

Algorithms and Networks

Logspace

Hans Bodlaender and Stefan Kratsch

April 14, 2011



Universiteit Utrecht

1

Introduction
Sorting in
logspace
A kernelization
in logspace
Some more facts
about logspace
Summary

Computation on restricted space

- ▶ most of the problems considered in recent lectures were NP-hard (or worse)
- ▶ for all other problems, we were content if we could solve them in polynomial time
- ▶ sometimes, even polynomial time may not be good enough
- ▶ in particular if it involves more space (informally: RAM) than we can afford
- ▶ different restrictions/models address this:
 - logspace computation (today)
 - streaming model
 - memory hierarchies



Universiteit Utrecht

2

Introduction
Sorting in
logspace
A kernelization
in logspace
Some more facts
about logspace
Summary

Computation on logarithmic space

today: What can you decide or compute when you only have space $\mathcal{O}(\log n)$ for inputs of size n ?

informally:

- ▶ the input is provided on a read-only device (a DVD, a database,...)
- ▶ you can produce as much output as you want (more than $\mathcal{O}(\log n)$)
- ▶ but it is write-only and your algorithm cannot read it again (similar to printing)



Universiteit Utrecht

3

Introduction
Sorting in
logspace
A kernelization
in logspace
Some more facts
about logspace
Summary

Again: why so little space?

- ▶ we may have much more input than we have RAM (think: databases, online repositories)
- ▶ to learn different algorithm design paradigms
- ▶ to understand (a bit) space-time-trade offs (e.g. things will take longer when saving space)
- ▶ to explore the structure of classes inside P



Universiteit Utrecht

4

Introduction
Sorting in
logspace
A kernelization
in logspace
Some more facts
about logspace
Summary

Why specifically logarithmic space?

- ▶ you need logarithmic space so you can at least address every position of the input
- ▶ it's enough to count up to n in binary
- ▶ mostly we will use $\mathcal{O}(\log n)$ to have multiple pointers to positions of the input
- ▶ it has turned out that this suffices to do decide/compute interesting things
- ▶ it's the largest amount of space that guarantees that we can spend only polynomial time



5

Introduction

Sorting in logspace

A kernelization in logspace

Some more facts about logspace

Summary

Outline

Introduction

Sorting in logspace

A kernelization in logspace

Some more facts about logspace

Summary



6

Introduction

Sorting in logspace

A kernelization in logspace

Some more facts about logspace

Summary

Outline

Introduction

Sorting in logspace

A kernelization in logspace

Some more facts about logspace

Summary



7

Introduction

Sorting in logspace

A kernelization in logspace

Some more facts about logspace

Summary

Sorting in logspace

- ▶ assume you are given a long string, say length n
- ▶ the string contains numbers in decimal, separated by commas, e.g.,

42, 4711, 2, 3, 5, 11, 4523890562389015231, 10, ...

- ▶ you want to output the numbers in the string in sorted order
- ▶ but you only have space $\mathcal{O}(\log n)$, i.e., you cannot keep more than $c \cdot \log n$ symbols in memory



8

Introduction

Sorting in logspace

A kernelization in logspace

Some more facts about logspace

Summary

Sorting in logspace

- ▶ we cannot do things like QuickSort or MergeSort, cause we have no space to write down sorted substrings*
- ▶ also there is no chance of having a data structure that contains all numbers
- ▶ in fact, single numbers might be longer than $c \cdot \log n$ positions
- ▶ essentially we are restricted to having pointers to positions in the input (length n , so this needs $\log n$ space for a binary counter)

*if the input was rewriteable, then *inplace* variants of QuickSort or MergeSort would help



Let's do something very simple

- ▶ what we can do is repeatedly finding the minimum number in the string and outputting that
- ▶ to do this, pass over the input, compare each number to the minimum one encountered
- ▶ do this only for those which are larger than the largest one output so far

again: we cannot necessarily store any number in memory



We keep the following in memory

in memory:

- ▶ current position (e.g. the number that we want to compare)
- ▶ position of the number last committed to output
- ▶ position of the current minimum

a detail: (easy) how to compare large numbers?

note: position could always be the rightmost position of the number in question



Final sorting algorithm (sketch)

1. pointer to first number as candidate for smallest
2. compare each number to the smallest, updating its pointer if a smaller is found
3. write the smallest number to the output, and keep a pointer to it for being the largest number already written
4. repeat the above, but
 - skip numbers that are smaller than the one last written (compare again to one that is pointed to)
 - for Step 1. pick the first number that is larger than the last written
 - if all numbers are larger than the last written, then stop
 - don't forget the comma ;-)

for simplicity: let all numbers be different (what to do if they are not?)



Outline

Introduction

Sorting in logspace

A kernelization in logspace

Some more facts about logspace

Summary

Introduction

Sorting in logspace

A kernelization in logspace

Some more facts about logspace

Summary



Universiteit Utrecht

13

A kernelization in logspace

Vertex Cover

Input: A graph G and an integer k .

Parameter: k .

Output: Is there a set S of at most k vertices such that each edge of G has at least one endpoint in S ?

- ▶ we have seen kernels with $2k^2$, $3k$, and $2k$ vertices
- ▶ now we would like to get the $2k^2$ kernel by a logspace computation
- ▶ main idea for that was the high-degree rule (if a vertex has degree $> k$ it must be selected)

Introduction

Sorting in logspace

A kernelization in logspace

Some more facts about logspace

Summary



Universiteit Utrecht

14

Setting

- ▶ let us assume that the instance is given by the adjacency matrix of G (i.e. a 0/1-matrix encoding the edges)
- ▶ followed by $\#$ and k in binary
- ▶ let N be the length of the string
- ▶ we have $\mathcal{O}(\log N)$ memory
- ▶ we have to write an adjacency matrix and a parameter value that encode an equivalent instance

example: 011101110#10

corresponding to the 3-vertex clique with parameter $k = 2 = 10_2$

0	1	1
1	0	1
1	1	0

Introduction

Sorting in logspace

A kernelization in logspace

Some more facts about logspace

Summary



Universiteit Utrecht

15

First steps

- ▶ compute the number of vertices by counting the length of the adjacency matrix, say L (space $\log L < \log N$)
- ▶ (not hard) find a way to get the square root of L , and store that value as n
- ▶ read the binary number after the $\#$ mark
- ▶ if it is larger than n then we answer YES (details later)
- ▶ else its value is at most n and we can store it as k in $\log n < \log N$ bits

Introduction

Sorting in logspace

A kernelization in logspace

Some more facts about logspace

Summary



Universiteit Utrecht

16

More steps

- ▶ count the number of vertices of degree greater than k
- ▶ to do so count the number of ones in each of the n rows of the adjacency matrix
- ▶ store as D the number of these high degree vertices (i.e. with more than k ones in the corresponding row)
- ▶ any solution of size at most k must contain those D vertices
- ▶ if D is greater than k then we answer NO
- ▶ we cannot explicitly store these vertices: to identify up to k vertices we need $k \cdot \log n$ space

Introduction

Sorting in logspace

A kernelization in logspace

Some more facts about logspace

Summary

Now we want to output all edges that do not have a high-degree endpoint, i.e., the adjacency matrix corresponding to the remaining vertices.



Universiteit Utrecht

17

Creating the output (almost)

1. for each row of the adjacency matrix
 - 1.1 if it has more than k ones (count again) then continue with next row
 - 1.2 for each position of the row
 - 1.2.1 if the vertex of that position has high degree (count the corresponding row) then continue with next position
 - 1.2.2 else write that position to the output
2. write $\#$ to the output
3. compute $k - D$ (the remaining budget) and write the outcome in binary to the output

Introduction

Sorting in logspace

A kernelization in logspace

Some more facts about logspace

Summary

Observe how we avoid the rows and columns corresponding to high degree vertices.



Universiteit Utrecht

18

Creating the output

- ▶ the kernelization also counted the edges and rejected if k vertices of degree at most k must cover $> k^2$ edges!
- ▶ additionally it deleted isolated vertices
- ▶ thus, run the algorithm as outline, but only count the ones that would be written (don't write)
- ▶ if more than k^2 edges then reject (write some small no-instance)
- ▶ else, run the algorithm as before but do the pass over each row twice
 1. check only if the vertex would be isolated by looking at what you would write
 2. if it's not isolated then write the row as before, except that you also have to check for each column, whether that vertex is kept...

Introduction

Sorting in logspace

A kernelization in logspace

Some more facts about logspace

Summary



Universiteit Utrecht

19

Outline

Introduction

Sorting in logspace

A kernelization in logspace

Some more facts about logspace

Summary

Introduction

Sorting in logspace

A kernelization in logspace

Some more facts about logspace

Summary



Universiteit Utrecht

20

Logspace

- ▶ similarly to P and NP ...
- ▶ ...one can define classes L and NL of languages that can be decided in space logarithmic in the input, deterministically (L) or nondeterministically (NL)
- ▶ same, one may define classes of functions that can be computed in logarithmic space
- ▶ it can be seen that L and NL are subclasses of P
- ▶ main observation for this: space $\mathcal{O}(\log n)$ has at most $\mathcal{O}(c^{\log n}) = \mathcal{O}(n^{c'})$ configurations

$$L \subseteq NL \subseteq P \subseteq NP \subseteq PSPACE = NPSPACE$$



Logspace subroutines

- ▶ like polynomial time algorithms...
- ▶ ...logspace algorithms can use other logspace algorithms as subroutines
- ▶ clearly the total space is $\mathcal{O}(\log n)$
- ▶ but there are some intricate details, e.g., there may not be enough space to write the input for the subroutine, or its output
- ▶ those are also implicit when considering logspace reductions

solution: use variants of the algorithms that can produce a requested position of their output – instead of creating the whole output, you run them on demand



Some interesting results/facts

- ▶ given a directed graph and vertices s, t it is complete for NL to decide whether there is a directed path from s to t
- ▶ fairly recent work of Reingold (2004), showed that the undirected version is in L , and thereby complete for L
- ▶ following from this various simple connectivity problems are complete for L
- ▶ a tree decomposition of G of width at most k can be computed in space $\mathcal{O}(f(k) \cdot \log n)$ if it exists, (Elberfeld et al. 2010)
- ▶ isomorphism of planar graphs can be decided in logspace (Datta et al. 2009)



Outline

Introduction

Sorting in logspace

A kernelization in logspace

Some more facts about logspace

Summary



Take away

- ▶ there are interesting refinements of P defined by restricted space
- ▶ important: only used memory is restricted
- ▶ one may generate more output, but all output is write-only (and non-erasable)
- ▶ our runtime is limited to polynomial in n , unless we have an endless loop (there are only $2^{c \log n}$ many states in the RAM)
- ▶ like P vs. NP there is a distinction into deterministic and nondeterministic variant



Introduction

Sorting in logspace

A kernelization in logspace

Some more facts about logspace

Summary

Open problems

- ▶ $L = NL?$ (logspace-equivalent of P vs. NP)
- ▶ $L = P?$



Introduction

Sorting in logspace

A kernelization in logspace

Some more facts about logspace

Summary

Questions?



Introduction

Sorting in logspace

A kernelization in logspace

Some more facts about logspace

Summary