mathcal{FL}$) can be defined as the set of functions $f : \Sigma^* \longrightarrow \Sigma^*$ such that there exist a deterministic logarithmic space bounded TM that for every input x outputs $f(x)$. The difference between $\mathcal{L}$ and $\mathcal{FL}$ is that $\mathcal{L}$ contains only decision problems while $\mathcal{FL}$ contains also search problems (see Definition 2.2). The word "Function" is slightly misleading because search problems may have a set of correct outputs for one input.

Many subroutines used for solving problems in $\mathcal{L}$ are actually $\mathcal{FL}$ problems, but since both classes are restricted to logarithmic space, such a composition is possible.

Important problems contained in $\mathcal{FL}$ are ADD, SUM and MULTIPLY.

ADD

Input: Two integers a, b in binary encoding.

Output: Sum of a and b.

SUM

Input: n integers $a_1, a_2, ..., a_n$ in binary encoding.

Output: $\sum_{i=1}^{n} a_i$.

MULTIPLY

Input: Two integers a, b in binary encoding.

Output: Product of a and b.

We cannot perform multiplying directly but rather in an elementary school manner. For example MULTIPLY$(10111, 101)$ is equivalent to ADD$(10111, 1011100)$.

3.3.6 $\mathcal{RL}$

$\mathcal{RL}$ (Randomized log-space) is a class of problems accepted by a probabilistic TM with one-sided error. A probabilistic TM is a deterministic TM with additional tape called the random tape that contains only random bits and the TM can read them in any step of the computation. A probabilistic TM with one-sided error is not allowed to accepts an input incorrectly but it may reject it incorrectly with a probability less than 1/3. Because of the probability setting we usually add a polynomial time restriction. Otherwise it can be shown that $\mathcal{RL}$ without time a bound is as powerful as $\mathcal{NL}$[2]. A relation between other logarithmic space classes is $\mathcal{L} \subseteq \mathcal{RL} \subseteq \mathcal{NL}$ but there is a common conjecture that $\mathcal{RL} = \mathcal{L}$ and Reingold et al. [RTV05] provided the strong evidence to support it.

[2]We refer to [Gol08] Section 6.1.5 for further details.

4 NL vs. UL

We have already given a brief introduction to $\mathcal{NL}$ and $\mathcal{UL}$ in Section 3.3. Here we will study them in more detail.

4.1 Problems in $\mathcal{UL}$

As we have already mentioned in Section 3.3.3 that all the problems in $\mathcal{L}$ are trivially contained in $\mathcal{UL}$. Moreover from Theorems 3.7 and 3.15 it follows that $\mathcal{L} = co\text{-}\mathcal{L}$, so the inclusion is actually $\mathcal{L} \subseteq \mathcal{UL} \cap co\text{-}\mathcal{UL}$. It is not known if $\mathcal{UL} = co\text{-}\mathcal{UL}$, but if $\mathcal{NL} = \mathcal{UL}$ is true then the equality follows from Theorem 2.15.

Other problems that are known to be in $\mathcal{UL}$ are those concerning STCONN. We will be particularly interested in the result of Burke et al. [BTV07] about planar reachability because it is closely related to our result. We will start by defining grid graphs.

Definition 4.1 (Grid graph)**.** A $n \times n$ grid graph is a directed graph whose vertices are pairs of numbers from $[n] \times [n] = \{1, 2, ..., n\} \times \{1, 2, ..., n\}$, and if $e = ((i_1, j_1), (i_2, j_2))$ is an edge then $|i_1 - i_2| + |j_1 - j_2| = 1$.

Pairs of numbers assigned to every vertex of a grid graph can be viewed as a coordinates and hence they represent a natural planar embedding of a graph. The condition on the edges in Definition 4.1 allows existence of only four kinds of edges. We say that for a vertex (i, j)

- the edge $((i, j), (i, j + 1))$ is a north edge,
- the edge $((i, j), (i, j - 1))$ is a south edge,
- the edge $((i, j), (i + 1, j))$ is an east edge and
- the edge $((i, j), (i - 1, j))$ is a west edge.

We will be concerned with some planar properties of grid graphs. We recall a result of Allender and Mahajan [AM04] that puts planarity testing in $\mathcal{SL}$.

Theorem 4.2 (Allender, Mahajan, 2004)**.** *Problem of deciding if a graph G is planar is in $\mathcal{SL}$.*

Since $\mathcal{SL} = \mathcal{L}$, there is a log-space algorithm for testing whether a given graph G is planar, and if so, it outputs a planar embedding of G.

Before we mention another important result of Allender et al. [ADR05] we will define two versions of STCONN for special classes of graphs, namely planar and grid graphs.

PlanarReach

Input: Directed planar graph G and two vertices $s, t \in V(G)$.

Question: Is there a directed path from s to t in G?

Similarly define GGR as PlanarReach restricted to the class of grid graphs.

Theorem 4.3 ([ADR05]). *The* PlanarReach *problem is log-space many-one reducible to* GGR.

Before we continue with proving that PlanarReach is in $\mathcal{UL} \cap co\text{-}\mathcal{UL}$ we introduce a technique from [RA98] for showing membership in $\mathcal{UL} \cap co\text{-}\mathcal{UL}$.

4.2 Possible Approaches to $\mathcal{NL} = \mathcal{UL}$ Problem

We saw in Section 3.3.2 that the directed connectivity problem is complete for $\mathcal{NL}$. To prove equality between $\mathcal{NL}$ and $\mathcal{UL}$ it is reasonable to make use of this property of STCONN. In general digraphs there might be several possible paths between two vertices (not necessarily disjoint) and hence there could be more than one accepting computation path of a corresponding TM. To deal with this issue Reinhardt and Allender [RA98] introduced a general technique for showing membership in $\mathcal{UL}$ by transforming a graph into a min-unique graph.

Definition 4.4 (Min-unique graph). A min-unique graph is a directed graph with positive weights associated with each edge such that for every two vertices u, v there is a unique path of minimal weight (if one exist) from u to v.

The following theorem of Reinhardt and Allender [RA98] shows it is possible to find a minimum weight path in min-unique graphs in log-space.

Theorem 4.5 (Reinhardt and Allender, 1998). *Let $\mathcal{G}$ be a class of graphs and let $H = (V, E) \in \mathcal{G}$. If there is a polynomially bounded log-space computable function f that on input H outputs a weighted graph $f(H)$ so that*

1. *$f(H)$ is min-unique and*
2. *H has a path from s to t if and only if $f(H)$ has a path from s to t*

then the STCONN *restricted to $\mathcal{G}$ is in $\mathcal{UL} \cap co\text{-}\mathcal{UL}$.*

In the statement of Theorem 4.5, an arbitrary class of graphs was used. We will now discuss two naturally defined classes studied in [BTV07].

4.2.1 Grid Graphs

By using Theorems 4.3 and 4.5 we will present a proof given in [BTV07] that $\mathsf{PlanarReach} \in \mathcal{UL} \cap co\text{-}\mathcal{UL}$.

Theorem 4.6 (Bourke et al., 2007)**.** $\mathsf{PlanarReach} \in \mathcal{UL} \cap co\text{-}\mathcal{UL}$.

Proof. For a grid graph G of size $n \times n$ we define a weight function $w : E(G) \longrightarrow \mathbb{N}$ as follows:

$$w(e) = \begin{cases} n^4 & \text{if}\, e \,\text{is an east or west edge,} \\ i + n^4 & \text{if}\, e \,\text{is a north edge in column}\, i, \\ -i + n^4 & \text{if}\, e \,\text{is a south edge in column}\, i. \end{cases} \tag{4.2.1}$$

We will now prove that the weight function w turns G into min-unique graph and hence from Theorem 4.5 $\mathsf{GGR} \in \mathcal{UL} \cap co\text{-}\mathcal{UL}$. Let P be a path in G from s to t and denote $w(P)$ the sum of the weights of the edges of the path P. By $a(P)$ denote the sum of column indexes of north and south (by the definition of w south edges have negative values) edges and and by $b(P)$ the total length of P. Thus, the weight of P is of the form $w(P) = a + bn^4$. It is clear that both $|a(P)|$ and $b(P)$ are less than n^3 (there are fewer than n^2 edges in P) and so for two paths P_1 and P_2 such that $w(P_1) = w(P_2)$ we have $a(P_1) + b(P_1)n^4 = a(P_2) + b(P_2)n^4$, thus $a(P_1) = a(P_2)$ and $b(P_1) = b(P_2)$.

The "a" component of the weight function w can be used to count the number of unit squares enclosed by a cycle C in G. Denote this number by $A(C)$. A cycle C can be viewed as a path that starts and ends at the same vertex so by $a(C)$ we mean the "a" component of such a path.

Claim 4.7. Let C be a simple cycle in G. Then $a(C) = +A(C)$ if C is a counter-clockwise cycle and $a(C) = -A(C)$ if C is a clockwise cycle.

Proof. Without loss of generality assume that C is a counter-clockwise simple cycle in G (for a clockwise cycle we have $a(C) = -a(-C)$ where $-C$ denote counter-clockwise cycle obtained by reversing the edges of C). We want to show that for any two rows j and $j + 1$ of G

- the south and north edges between them alternate when ordered by their column index,
- the westmost edge being south edge,
- the eastmost edge being north edge.

Denote by $(S_j, <)$ the ordered set of north and south edges of C between the j-th and the $(j + 1)$-th row where the set of edges is ordered by the column index. For contradiction suppose that there exist two consecutive edges $e_1 = (u_1, v_1)$, $e_2 = (u_2, v_2)$ from S_j with the same direction. A simple path from v_1 to u_2 has to use an edge $e' \in S_j$ such that $e' > e_2$ or $e' < e_1$. By symmetry this holds for a simple path from v_2 to u_1 so there is an edge $e'' \in S_j$ such that $e'' < e_1$ or $e'' > e_2$. In both cases such paths have to cross each other which implies that C is not a simple cycle. That is a contradiction with the assumption. Since C is a counter-clockwise cycle, the westmost edge is a south edge and the eastmost edges is a north edge.

Now for all $j \in \{0, 1, ..., n-1\}$, if $S_j \neq \emptyset$, the westmost edge is a south edge with weight $-i_1 + n^4$ followed by a north edge with weight $i_2 + n^4$ where $i_1 < i_2$. From the definition of the weight function it follows that $i_2 - i_1$ equals to the number of columns between those two edges. This holds for all consecutive pairs of south-north edges (unit squares in the interior of C are squares between the south edges e_k and the north edges e_l, where $k < l$ for every such a pair) in S_j so we can sum all the unit squares inside the cycle C in row $(j, j+1)$. By summing over all $j \in \{0, 1, ..., n-1\}$ we obtain the area of C. For a clockwise cycle the area of C will be $-A(C)$. □

We will use this property to prove the following claim and the theorem.

Claim 4.8. Let G be a grid graph. With respect to the weight function w, for any two vertices u and v, the minimum weight path from u to v, if one exists, is unique.

Proof. For contradiction assume that there are two paths P_1 and P_2 from u to v such that they are both minimal with respect to the weight function w, so $w(P_1) = w(P_2)$. To fulfill this condition P_1 and P_2 there exist the point u' where they diverge for the first time and the point v' where they again meet. Consider a cycle C obtained by concatenating a subpath P_1' of P_1 from u' to v' and the subpath P_2' of P_2 from v' to u'. $A(C)$ must be nonzero by Claim 4.7 so $w(P_1') \neq w(P_2')$. Without loss of generality suppose that $w(P_1') < w(P_2')$. Since the weights of both P_1 and P_2 are equal, the weight of the subpath of P_2 from v' to v must be smaller than the weight of the subpath of P_1 from v' to v. Consider the path Q obtained by merging the subpath of P_1 from u to u', P_1' from u' to v' and then the subpath of P_2 from v' to v. $w(Q)$ is clearly smaller than $w(P_1) = w(P_2)$ so neither of them is a path of minimum weight, which is a contradiction. □

By using Claim 4.8 and Theorems 4.3 and 4.5 it follows that PlanarReach $\in \mathcal{UL} \cap co\text{-}\mathcal{UL}$. □

4.2.2 3D monotone Grid Graphs

3D grid graphs are a natural extension of grid graphs (Definition 4.1) obtained by adding a third coordinate. The vertices are then labeled by $[n] \times [n] \times [n]$. For every edge $e = ((i_1, j_1, k_1), (i_2, j_2, k_2))$ of a 3D grid graph it holds (similarly as in the 2D case) that $|i_1 - i_2| + |j_1 - j_2| + |k_1 - k_2| = 1$. A layer of a 3D grid graph is a subgraph induced by vertices with the same third coordinate. In addition to the edges in a 2D case we have two new kinds of edges that lead between layers. We denote edge $((i,j,k),(i,j,k+1))$ as an inward edge and edge $((i,j,k),(i,j,k-1))$ as an outward edge.

Definition 4.9 (3D monotone grid graph)**.** A graph G is a 3D monotone grid graph if it is a 3D grid graph that has only north, east and inward edges.

In further text we will denote this restriction of STCONN by 3DmGGR.

Theorem 4.10 ([BTV07])**.** 3DmGGR *is complete for* $\mathcal{NL}$*.*

In the proof of Theorem 4.6 we presented a weight function that makes a planar graph min-unique. It is not clear how to define a weight function for 3D monotone grid graphs though. The problem is that paths with the same weights do not have to cross each other any longer, and hence we are unable to use the same argument as in proof of Theorem 4.6. It is an open problem whether an appropriate weight function for 3D monotone grid graphs exist.

4.2.3 Thickness-two Graphs

The other class of graphs mentioned above is the class of thickness-two graphs. A graph G has thickness k if the minimum number of planar subgraphs of G whose union is G equals k. From the definition it follows that planar graphs are exactly thickness-one graphs. One should also notice that 3D monotone grid graphs are a subset of thickness-two graphs. To see this, embed all the layers in a plane, which can be easily done by placing the i-th layer above the $(i-1)$-th layer so they do not intersect. Then join the vertices of embedded layers with (inward) edges that lead between the layers. Such an embedding is a union of two planar graphs. One planar graph is created by the embedded layers and the second one by the inward edges, hence we obtain thickness-two graph.

Theorem 4.11 ([BTV07])**.** STCONN *for thickness-two graphs is complete for* $\mathcal{NL}$*.*

Showing that STCONN for thickness-two graphs is in $\mathcal{UL}$ would imply $\mathcal{UL} = \mathcal{NL}$ but finding an appropriate weight function to make such graphs min-unique seems to be hard.

5 Main result

In this chapter we will present a result on the min-uniqueness of the restricted class consisting of 3D monotone grid graphs with bounded height.

In the rest of this chapter, a 3D monotone grid graph G will be of size $n \times n \times k$ where k is a constant parameter. The vertices are triplets of numbers from $[n] \times [n] \times [k]$ and G has only east, north and inward edges.

Theorem 5.1. STCONN *for 3D monotone grid graphs with bounded height is in* $\mathcal{UL} \cap$ *co-*$\mathcal{UL}$.

Proof. First we define a weight function that is an extension of the weight function used in the proof of Theorem 4.6. For all edges $e = ((x, y, z), (x', y', z')) \in E(G)$,

$$w(e) = \begin{cases} n^4 n^{7z} & \text{if}\, e \,\text{is an east edge,} \\ (n^4 + y)n^{7z} & \text{if}\, e \,\text{is a north edge,} \\ 1 & \text{if}\, e \,\text{is an inward edge.} \end{cases}$$

We will be using Theorem 4.5 later on so we have to make sure that w fulfills all the assumptions. Thus we will show that w is log-space computable and polynomially bounded. To see that w is polynomially bounded we can upper-bound $n^4 n^{7z}$ by n^{4+7k} and $(n^4 + y)n^{7z}$ by $(n^4 + y)n^{7k}$. Since k is a fixed constant, w is polynomially bounded and because of its form it is clearly log-space computable.

We can imagine the edge weights as vectors (denote them $\mathbf{v}(e)$) of length k (the height of G) where for an edge $e \in E(G)$, all but the z-th coordinate of $\mathbf{v}(e)$ are equal to 0. This unique nonzero coordinate in the vector $\mathbf{v}(e)$ has value n^4 if e is an east edge and $n^4 + y$ if e is a north edge. Denote the i-th coordinate of $\mathbf{v}(e)$ by $\mathbf{v}_i(e)$.

$$w(e) = \begin{cases} \mathbf{v}_i(e) n^{7i} & \text{if } e \text{ is an east or north edge in } i\text{-th layer of } G, \\ 1 & \text{otherwise.} \end{cases}$$

For any path P from s to t in G we can define a vector $\mathbf{v}(P) = \sum_{e \in E(P)} \mathbf{v}(e)$. Its i-th coordinate $\mathbf{v}_i(P)$ is the sum of the weights of the east and north edges used by P in the i-th layer of G. From the definition of the weight function w, $\mathbf{v}_i(P)$ is of the form $a + bn^4$.

Notice that the sum of the weights of east and north edges of P in the i-th layer of G equals $\mathbf{v}_i(P)n^{7i}$. There may be at most n east edges with weight n^4 and at most n north edges with weights less or equal to n^4+n so the sum of the weights of P in the i-th layer of G is no more than $(2n^5+n^2)n^{7i} \leq 3n^{7i+5}$ and this is strictly less than $n^{7(i+1)} = n^{7i+7}$ for $n > 1$. Furthermore it holds that

$$\sum_{j=0}^{i} \mathbf{v}_j(P)n^{7j} < n^{7(i+1)}. \tag{5.0.1}$$

Thus, we can calculate the weight of P by $\sum_{i=1}^{k}(\mathbf{v}_i(P)n^{7i} + \operatorname{sgn}(\mathbf{v}_i(P))) - 1$.

Claim 5.2. The weight function w makes the graph G min-unique.

Proof. We proceed by induction on k. For $k = 1$ the statement follows from Theorem 4.6 because all the edges in G are of form $((x,y,0),(x',y',0))$. Thus, their weights are n^4 for east edges and n^4+y for north edges so w is identical to the weight function (4.2.1).

To prove the induction step assume that claim holds for some $k \geq 2$. We want to show that it also holds for $k+1$. Let $s = (x_s, y_s, z_s), t = (x_t, y_t, z_t) \in V(G)$ be two vertices. If $z_s \leq k$ and $z_t \leq k$ then from the induction hypothesis, there is either a unique path of minimal weight from s to t or there is no path from s to t.

Another trivial case is when $z_s = z_t = k+1$, that is z_s and z_t are both in the same layer. Since G is a monotone grid graph, all the possible paths from s to t use only the edges in the $(k+1)$-th layer. We may then remove all vertices $v \in V(G)$ with the z coordinate smaller than $k+1$ because they appear in no solution of the problem. By doing so we end up with the grid graph G' that is acyclic and has only east and north edges with weights $n^4n^{7(k+1)}$ and $(n^4+y)n^{7(k+1)}$, respectively. The term "$n^{7(k+1)}$" is the same for all edges in G' so it has no effect on the structure of minimal weight path from s to t. By omitting it we obtain exactly the weight function (4.2.1) on grid graph G'. Thus by applying Theorem 4.6 the statement follows.

The last case we have to verify is when $z_s \leq k$ and $z_t = k+1$. Let P be a path of minimal weight from s to t and let $e_k = ((x,y,k),(x,y,k+1))$ be an inward edge contained in a minimal path from s to t with following properties.

(P1) $x_t - x + y_t - y$ is minimal,

(P2) with respect to (P1), y is maximal.

Let v_k and v_{k+1} be the end vertices of the edge e_k. Because v_k is a vertex in k-th layer by the induction hypothesis there is a unique minimal path from s to v_k and by Theorem 4.6 there is a unique minimal path from v_{k+1} to t. Hence to prove the theorem it remains to show that e_k lies on every path of minimum weight from s to t.

For contradiction suppose that $e'_k = ((x', y', k), (x', y', k+1))$ does not satisfy (P1) and is an edge of P' (path of minimum weight from s to t). Thus

$$x_t - x' + y_t - y' > x_t - x + y_t - y. \tag{5.0.2}$$

Clearly $x_t - x'$ is the number of east edges that a path from v'_{k+1} to t has to use. Similarly $y_t - y'$ is the number of north edges that a path from v'_{k+1} to t contains. So the term $x_t - x' + y_t - y'$ represents the number of edges used by the path P' in the $(k+1)$-th layer. From (5.0.2) we get that $\mathbf{v}_{k+1}(P') > \mathbf{v}_{k+1}(P)$ and by using (5.0.1) it follows that P' is not a path of minimum weight, a contradiction.

By (P1) all the possible candidates for the edge e_k (their end vertices) have the same distance from t. We will show that condition (P2) serves to pick the right candidate.

Suppose for contradiction that P' is again a path of minimum weight containing an edge e'_k that fulfills (P1) but does not fulfill (P2). Hence e'_k has the same distance from t as e_k and $y' < y$. It means that a path from the end vertex of e_k uses more east edges than a path from the end vertex of e'_k. So $\mathbf{v}_{k+1}(P) = a + bn^4$ and $\mathbf{v}_{k+1}(P') = a' + b'n^4$ where $b = b' = x_t - x + y_t - y$ and because G is acyclic, P (resp. P') contains $y_t - y$ (resp. $y_t - y'$) north edges whose "y" coordinates form an arithmetic sequence, hence $a = \frac{y_t+y}{2}(y_t - y)$ and $a' = \frac{y_t+y'}{2}(y_t - y')$. Let $c = y - y'$. From the assumption that $y' < y$, it follows that

$$\begin{aligned} a' = \frac{y_t + y'}{2}(y_t - y') = \frac{y_t^2 - y'^2}{2} >& \\ >& \frac{y_t^2 - y'^2 - 2cy' - c^2}{2} = \frac{y_t + y' + c}{2}(y_t - y' - c) \\ =& \frac{y_t + y}{2}(y_t - y) = a, \end{aligned}$$

hence $\mathbf{v}_{k+1}(P) < \mathbf{v}_{k+1}(P')$ and from (5.0.1) we get a contradiction with the minimality of P'. $\square$

This shows that the weight function w makes G a min-unique graph. The proof is completed by applying Theorem 4.5. $\square$

By restricting to a subclass of 3D monotone grid graphs we managed to prove that the reachability problem for these graphs is in $\mathcal{UL} \cap co\text{-}\mathcal{UL}$. Note that the extension of the weight function to the general 3D monotone grid graphs is not possible because due to the term exponential in the height of the graph the weight function w will no longer be polynomially bounded.

6 Conclusion

In the first part of the thesis we gave an overview of some basic facts from computational complexity theory followed by further details about logarithmic space complexity classes.

The next part was devoted to an open problem whether $\mathcal{NL} = \mathcal{UL}$. We summarized some known results and mentioned several important techniques.

In the last part we presented a proof that the directed connectivity problem on 3D monotone grid graphs with bounded height is in $\mathcal{UL}$ adding another problem into this class.

Many open problems remain in this field and it would be interesting to investigate them further.

A major question is, of course, whether $\mathcal{NL} = \mathcal{UL}$. To approach this problem it might be useful to consider some restricted classes of graphs and prove that the directed connectivity problem remains $\mathcal{NL}$-complete for such classes. One might consider following classes such as:

- 3D monotone grid graphs that are union of k forests,
- 3D monotone grid graphs with no two consecutive edges of same kind (if a vertex v is an end vertex of a north edge then it cannot be a start vertex of another north edge and similarly for east and inward edges),
- 3D monotone grid graphs with out-degree of vertices smaller or equal 2.

All these classes are restriction of the general 3D monotone grid graphs. It might be easier to find a weight function that would make such graphs min-unique and by Theorem 4.5 prove that $\mathcal{NL} = \mathcal{UL}$.

Another problem known to be $\mathcal{NL}$-complete is 2-SAT ([Pap94] Theorem 16.3). 2-SAT is a problem of satisfiability of Boolean formula in conjunctive normal form with two variables per clause. Is it possible to use 2-SAT for solving $\mathcal{NL} = \mathcal{UL}$?

Bourke et al. [BTV07] noted that there are no problems known to be complete for $\mathcal{UL} \cap co\text{-}\mathcal{UL}$. Do any such problems even exist?

We saw that directed planar reachability is in $\mathcal{UL}$ (Theorem 4.6). Limaye et al. [LMN09] presented a result about a problem of the longest path in planar directed acyclic graph and showed that it is also in $\mathcal{UL}$. Even more recent result comes from Thierauf and Wagner [TW10]. They showed that the isomorphism problem for planar 3-connected graphs is in $\mathcal{UL}$. Are there any other problems contained in $\mathcal{UL}$?

Bibliography

[AB09] S. Arora and B. Barak. *Computational Complexity: A Modern Approach.* Cambridge University Press, 2009.

[ADR05] Eric Allender, Samir Datta, and Sambuddha Roy. The directed planar reachability problem. In *In Proc. 25th annual Conference on Foundations of Software Technology and Theoretical Computer Science, number 1373 in Lecture Notes in Computer Science*, pages 238–249. Springer, 2005.

[AG00] Carme Àlvarez and Raymond Greenlaw. A compendium of problems complete for symmetric logarithmic space. *Comput. Complex.*, 9:123–145, April 2000.

[AJ93] Carme Àlvarez and Birgit Jenner. A very hard log-space counting class. *Theoretical Computer Science*, 107(1):3 – 30, 1993.

[AKG12] Scott Aaronson, Greg Kuperberg, and Christopher Granade. Complexity zoo. `http://qwiki.stanford.edu/index.php/Complexity_Zoo`, 2012.

[AM04] Eric Allender and Meena Mahajan. The complexity of planarity testing. *Inf. Comput.*, 189(1):117–134, February 2004.

[BJLR91] G. Buntrock, B. Jenner, K. J. Lange, and P. Rossmanith. *Unambiguity and fewness for logarithmic space*, volume 529 of *Lecture Notes in Computer Science.* Springer Berlin / Heidelberg, 1991.

[BTV07] Chris Bourke, Raghunath Tewari, and N. V. Vinodchandran. Directed planar reachability is in unambiguous logspace. In *In Proceedings of IEEE Conference on Computational Complexity CCC*, 2007.

[Cla12] Millennium prize problems. `http://www.claymath.org/millennium/`, 2012.

[CM87] Stephen A. Cook and Pierre McKenzie. Problems complete for deterministic logarithmic space. *J. Algorithms*, 8(5):385–394, September 1987.

[Coo71] S.A. Cook. *The Complexity of Theorem-proving Procedures.* 1971.

[Die06] R. Diestel. *Graph Theory.* Graduate Texts in Mathematics. Springer, 2006.

[DP11] Samir Datta and Gautam Prakriya. Planarity testing revisited. In *Proceedings of the 8th annual conference on Theory and applications of models of computation*, TAMC'11, pages 540–551, Berlin, Heidelberg, 2011. Springer-Verlag.

[Gol08] Oded Goldreich. *Computational complexity: a conceptual perspective.* Cambridge University Press, 2008.

[HLWO06] Shlomo Hoory, Nathan Linial, Avi Wigderson, and An Overview. Expander graphs and their applications. *Bull. Amer. Math. Soc. (N.S*, 43:439–561, 2006.

[HMU06] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[Imm88] Neil Immerman. Nondeterministic space is closed under complementation. *SIAM J. Comput.*, 17(5):935–938, October 1988.

[Jan98] Jesper Jansson. Deterministic space-bounded graph connectivity algorithms. `http://www.df.lth.se/~jj/Publications/STCON2.ps`, 1998.

[Koz06] Dexter Kozen. *Theory of computation.* Texts in computer science. Springer, 2006.

[LMN09] Nutan Limaye, Meena Mahajan, and Prajakta Nimbhorkar. Longest paths in planar dags in unambiguous logspace. In *Proceedings of the Fifteenth Australasian Symposium on Computing: The Australasian Theory - Volume 94*, CATS '09, pages 101–108, Darlinghurst, Australia, Australia, 2009. Australian Computer Society, Inc.

[LP82] Harry R. Lewis and Christos H. Papadimitriou. Symmetric space-bounded computation. *Theoretical Computer Science*, 19(2):161 – 187, 1982.

[LR97] Maciej Liśkiewicz and Rüdiger Rieschuk. Computing with sublogarithmic space. 1997.

[NTS95] Noam Nisan and Amnon Ta-Shma. Symmetric logspace is closed under complement. In *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, STOC '95, pages 140–146, New York, NY, USA, 1995. ACM.

[Pap94] C.H. Papadimitriou. *Computational complexity.* Addison-Wesley, 1994.

[RA98] Klaus Reinhardt and Eric Allender. Making nondeterminism unambiguous. In *SIAM Journal of Computing*, pages 244–253, 1998.

[Rei85] John H. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20(5):229 – 234, 1985.

[Rei08] Omer Reingold. Undirected connectivity in log-space. *J. ACM*, 55(4):17:1–17:24, September 2008.

[RTV05] O. Reingold, L. Trevisan, and S. P. Vadhan. Pseudorandom walks in biregular graphs and the rl vs. l problem. *Electronic Colloquium on Computational Complexity (ECCC)*, 2005.

[RVW00] O. Reingold, S. Vadhan, and A. Wigderson. Entropy waves, the zig-zag graph product, and new constant-degree expanders and extractors. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, FOCS ’00, pages 3–, Washington, DC, USA, 2000. IEEE Computer Society.

[Ryj09] Z. Ryjáček. Teorie grafů a diskrétní optimalizace 2. `http://www.kma.zcu.cz/TGD2`, 2009.

[Ryj11] Z. Ryjáček. Teorie grafů a diskrétní optimalizace 1. `http://www.kma.zcu.cz/TGD1`, 2011.

[Sav70] Walter J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177 – 192, 1970.

[Sch03] Alexander Schrijver. *Combinatorial optimization: polhyedra and efficiency*. Algorithms and combinatorics. Springer, 2003.

[Sip06] M. Sipser. *Introduction To The Theory Of Computation*. Computer Science Series. Thomson Course Technology, 2006.

[Sze88] Róbert Szelepcsényi. The method of forced enumeration for nondeterministic automata. *Acta Informatica*, 26:279–284, 1988.

[TW10] Thomas Thierauf and Fabian Wagner. The isomorphism problem for planar 3-connected graphs is in unambiguous logspace. *Theory of Computing Systems*, 47:655–673, 2010. 10.1007/s00224-009-9188-4.

[Val76] Leslie G. Valiant. Relative complexity of checking and evaluating. *Information Processing Letters*, 5(1):20 – 23, 1976.