Dear EATCS members,

First of all, I belatedly wish you all the best for 2013. This year will be marked by the 40th ICALP conference, which will be held in Riga, Latvia, in the period 8–12 July 2013. The submission deadline for ICALP 2013 has just passed. The conference has received 427 submissions overall (250 for Track A on ‘Algorithms, Complexity and Games’, 116 for Track B devoted to ‘Logic, Semantics, Automata and Theory of Programming’, and 61 for Track C dealing with ‘Foundations of Networked Computation: Models, Algorithms and Information Management’). These are excellent numbers of submissions and bode well for a high-quality scientific programme in keeping with the standards of the ICALP conference. I wish the track chairs (Fedor V. Fomin, Marta Kwiatkowska and David Peleg), their PCs and the conference organizers the best of luck with the hard work that awaits them over the coming months.

Several other events will contribute to making the 40th ICALP an inspiring scientific meeting. First of all, I am pleased to inform you that Jon Kleinberg (Cornell University) will deliver a special EATCS Lecture at ICALP 2013 to mark the 40th anniversary of ICALP. This lecture is in addition to the previously announced invited talks, which will be given by Susanne Albers (Humboldt University), Orna Kupferman (Hebrew University), Paul Spirakis (University of Patras), Daniel Marx (Hungarian Academy of Sciences) and Peter Widmayer (ETH Zürich).
Second, I am happy to announce that the EATCS Award 2013 Committee, consisting of Friedhelm Meyer auf der Heide (chair), Leslie Ann Goldberg and Vladimiro Sassone, has selected Martin Dyer (University of Leeds) as the recipient of the EATCS Award 2013. The Presburger Award 2013 Committee, manned by Monika Henzinger (chair), Antonin Kucera and Peter Widmayer, has instead proposed Erik Demaine (MIT) as the recipient of the Presburger Award 2013. On behalf of the EATCS, I congratulate Erik and Martin for these prestigious awards. Both Erik and Martin will deliver award presentations at ICALP 2013 and have agreed to contribute a piece on their work to the October 2013 issue of the Bulletin of the EATCS.

In passing, I warmly thank CWI Amsterdam for sponsoring the Presburger Award 2013. The invited talks and the award presentations at ICALP 2013 will be taped and will be made available on the web after the conference. I expect that this will be the norm from now on at ICALP conferences and will further increase the impact of the conference on the TCS community.

In order to encourage joint registration to the EATCS and its French, Italian and Japanese chapters, from 2013 the EATCS will offer a five euro discount on the EATCS registration fee for those who register to both the EATCS and one of its chapters. I hope that many members of the TCS community in those countries will take advantage of this possibility. Let me also remind you that the EATCS offers a five euro discount per year to SIGACT members.

The preparations for ICALP 2014 in Copenhagen are going well. The PC chairs
for the three tracks of ICALP 2014 will be Elias Koutsoupias (University of Oxford) for Track A, Javier Esparza (Technische Universität München) for Track B and Pierre Fraigniaud (Université Paris Diderot) for Track C. Elias, Javier and Pierre are currently forming the PCs for their tracks and are selecting the invited speakers for ICALP 2014. As usual, the call for papers for ICALP 2014 will be available at ICALP 2013 in Riga.

We are also making plans for hosting the first ICALP conference outside Europe in 2015. Kazuo Iwama (ICALP) and Masahito Hasegawa (LICS) are making excellent progress in their planning for hosting a joint ICALP-LICS 2015 in Kyoto in the period 6–10 July 2015. Kazuo Iwama will present the plans for having ICALP 2015 in Kyoto at the General Assembly during ICALP 2013 in Riga, where I look forward to meeting many of you.

Luca Aceto, Reykjavik
February 2013
1 Scientific and Community News

0. The latest CDMTCS research reports are (http://www.cs.auckland.ac.nz/staff-cgi-bin/mjd/secondcgi.pl):

425. S. Hartmann and S. Link. The Implication Problem of Data Dependencies over SQL Table Definitions: Axiomatic, Algorithmic and Logical Characterizations, 10/2012

426. M. Kirchberg, S. Hartmann and S. Link. Design by Example for SQL Table Definitions with Functional Dependencies, 10/2012

427. F. Ferrarotti, S. Hartmann and S. Link. Efficiency Frontiers of XML Cardinality Constraints, 10/2012

428. S. Link. Sound Approximate Reasoning about Saturated Conditional Probabilistic Independence under Controlled Uncertainty, 10/2012

429. C.S. Calude, E. Calude and M.S. Queen. Inductive Complexity of the P Versus NP Problem, 10/2012
2 A Dialogue with Ian H. Witten: My story so far: a stroll through the gardens of computer science

Cristian Calude: How was computer science during your studies at Cambridge University (MA in Mathematics), the University of Calgary (MSc in Computer Science) and Essex University (PhD in Electrical Engineering)?

Ian H. Witten: Very different from today! At Cambridge in 1968 I took a course on numerical analysis from the famous computer science pioneer Maurice Wilkes, but I’m sorry to say I found it very boring. I wrote a couple of programs in a language called ÔFocalÓ, a precursor to Basic I think. At Calgary in 1969–70 I met both Fortran (punched cards) and a PDP-12 (teletype and paper tape), which I used interactively—very cool, because I could actually see the computer! At
Essex in the early 1970s I was bequeathed an ancient PDP-1, and had a lab of several PDP-8s, each with 4 KB RAM—one even had a massive 10 KB disk. Around 1976 I installed the second UNIX installation in the UK. I loved UNIX because of its openness (still do). All of these computers had less power than a digital watch does today.

CC: Did the education in three subjects—Mathematics, Computer Science and Electrical Engineering—help you well in your career?

IHW: It’s a great selling point to have degrees that cover these three fields; everyone’s terrifically impressed. But in actuality my Computer Science MSc and Electrical Engineering PhD were in very similar fields, which would be more accurately called applied statistics. I do think that mathematics is a great foundation for thinking in general, though I have no regrets about abandoning my early aspirations to become an actuary in favour of moving to computer science.

CC: You lived in three corners of the world and eventually settled in New Zealand in 1991. What was the motivation?

IHW: When others ask me that I tell them that if they had ever been to New Zealand the answer would be obvious, but you know that already, Cris. It’s like asking someone in heaven why they chose that over the alternative. I wanted a more relaxed lifestyle with a greater emphasis on a balanced life and more outdoor recreational opportunities, but also with a good hi-tech environment. New Zealanders are very quick to pick up on new technologies, particularly networking, which is obviously very important over here.

CC: As you said, I know and I fully agree. Tells us about Greenstone, the suite of software for building and distributing digital library collections.

IHW: Greenstone emerged from a desire to apply the Managing Gigabytes indexing and searching techniques on a grander scale. A key event was our involvement with UNESCO beginning around 1999/2000, which happened completely by accident. It made me aware of the enormous potential of end-user-built collections for disseminating information in developing countries. Of course, Greenstone is not just for developing countries—it’s widely used almost everywhere, including the US—but I became passionate about its use in the developing world. We take libraries, and the internet, and the ready availability of information, for granted; but life is very different in other places—and the degree of western cultural hegemony is awful. Greenstone enables people to build and disseminate collections of their own information, in their own language—the interface has been translated, by volunteers, into over 50 languages, some of which you’ve probably never heard of. And it’s been a tremendous personal opportunity for me as well: Greenstone has taken me to meet new friends in places like Cuba, Fiji, Micronesia, Nepal, Trinidad, as well as several African countries and all over India.
CC: Fascinating, I think you have a myriad of stories from these trips.

IHW: Stories and stories. I’ll never forget a week-long workshop in a packed computer lab in Havana with ancient Windows computers and no air conditioning. It was glorious bedlam!—my first real encounter with the Hispanic temperament, I guess. People chatting and singing and flirting on the side, and me fighting to retain control. And then the computers started acting up. “Over here, Ian, Greenstone’s stopped working on this one.” “And here too!” I’d take a look: garish windows popping up autonomously all over the screen. Viruses. One by one the lab computers succumbed, until at the end only a couple were still usable. I since learned that viruses, essentially invisible in my own, obviously well-protected, computer environment, are the curse of the developing world, a productivity-sapping “white man’s disease” created by affluent westerners that cripples poor countries that lack the technical support needed to fight them. Once (and only once) I passed my memory stick around a class in Africa to distribute sample files: it took just minutes to pick up hundreds of viruses! Spam email is a similar issue: virtually non-existent for me at home, but a plague in these people’s lives. Imagine deleting hundreds of spam messages on a painfully slow internet connection, every day. I have learned to respect other things that we take for granted, like electricity. Just minutes before starting a hands-on workshop in a computer lab in Kathmandu I was told of the scheduled rotating 8-hour electricity cuts (“didn’t you know?”), and today’s began ... (“let me check the schedule”) ... now! And different cultural mores: after a conspiratorially whispered question “do you like alcohol?” (the answer is obvious to those who know me) I was hijacked from a lively and eagerly-anticipated student party in India by smart-suited top officials who insisted I join them in a clandestine, prolonged, and rather dismal drinking session.

CC: Please explain the importance of “keyphrase indexing” and your work in this subject.

IHW: Here’s an area where computers outperform people at a task that is obviously human! When you choose keyphrases, or index terms or whatever you want to call them, for a document, the aim is to be consistent with what other people are likely to choose, because ultimately the terms are going to be used for searching and browsing by others in order to find the information they seek. The aim is consistency with others, doing what others would do—to be boring, if you like. And success can be measured in terms of the degree of agreement with other people who independently index the same documents. In terms of this measure—agreement with humans—our experiments have shown that computer indexing technology can outperform “ordinary” people and even rival specialists, including professional indexers. That sounded outrageous when my student speculated that that was possible, but she turned out to be right and I was wrong. This
seems to happen a lot with me and my students.

CC: Your book *Managing Gigabytes* (co-authored with Alistair Moffat and Timothy C. Bell) published by Morgan Kaufmann Publishing, San Francisco in two editions, is about compressing and indexing documents and images. “This book is the Bible for anyone who needs to manage large data collections”, wrote Steve Kirsch, cofounder of Infoseek. How do you see this subject after 18 years since the first edition and 13 years since the second one?

IHW: The Bible?—I know I live in heaven, but did Steve really say that?

CC: Yes, he did.

IHW: I also heard that the book was required reading for early employees at Google. We made a big mistake by saying in the first edition’s Preface that maybe the second edition would be called *Managing Terabytes*, and we had to eat our words when we came to the second edition. I’d love to write Managing Terabytes, but in truth the problems are completely different when you move upscale. What amazes me about today’s search engines is not so much that they can answer queries so quickly but that they can keep going in the face of continuous failure. With a hundred thousand disks, a dozen must fail every day; with a million, one must fail every ten mins. And it’s not just disks…

CC: Data deluge is not a danger? Sciences are shifting to engineering by using statistical techniques to sniff through huge databases to find patterns, and, amazingly, with good and very good predictive results. This is fine unless this paradigm “kills” one of the important scientific quests, the effort to understand. This issue was discussed in a very interesting conversation between Noam Chomsky and Yarden Katz titled *On Where Artificial Intelligence Went Wrong*, recently published in *The Atlantic*.

IHW: Yep, understanding is indeed degenerating in favour of publishable “results”, and I regret to say that I have personally contributed to the problem. I cringe when I read papers that compare this machine learning technique to that one on half a dozen standard datasets and present nothing but a nicely-formatted table of statistical results (to 5 significant digits), with no insights at all. This is pointless research, Weka-enabled. The pressure to publish has become all-consuming. (The only upside, and it’s not a very big one, is that such “research” does contribute to my citation count. (That was a joke, by the way.))

As the years pass I’ve become less interested in philosophical issues surrounding artificial intelligence, brain theory, the mind/body problem, consciousness, and so on. However, one strand of my current work is stimulated by the need for computers to apply knowledge rather than, or as well as, statistics. I think that Wikipedia, although embarrassingly primitive and limited at present, signals a sea change in how our society deals with knowledge. A few hundred years ago, control of society’s knowledge was wrested from the Church and relocated
in academic institutions. Now, to the great chagrin of us university professors, our monopoly is under threat: society can collaboratively create, edit, and refine knowledge artefacts without even asking us! (And we are threatened on the teaching side too by the rise of instant internet universities and MOOCs, but that’s another story.) An important side effect is that computers can now pursue these knowledge artefacts and benefit from them too. So I’m interested in knowledge mining from Wikipedia and other public information sources. Also—remember Cyc, Doug Lenat’s common-sense knowledge project from the 1980s? The project’s still going, and although it may seem as though time (and crowdsourcing) has passed it by, Cyc does contain a wealth of core knowledge (about disjunctive concepts, for example, and argument restrictions) carefully hand-coding by professionals. Some of our current work is aimed at reaping that and using it for ontological quality control of information garnered from the Linked Data movement and inferred from less reliable sources such as Wikipedia.

**CC:** Your book *Web Dragons* was described as “not a resource on how search engines work, but rather what ideas and ideals have been realised in the development of search engines, the political and human challenges they face and problems and opportunities they present to humans and to the nature of knowledge and information.” How do you see the future of search engines? What about their social role?

**IHW:** Boy, if only I could answer questions like that! *Web Dragons* was written in 2006, before the astounding rise of Facebook and Twitter, and predicted a social dimension to information retrieval that has now become commonplace. But the future? I think people are less interested in information than I realised, and more interested in phatic communication. Obviously I think social interaction is important, but so is real information—and perhaps it’s getting lost in our obsession with trivia. Personally I prefer my social interactions to be face to face. Several followers of my Twitter account are still eagerly awaiting my first post!

**CC:** Machine learning is another area in which your group has excelled over the years. You have written state-of-the-art software for developing machine learning techniques and then applied it to real-world data mining problems. Please describe the software WEKA and MOA.

**IHW:** When I came to New Zealand in 1992 I wanted to initiate a project that enabled people in the rather obscure little computer science department I had joined to work together and develop new lines of research. We hit upon machine learning as a technology that was interesting, futuristic-sounding, and had potential relevance to agriculture, the staple regional and national industry. We began work on a C++ machine learning workbench, which at the time was in stiff competition with Stanford’s MLC++, and moved to Java early on, which was risky because of performance issues at the time but proved to be an excellent decision.
WEKA has really taken off—which is astonishing considering that it denotes a small flightless bird. But meanwhile I have moved on to other projects, and have not been involved with MOA, a stream-oriented ML project that stands for Massive Online Analysis and denotes a huge extinct New Zealand bird twice the size of an ostrich.

CC: What industrial applications have you developed?

IHW: I’m not very good at working directly with industry. Everything my students and I do is issued as open source software, which is picked up by academics and industry alike. As well as the better known Weka, Greenstone, and Wikipedia Miner, we have Kea (keyphrase assignment), MAUI (multi-purpose automatic topic indexing), Realistic Books, Katoa (knowledge assisted text organisation algorithm), FLAX (flexible language acquisition), and, very recently, FFTS (the fastest FFT in the south, and considerably faster than FFTW, the fastest in the west). All done by my students, I should emphasise; not by me.

CC: Programming and proving are very similar mental activities. It took me a long time to understand that programming is more demanding than proving: the difference comes from the agent validating the product—code or proof, a computer or a human expert. What is programming by demonstration?

IHW: Programming by demonstration involves showing a computer what to do rather than instructing it in some programming language. People often have to do boring, repetitive tasks on computers—reformatting references or addresses; processing lists; drawing sequences of boxes. Given such a chore, perhaps you should write a program—or perhaps it’s quicker to just go ahead and do the job manually. Some of my early research was on calculators that inferred iterative computations from the beginning of a sequence of key-presses, and a predictive typing interface for the disabled that set the scene for predictive text entry on today’s cellphones. My students and I created a “smart mouse” that automated repetitive graphical editing tasks, an instructible interface that acquired data descriptions and procedures by being taught rather than programmed, and a programming-by-demonstration agent that worked with a set of common, unmodified applications on a popular computer platform. This was amongst my best and most creative research: highly interactive interfaces that incorporate a learning component, with enormous potential to expedite many human-computer dialogues. But I became discouraged. Reviewers rejected our papers, demanding more tightly controlled human evaluation, which we could not deliver because interactions between a user and a “learning agent” evolve over time.

CC: Even top researchers can get discouraged...

IHW: I don’t know about top researchers, but I certainly can. Discouraged and rejected. My Dean once said “it’s all right for you, Ian; no-one ever rejects your papers”. But nothing could be further from the truth! I’m sure there can’t be
many people who’ve had more rejections than me. Our book *Computer Science Unplugged* (with Tim Bell, principal author, and Mike Fellows) was rejected by 27 publishers before we gave up. Sixteen years later it has spawned a major movement in the teaching of computer science to school children all over the world, including the US and UK, and has been translated (with our permission) into Arabic, Chinese, French, German, Hebrew, Italian, Japanese, Korean, Polish, Portuguese, Russian, Spanish, and Swedish. This has been possible because, since it’s unpublished, we still own the copyright. The silver lining.

**CC:** If your Dean thinks that you (read: as a top researcher) are spared the misery of rejection (sometimes without real base, using unnecessary harsh arguments like “my weakest student would have done it better”, “there is no subject for the paper”), imagine how the younger colleagues imagine the “status” of the academic establishment . . .

**IHW:** Well, this was just one offhand remark of one Dean: other Deans I have worked with are far better informed about the realities of academic publishing. However, I agree with you entirely: the academic establishment is often really tough on younger colleagues. And I have learned that university administrators tend to present different persona to senior professors than to junior staff (which I think is reprehensible), so that in many cases I experience entirely different, and more humane, personalities than colleagues do. But neither do I think university academics should complain too much: life is far tougher, I believe, on most other working people, whether they are digging ditches, teaching school kids, or staring into people’s mouths all day. And when I look at referee reports, many of the really harsh ones come from younger colleagues. We all need to do better and be more understanding when evaluating each others’ work.

**CC:** The story of your unpublished book is fascinating: please tell us more.

**IHW:** Unplugged and unpublished. The book, as the title implies, is about the teaching of computer science without using computers, in contrast with IT skills such as the use of Microsoft products, which is what many schoolchildren have experienced up until recently (and they think university computer science will be about advanced word processing). So it’s intentionally revolutionary, or at least runs counter to the established culture in the teaching world. The publishers’ responses were hilarious: hilarious, that is, if you hadn’t invested a large chunk of your life in what they were rejecting. One wrote that they “will not pursue the idea of publishing the book”, yet described it as “your wonderful volume ...” and said it “would be a real pity not to have this book released”. A children’s publisher said it “may be too academic for children”, while an academic publisher referred us to a children’s publisher. One publishers’ educational arm referred it to their computing department, who responded that they couldn’t publish a book if it wasn’t about how to do things on a computer. Yet now, if you google “unplugged”,

30
we come in above Eric Clapton!

CC: What does it mean “unplugged”?

IHW: You know, stop fooling around, unplug your computer, and start to learn some real computer *science*. As you know, probably better than anyone else, there are fundamental ideas about computation that do not depend on computers at all. *Computer Science Unplugged* has games and activities that teach kids about computation but do not involve computers. My favourite is a kind of formation dance that expresses a parallel sorting algorithm. Kids follow lines chalked on the playground that represent a sorting network and end up sorting $N$ numbers in $O(N)$ time. The [http://csunplugged.org/](http://csunplugged.org/) website has a video of 21 kids sorting 21 numbers in...um...about 7 seconds (admittedly the video is played in fast-forward mode).

CC: I have rushed to [http://csunplugged.org/](http://csunplugged.org/) to download it as I will soon become grandfather...

IHW: Congratulations! The activities are suitable for kids of all ages, 8–80. You’ll have to wait a while to try them on your grandchildren, but you might be able to start with your parents!

CC: Recently you have been appointed Chief Scientific Advisor at Pingar.

IHW: Yes. Pingar is a small NZ company that is developing really interesting technology for document analysis and organisation. My ex-student Alyona Medelyan is their Chief Research Officer and it’s great to keep in touch and involved with what they’re doing.

CC: You play jazz...

IHW: Now you’re talking. Yes. Live music is the really big thing in life, far more important than computers and technology. A couple of weeks ago at the SPIRE conference in Cartagena, Colombia, I had the great pleasure of jamming with a duo from Spain in their open-air concert and in the bar afterwards into the wee small hours of the morning. But I play classical music as well as jazz. I play second clarinet in the Trust Waikato Symphony Orchestra—this weekend we have a concert called the Waikato Proms, modelled after the famous BBC promenade concerts in London. And a clarinet group meets at my house every week. Often it’s quartets or quintets, but last night it was trios—an excellent evening sight-reading 19th and 20th Century music; a real challenge. I’m lucky enough to play with musicians who are better than me, so it’s a constant learning experience. We play everything: classical, modern, light, jazz, and they say I have the most comprehensive library of clarinet ensemble music in the country. Currently, having spent two months in Buenos Aires last year, I’m obsessed with Piazzolla tangos.
CC: Your yacht Beulah is a 28 foot Nova, New Zealand designed, launched in the early 80s, built of double diagonal kauri wood, fibreglassed over . . .

IHW: Ah yes. Sailing is my other passion, and part of the reason for moving to New Zealand from Calgary where the sea is a couple of day’s drive away. I began sailing as a kid. Indeed I once raced internationally, picked to represent Northern Ireland against the South in a youth championships. We lost. And I was never chosen again. Moving quickly on, NZ’s Hauraki Gulf is the best water in the world for the kind of weekend cruising that I enjoy now; hundreds of beautiful islands with lovely anchorages. Beulah, my pride and joy, is where I get away from it all: no computer, no internet, no phone even—and we rarely use the engine. There’s hardly any electricity, but of course there’s live music wherever you go in Beulah. The simple life. We sail and swim and play with dolphins; watch the sun set, drink wine, eat well, and sleep. And we have adventures. You can read about them in our annual family Christmas letters, which Google will find for you if you ask it nicely.

CC: Many thanks.

IHW: My pleasure. I could go on for hours. It’s always nice to talk about oneself.
Mihai Pătrașcu: Obituary and Open Problems

Mikkel Thorup

Figure 1: Mihai lived his short life in the fast lane. Here with his second wife Mirabela Bodic that he married at age 25.

Mihai Pătrașcu, aged 29, passed away on Tuesday June 5, 2012, after a 1.5 year battle with brain cancer. Mihai’s academic career was short but explosive, full of rich and beautiful ideas as witnessed, e.g., in his 20 STOC/FOCS papers. His many interesting papers are available online at: http://people.csail.mit.edu/mip/papers/index.html.
Mihai’s talent showed early. In high school he received numerous medals at national (Romanian) and international olympiads including prizes in informatics, physics and applied math. He received gold medals at the International Olympiad in Informatics (IOI) in both 2000 and 2001. He remained involved with olympiads and was elected member of the International Scientific Committee for the International Olympiad of Informatics since 2010.

Mihai’s main research area was data structure lower bounds. In data structures we try to understand how we can efficiently represent, access, and update information. Mihai revolutionized and revitalized the lower bound side, in many cases matching known upper bounds. The lower bounds were proved in the powerful cell-probe model that only charges for memory access, hence which captures both RAM and external memory. Already in 2004 [17], as a second year undergraduate student, with his supervisor Erik Demaine as non-alphabetic second author, he broke the $\Omega(\log n / \log \log n)$ lower bound barrier that had impeded dynamic lower bounds since 1989 [6], and showed the first logarithmic lower bound by an elegant short proof, a true combinatorial gem. The important conclusion was that binary search trees are optimal algorithms for the textbook problem of maintaining prefix sums in a dynamic array. They also proved an $\Omega(\log n)$ lower bound for dynamic trees, matching Sleator and Tarjan’s upper bound from 1983 [20]. In 2005 he received from the Computing Research Association (CRA) the Outstanding Undergraduate Award for best undergraduate research in the US and Canada.

I was myself lucky enough to meet Mihai in 2004, starting one of most intense collaborations I have experienced in my career. It took us almost two years to find the first separation between near-linear and polynomial space in data structures [19]. What kept us going on this hard problem was that we always had lots of fun on the side: playing squash, going on long hikes, and having beers celebrating every potentially useful idea we found on the way. A strong friendship was formed.

Mihai published more than 10 papers while pursuing his undergraduate studies at MIT from 2002 to 2006. Nevertheless he finished with a perfect 5.0/5.0 GPA. Over the next 2 years, he did his PhD at MIT. His thesis “Lower Bound Techniques for Data Structures” [11] is a must-read for researchers who want to get into data structure lower bounds.

During Mihai’s PhD, I got to be his mentor at AT&T, and in 2009, after a year as Raviv Postdoctoral Fellow at IBM Almaden, he joined me at AT&T. We continued our work on lower bounds, but I also managed to get him interested in hashing which is of immense importance to real computing. We sought schemes that were both truly practical and theoretically powerful [15].

With his amazing energy and creative spirit, Mihai continued his work with many different collaborators on diverse topics. Data structure lower bounds, however, remained his core research area. In one of his favorite papers “Unifying the
landscape of cell-probe lower bounds” [14], he identified a whole new level of structure and connectivity in the field.

On January 1, 2011, Mihai was diagnosed with brain cancer. After a partially successful operation on March 21, 2011, anti-seizure medicine took away most of his energy and clarity, but he still had an amazing intuition. He wanted to work till the very end, even though this meant I had to push him to work in a wheelchair the last 4 months. Less than a month before his passing, he was notified that he was co-winner of the 2012 EATCS Presburger Young Scientist Award, recognizing his huge contribution to the field.

1 Open Problems

In the last weeks before Mihai passed away, we talked about what were the important challenges in data structures. Our experience with data structure lower bounds has been that the strongest lower bounds have not been for abstract problems, but rather for concrete well-known problems. The list below contains concrete problems that we believe capture types of lower bounds that have so far never been proved. They are ordered with the ones I believe to be the most significant first.

Problem 1: Deterministic dictionaries

One of the most fundamental data structures is a dictionary which allows us to store and look up information associated with keys. Dictionaries are often identified with hash tables, which are the most common way they are implemented, but they could be implemented with binary search trees, supporting each operation in $O(\log n)$. Dictionary operations are a bottleneck for many kinds of data analysis, including the processing of high volume data streams. They are also in the inner loops of many algorithms and have been central to computing as long as we have had computers.

For more than 50 years, we have had good randomized solutions. The space is linear in the number of stored keys, and we support both updates and look-ups in constant expected time [4]. Lots of solutions are known pushing the randomness around, e.g., in a breakthrough, Fredman, Komlos, and Szemerédi [5] proved that there are dictionaries with constant deterministic look-up time. Dietzfelbinger et al. [2] showed that this deterministic constant look-up time can be maintained with randomized updates in constant expected time. The randomization implies a fundamental unreliability. We have to be prepared for some updates being slow, which is a problem for time-critical systems. The probability of such events can be reduced at the expense of longer average update times. The big open question is
Does there exist a perfect deterministic dictionary using linear space while supporting both lookups and updates in worst-case constant time?

Apart from $\text{RP}^2 = \text{P}$, we thought of this as the most important derandomization problem left in theoretical computer science. The current positive deterministic results are quite far away. If we want constant query time, the best known update time is $O(n^\varepsilon)$ where $\varepsilon$ is any positive constant [7]. If we are willing to settle for doubly-logarithmic query time, the update time can be improved to logarithmic [9]. If we want a joint bound for both lookups and updates, the best known bound is $O(\sqrt{\log n / \log \log n})$ [1]. The dream is to get down to constant time for both lookups and updates. Even with amortization, this would be a major breakthrough.

We believe that the true answer is negative, but how can we prove it? Often it is much easier to prove deterministic lower bounds than randomized lower bounds, but this does not give a separation unless the deterministic lower bounds are higher than the known randomized upper bounds. A separation for dictionaries would be extremely interesting, and likely have ramifications for many other problems.

**Problem 2: Multiphase problem**

In data structures, the largest proven lower bounds are polylogarithmic [8], and a major challenge is to prove polynomial lower bounds like $n^{\Omega(1)}$. Mihai proposed a very interesting line of attack via his so-called multiphase problem in [13].

**Problem 3: Set Intersection**

In [19], we proved a separation between near-linear and polynomial space, e.g., showing that certain queries that can be supported in constant time with $n^{1.001}$ space, require $\Omega(\log \log n)$ time with $n \log^{O(1)} n$ space. However, no such separation is known between space $n^{1.001}$ and space $n^{100}$. In [18], together with Roditty, we conjectured a concrete hard problem for space $n^\alpha$ for any $\alpha \in (1, 2]$. The hardness is all based on set intersection:

**instance for preprocessing:** The construction algorithm receives the $n$ sets $S_1, \ldots, S_n \subseteq [u]$. In a regular instance, for some set size parameter $s \leq u$, each set has size at most $s$, and each element appears in $ns/u$ sets.

**query:** for given $(i, j) \in [n]^2$, the boolean query is whether $S_i$ intersects $S_j$.

The two obvious solutions are to either store all the (positive) answers in the preprocessing phase, or to simply store the sets directly and intersect them during the query. A popular belief, consistent with all current upper bound ideas, is
that in general there is no smooth trade-off between these two extreme types of solutions. In fact, the problem seems hard even for random instances where each $S_i$ is a random subset of $[u]$ each of polylogarithmic size. Then the expected number of intersections is $\tilde{O}(n^2/u)$ and they can be represented in a hash table of this size. The following conjecture states that this is the best possible.

**Conjecture 1.** Let $a$ and $b$ be sufficiently large constants. Consider regular set intersection instances with $n$ sets, universe size $u$, and set size $s = \log^a n$. If a data structure with constant query time uses only $O(n^2/(u \log^b n))$ space and makes no false negatives, then for some set intersection instance, the fraction of false positives is $\Omega(1)$ over all $\binom{n}{2}$ possible queries.

In [18] we used this conjectured hardness to prove hardness of distance oracles for graphs. Proving the conjecture would likely lead to lower bounds for many other problems.

**Problem 4: Dynamic 2D Range Counting**

In the 2D range counting problem, we have points in 2D. We want to query the number of points in a rectangle specified by the 4 corner coordinates. In a decision version, we may just ask if the parity is odd. In [10] Mihai proved that for the static problem where $n$ points have to be represented with near-linear space, the query time is $\tilde{\Omega}(\log n)$. He believed that in the dynamic case, the update time would be $\tilde{\Omega}(\log^2 n)$. This would be the first superlogarithmic lower bound for a decision problem.

We note that Larsen [8] provided the $\tilde{\Omega}(\log^2 n)$ lower bound for the case where each point has an $O(\log n)$-bit weight, and where the query is about the range sum (not just the parity). However, such large weights can code more information than a memory address, and this is exploited heavily by the technique from [8].

**Problem 5: Succinct Dictionary**

In succinct data structures, we try to represent data using space close to the entropy $H$, yet provide efficient access. In his FOCS’08 best student paper [12], Mihai showed that for many data structure problems, we can pick an arbitrary parameter $t$, and represent the data using space $H + O(H/\log^t H)$ space, answering queries in time $O(t)$. For most of the problems, no such result was known for any $t > 2$. For some of the problems, he later presented matching lower bounds together with Viola [16]. One problem for which he could not find a matching lower bound was the dictionary problem, where we are given $n$ elements from $[u]$. The entropy is $H = \log_2 \binom{n}{u}$. The question is if we can do better than the $H + O(H/\log^t H)$ space
from [12]? Inspired by our paper [3], Mihai was hopeful that much better bounds would be possible. In [3], for the so-called trits problem, we improved the space from $H + O(H/\log H)$ to $H + O(1)$ while maintaining constant query time.

References


On December 13, 2012, our colleague Jacobus Willem (Jaco) de Bakker, member of the Section Informatics of the Academia Europaea since 1990, passed away surrounded by his family in his home in Amsterdam after a short illness. He is survived by his wife Angeline, his children Bas, Jaska, Catrien, Jacob and Lisa, and two grandchildren.

Jaco was born on March 7, 1939, in Ede, the Netherlands. He was for more than 38 years, from 1964 until 2002, connected as Head of the Computer Science Department to the Mathematical Centre, later called CWI (Centrum Wiskunde & Informatica) in Amsterdam. He was a Fellow of CWI since 2002. In 1973 he was appointed as Professor in Computer Science, in particular for the mathematical semantics of programming languages and reasoning on program correctness,
at the VU University Amsterdam, at that time called Vrije Universiteit Amsterdam. He occupied this professorship until his emeritate in 2002. In 1989 he was appointed as a member of the Royal Netherlands Academy of Arts and Sciences (KNAW), in the Section Mathematics. In 1972, Jaco was one of the founding fathers of the EATCS, the European Association for Theoretical Computer Science; he was Vice-President of the EATCS from 1972 until 1982, and Member of the Board until 1988. Since 1998 he was honorary member of IFIP Working Group 2.2, Formal Description of Programming Concepts. In 2002, during his retirement symposium at CWI, he received the Royal Decoration Knight of the Order of the Lion of the Netherlands (Ridder in de Orde van de Nederlandse Leeuw).

Jaco de Bakker started his scientific career with his Ph.-D. thesis in 1967 at the University of Amsterdam, with promotor Adriaan (Aad) van Wijngaarden, entitled: Formal Definition of Programming Languages: with an Application to the Definition of ALGOL 60. Jaco de Bakker was world-wide known and recognized for his pioneering work in developing the denotational and operational semantics of many basic features in programming languages, in a precise and rigorous mathematical style. One of its highlights became known as the induction rule of De Bakker and Scott. This culminated in his book Mathematical Theory of Program Correctness (1980). Later on, in the early eighties, he turned to the theory of communicating processes, introduced by Hoare and Milner, a theory known in those days as "concurrency". His initial investigations in this field were in cooperation with Jeffery Zucker. The basic features in this theoretical area were treated in the same mathematically rigorous style in his book Control Flow Semantics (1996) together with Erik de Vink. Apart from these books, he wrote more than 150 scientific articles.

In the Netherlands Jaco de Bakker was the originator of an extensive school of theoretical computer scientists. He supervised many Ph.D.-theses, and was the driving force in the eighties, together with Willem-Paul de Roever and Grzegorz Rozenberg, behind several nation-wide programmes for research and education in the Netherlands, such as REX (Research and Education in Concurrent Systems). REX lasted from 1988 to 1993; it was preceded by LPC (Landelijk Project Concurrency, National Project Concurrency) from 1984-1988. Prior to these programmes Jaco was Director, together with Jan van Leeuwen, of the 'Advanced Course on Foundations of Computer Science', a biennial series of influential courses with international attendance, from 1974 to 1982, held in Amsterdam. Jaco was also one of the founding fathers in 1979 of the Dutch Association for Theoretical Computer Science (WTI, Werkgemeenschap Theoretische Informatica), since 1995 called NVTI (Nederlandse Vereniging voor Theoretische Informatica). Jaco was Chairman of the WTI from 1979 until 1987. Jaco was proud of the fact that 32 scientists who at some time worked in his group were eventually appointed full professor.
Also in the eighties, Jaco was instrumental in stimulating the involvement and participation of the Dutch research community in the big European computer science frameworks such as FAST, Meteor, ESPRIT (European Programme for Research in Information Technology) and BRA (Basic Research Actions). As Head of the CWI Department Software Engineering he stimulated intensive contacts with the European research community, resulting in a lively and productive research atmosphere in which researchers of many nationalities cooperated on a regular basis.

In addition to playing a crucial role in education and research in theoretical computer science, Jaco de Bakker was also a gifted and respected science director and administrator. He influenced the lives of many of us. We all remember him as a great scientist and an amiable person. Moreover many computer scientists will remember him as a friend.

Jan Bergstra, Jan Willem Klop and Jan Rutten
Mobile Agents in Distributed Computing: Network Exploration

Shantanu Das
LIF, Aix-Marseille University & CNRS, France
sdas@bitvalve.org

Abstract

One of the recent paradigms in networked distributed computing is the use of mobile agents. Mobile agents are software robots that can autonomously migrate from node to node within a network. Although mobile agents can be easily implemented over a message passing network, they provide an abstraction for designing algorithms in a non-traditional way which can be
quite natural for certain problems, such as searching, monitoring or intruder
detection. A principle sub-task in most algorithms for mobile agents is the
traversal of the network. We focus on this problem of exploring an initially
unknown network with one or more mobile agents. We also consider the
related problem of constructing a map of the environment being explored by
the mobile agents.

1 Introduction

Mobile agents are software robots (mobile code) that can move autonomously
from node to node within a network, executing its operations and interacting with
the host environment at each node that it visits. Such software robots (sometimes
called bots, or agents) are already prevalent in the Internet, and are used for perform-
ing a variety of tasks such as collecting information, negotiating a business
deal, or for online shopping. When the data needed for a computation is phys-
ically dispersed, it can be sometimes beneficial to move the computation to the
data, instead of moving all the data to the node performing the computation. The
paradigm of mobile agent computing is based on this idea.

The use of mobile agents has been advocated for various reasons such robust-
ness against disruptions in network connectivity, improving the network latency
and reducing network load, providing more autonomy, and so on (see e.g. [26]). In
the context of distributed computing, the use of mobile agents has been suggested
by Fukuda et al. [22] as early as the 90’s and some algorithms for message-passing
networks that use token circulation (notably [25]) can be regarded as early exam-
pies of mobile agent algorithms in this area.

On the practical side, one of the major concerns with mobile agents has been
ensuring the agents are safe from tampering by potentially malicious host nodes
and on the other hand the host computer are not harmed by malicious agents[33,
28]. One typical example of a malicious agent is a computer virus that propa-
gates itself through the network. As there could be malicious agents, we could
use good agents to track down and eliminate the harmful agents. This gave rise to
several cops-and-robber games on networks, where a team of “good” agents move
through the network searching for the malicious agent (the robber) and the latter
tries to evade the good agents (the cops). This is only one of several problems
involving mobile agents that has recently captured the interest of the distributed
computing community. Other problems that have studied in the context of dis-
tributed computing are:

• The Rendezvous problem : Gathering multiple mobile agents at a single
  location.
• **Black hole search**: Locating harmful nodes in the network.

• **Distributed treasure-hunt**: Locating a resource available at an unknown node of the network.

• **Map construction**: Building a map of an unknown network.

• **Distributed verification**: Verifying some global property of the network.

Some of these areas are already too vast to cover in one combined survey. The rendezvous problem has been discussed in the recent surveys [30] and [9] as well as the book [2], while the black hole search problem has been studied by Markou [27] in the previous edition of this column.

In this article we would focus on the exploration problem which is usually a basic subtask useful for solving the other problems. The algorithm used for traversal of a network usually has a significant impact on both the solvability and the cost of solving these other problems. We would discuss efficient techniques for traversing an unknown network and for building a map of the network. The article is structured as follows. Section 2 explains the mobile agent model in detail. The following section presents exploration techniques for a single mobile agent. We consider both exploration of labelled as well as unlabelled networks. Section 4 discusses the special case of an agent having limited memory. Finally, Section 5 considers the collaborative exploration of networks by multiple mobile agents.

## 2 The Mobile Agent Model

In the mobile agent model, the network or the environment is modelled as an undirected¹ connected graph \( G = (V(G), E(G)) \). The computation is performed by a set \( Q \) of mobile entities called agents. An agent is an automaton that starts at some vertex of \( G \), in some given state \( s_0 \). Each agent has a finite number of states and the size of the state-space \( S \) depends on the amount of memory available to an agent (which may or may not depend on the size of the network). An agent with \( b \) bits of memory is assumed to have \( 2^b \) states. The initial placement of the agents is denoted by the function \( p : Q \rightarrow V(G) \). For any \( a \in Q \), \( p(a) \) is called the **homebase** of agent \( a \).

At any step of the algorithm, a mobile agent located at a node \( v \) of the network, may (1) detect the presence of other agents at the node \( v \) and communicate with them, and (2) perform any computation at node \( v \), using the information available at node \( v \) and its own state information, and (3) change its state and either decide

---

¹This article considers only the undirected environment although there exists results for directed graphs too.
to stay at node $v$ or move to an adjacent node. When performing computations at a node $v$, the agent has access to the (possibly unbounded) memory and computational power of the node $v$. When moving from a node $v$ to an adjacent node $u$, the agent is allowed to carry only its state information.

In order to enable navigation of the agents in the graph, at each node $v \in V(G)$, the edges incident to $v$ are distinguishable to any agent $a$ at node $v$. In other words, there is a bijective function

$$\delta_v : \{(v, u) \in E(G) : u \in V(G)\} \rightarrow \{1, 2, \ldots d(v)\}$$

which assigns unique labels to the edges incident at node $v$ (where $d(v)$ is the degree of $v$). The function $\delta = \{\delta_v : v \in V(G)\}$ is called the local orientation or port-numbering.$^2$

We consider three different models for communication between mobile agents:

1. Face-to-Face: In this model, two agents can exchange information only when they are both located in the same node.
2. Pebble Model: In this case, the agent is allowed to leave a pebble at its current location $v$. This pebble is visible to (and can be picked up by) any agent that subsequently visits node $v$.
3. Whiteboard Model: In this case, an agent may write any information at public whiteboards at its current node $v$. This information is visible to (and can be modified by) any agent that subsequently visits node $v$.

In some cases, the vertices of $G$ may be initially labelled over the set of symbols $L$ by $\lambda: V(G) \rightarrow L$ which is the labelling function. When this labelling is injective, we say that the nodes have distinct identifiers. On the other hand, when all nodes have the same label $c \in L$ we say that the network is anonymous.

The environment is thus, represented by the tuple $(G, \lambda, Q, p, \delta)$. In case the nodes of the graph are anonymous, we shall omit $\lambda$. For the rest of this paper, $n = |V(G)|$ and $m = |E(G)|$ respectively denotes the numbers of vertices and of edges of $G$, while $k = |Q|$ denotes the number of agents and $\Delta$ denotes the maximum degree of a vertex in $G$. We shall use the words vertex and node as well as the words graph and network interchangeably.

Since we consider asynchronous networks, the cost of an algorithm is measured in terms of the number of moves performed by the agents where each move corresponds to the traversal of a single edge by a single agent. In case of single agent exploration, the moves cost is same as the time complexity, if we assume each move to take the same time.

Finally we remark here that the mobile agent model as explained above can be easily implemented on message-passing networks. In fact, there exist simulation techniques for importing algorithms from mobile agent systems to message-passing systems and vice versa [10, 15].

$^2$The labels on the edges may correspond to port numbers on a network.
3 Network Exploration by a Mobile Agent

The problem of exploring an unknown environment has been studied extensively starting from the work of Shannon [34] who first considered the problem for a finite state automaton moving in a maze. One of the questions that has been studied is whether finite state automata can explore graphs of arbitrary size. This question was answered by Rollik [32] who showed that any finite team of finite state automata cannot explore all graphs. Later Fraigniaud et al.[21] showed that a single agent needs $\Omega(\log n)$ bits for traversing arbitrary graphs of size $n$ and maximum degree at least three. We consider the problem of exploration with memory restrictions in Section 4.

Depending on the objective of exploration, an algorithm may terminate before visiting all nodes, or only after visiting each node at least once, or it may not stop and continue indefinitely. The latter scenario is called perpetual exploration and is sometimes useful e.g. for network monitoring. In the perpetual exploration problem [21], the agent is required to periodically visit each node and optimizing the period of the traversal may be one of the objectives. On the other hand, for the treasure-hunt problem [35], the agent can stop as soon as it reaches a node containing the “treasure” (i.e. the resource being searched).

In some cases visiting all edges (not just the nodes) may be necessary (e.g. for the edge-search problem). In some other cases, the agent may be required to build a map of the environment (such a map enables the agent to find a shorter traversal path for subsequent traversals). For any agent traversing a graph, a map of the network is a copy of the graph $G$ where the edges are labelled with port-numbers and the node where the agent is located is specifically marked. Access to a map allows the agent to traverse the network faster and also to perform certain computations without the need to traverse the network. Thus, a map construction algorithm is a useful primitive in mobile agent computing.

3.1 Exploration of Labelled Environments

When the nodes of the graph are labelled with distinct identifiers, exploration can be achieved by the conventional depth-first search (DFS) or breadth-first search (BFS) algorithms. In fact, the depth-first traversal is asymptotically optimal in terms of number of moves because it requires $\Theta(m)$ edge traversals. A modified version of the algorithm has been proposed by Panaite and Pelc [29] which makes $m + O(n)$ moves. In this case, the penalty (i.e. the number of additional edges traversed) is linear in the numbers of nodes (and not the number of edges). The algorithm achieves this by using some clever rules for backtracking that avoids traversing too many of the already explored edges.

Another version of the problem that has been studied by Awerbuch et al. [3]
is called *piecemeal* exploration where the agent has to periodically return to its homebase (e.g. for refuelling), during the exploration. For this version of the problem, the standard DFS algorithm is not a feasible algorithm; During DFS, an agent may make $\Omega(n)$ moves between two subsequent visits to the homebase, even if the diameter of the graph is very small. The BFS algorithm can be used to solve the problem of piecemeal exploration, but it could require $O(m^2)$ moves in the worst case. Instead, the authors use a combination of DFS and BFS, where BFS is performed locally within small regions, called *strips*, while DFS is performed on a higher level spanning tree that connects these strips together. This algorithm requires $O(m + n^{1+\epsilon})$ moves. This was later improved upon by Awerbuch and Kobourov [4] who gave a recursive piecemeal exploration algorithm that requires $O(m + n \log^2 n)$ moves. Eventually Duncan et al. [19] gave an asymptotically optimal algorithm for performing piecemeal exploration in $O(m)$ moves.

Note that all the above traversal methods require the agent to have sufficient memory to remember the identifiers of the visited vertices. Thus the agent must have at least $\Omega(n \log n)$ bits of memory. For agents with smaller memory, more involved techniques may be required (c.f. Section 4), thus increasing the time complexity of exploration. On the other hand, an agent having sufficient memory can also construct the map of the graph using the information obtained during a traversal of the graph. Thus, in this case, the problem of map construction is not different from the task of simply traversing all edges of the graph in an organized manner. When the nodes of the network are anonymous (not labelled with unique identifiers), then the task of map construction could be more difficult than simply performing a traversal of the network. This scenario is considered in the next two sections.

### 3.2 Exploration of Anonymous Networks

In an anonymous network, the nodes are not labelled with identifiers and thus the agent may not be able to distinguish between any two nodes which have the same degree. Note the edges of the graph are still locally oriented with port-numbers since otherwise exploration is not possible (as explained before).

Traversal of an anonymous graph can always be achieved by a breadth-first traversal of all paths up to a depth of $D$, the diameter of the graph (if the value of $D$ is known), or up to depth $n$ or, any other upper bound on the diameter. Note that the knowledge of the size $n$ or the diameter $D$ of the graph is necessary only for terminating the traversal (one could in principle continue infinitely and thus ensure that all nodes and edges of the graph will be visited within a finite time). If the agent knows the value of $n$, it could perform a depth-first traversal of the tree containing all paths of length $n$ from the starting vertex. This incurs a cost of $\Omega(\Delta^n)$ moves in graphs of maximum degree $\Delta$. It is possible to have
a more efficient traversal of the graph using the concept of universal exploration sequences.

A universal traversal sequence for a graph $G$ is a list of port numbers, such that, if an agent starting in any node of $G$ chooses to move according to the specified port numbers, it will eventually visit all nodes of $G$. For our purpose, a more useful notion is that of a Universal Exploration Sequence (UXS) defined by Koucký [24] as follows. For any node $u \in G$, the $i$th successor of $u$, denoted by $\text{succ}(u, i)$ is the node $v$ reached by taking port number $i$ from node $u$ (where $0 \leq i < d(u)$). Let $(a_1, a_2, \ldots, a_t)$ be a sequence of integers. An application of this sequence to a graph $G$ at node $u$ is the sequence of nodes $(u_0, \ldots, u_{t+1})$ obtained as follows: $u_0 = u$, $u_1 = \text{succ}(u_0, 0)$; for any $1 \leq i \leq t$, $u_{i+1} = \text{succ}(u_i, (p + a_i) \mod d(u_i))$, where $p$ is the port-number at $u_i$ corresponding to the edge $\{u_{i-1}, u_i\}$. A sequence $(a_1, a_2, \ldots, a_t)$ whose application to a graph $G$ at any node $u$ contains all nodes of this graph is called a UXS for this graph. A UXS for a class of graphs is a UXS for all graphs in this class. An important result shown in [1] is the following.

**Property 3.1.** For any positive integers $n, \Delta$, $\Delta < n$, there exists a UXS of length $O(n^3\Delta^2 \log n)$ for the family of all graphs with at most $n$ nodes and maximum degree at most $\Delta$.

It is also known that such universal sequences are easy to construct and thus, using the above property a mobile agent can traverse any graph $G$ of size $n$, incurring a cost polynomial in $n$, which compares favorably with the exponential cost of the brute-force algorithm described earlier. However, it is not known whether there is a more efficient way of traversing arbitrary unknown graphs other than using universal exploration sequences. Thus there is a big gap in the cost of exploring anonymous networks compared to networks with uniquely labelled nodes.

If we consider the pebble model of communication where an agent is allowed to place a pebble on a node to mark it, then it is possible to efficiently traverse an anonymous graph $G$ (as well as build a map of $G$). The agent can start with an initial map containing only its current node and start exploring new edges and adding them to the map. Whenever the agent traverses an edge $e = (v, u)$ and arrives at an unknown vertex $u$, it can place the pebble on $u$, return back to the previous vertex $v$ and perform a traversal of the known part of $G$ (that is included in the current map). If the agent encounters the pebble during this traversal, it can identify the vertex $u$ in its map. Otherwise, the node $u$ is an unexplored node and the agent can add this node and the edge $e$ to its map. In both cases, the agent has extended its map by a single edge and it can now pick up the pebble from node $u$ and continue the exploration. Using this process a mobile agent with a single pebble can explore and map an anonymous graph in $O(nm)$ (i.e. $O(n^2\Delta)$) moves. Moreover the agent does not need any prior knowledge of the size $n$ of the network or even an upper bound on $n$. 

60
As an aside, we remark here that exploration of (strongly connected) directed graphs using a pebble has also studied \[6\] and the best known algorithm for mapping a digraph using a single pebble has a cost of \(O(n^8\Delta^2)\). Thus it seems that exploring digraphs is much more difficult than exploring undirected graphs and there exists a large gap between the costs of exploration in these two cases. Moreover, when the agent does not know any upper bound on the size of the network, a single pebble is not sufficient for exploring digraphs and at least \(\Omega(\log \log n)\) pebbles are necessary, as shown in the above paper.

### 3.3 Map Construction in Anonymous Networks

Although any arbitrary graph can be traversed by an agent having sufficient memory, the problem of map construction cannot be solved in all graphs. There exists graphs which are not recognizable, i.e. an agent traversing such a graph can not build a map of the graph, even if it is traverses every edge of the graph and even if it has an unbounded amount of memory allowing it to remember everything that it sees. This impossibility comes from the existence of symmetries in certain graphs. Consider for example the two graphs shown in Figure 1. The two graphs are distinct (non-isomorphic) but an agent traversing the first graph sees exactly the same as an agent traversing the second graph. Thus, both these graphs are non-recognizable. There exists a characterization of the family of graphs that are not recognizable, given by Yamashita and Kameda \[36\]. This can be explained in terms of the concept of graph coverings \[7\]. For any two graphs \(G\) and \(H\) that are indistinguishable by an exploring agent, there exists a common base graph \(B\) which is the smallest multi-graph that cannot be distinguished from either \(G\) or \(H\). We say that graph \(G\) covers graph \(B\). Map construction can be solved in a graph \(G\) only if it is covering-minimal, i.e. there exists no smaller graph \(B\) that is covered by \(G\).
Algorithm 1: Map Construction from a uniquely marked node \( r \)

\[
\begin{align*}
Map & := T := \{ r \}; \\
\text{Add } r \text{ to Queue;} \\
\text{ROOT_PATHS} & := \emptyset; \\
\text{while } \text{Queue is not empty do} & \\
\text{Get next node } v \text{ from Queue and go to } v \text{ using Map;} & \\
\text{while node } v \text{ has unexplored edges do} & \\
\text{Traverse the next unexplored edge } e = (v, u); & \\
\text{for each path } P \in \text{ROOT_PATHS do} & \\
\text{Apply label sequence } P \text{ at node } u; & \\
\text{if successfully reached a marked node then} & \\
\text{Add to } Map \text{ a cross-edge from } v \text{ to Start}(P); & \\
\text{Update the number of explored edges at the node Start}(P); & \\
\text{Return to node } v \text{ using } T \text{ and exit Loop;} & \\
\text{else} & \\
\text{Backtrack to node } u; & \\
\text{if All path sequences failed to reach a marked node then} & \\
\text{Add a new node } u \text{ to } T \text{ and } Map; & \\
\text{Add edge } (v, u) \text{ to } T \text{ and } Map; & \\
\text{Insert } u \text{ to Queue;} & \\
\text{ROOT_PATHS} & := \text{ROOT_PATHS} \cup \text{Path}_T(u, r); & \\
\text{Backtrack to node } v; & \\
\end{align*}
\]

The impossibility of mapping arbitrary anonymous graphs can be overcome when the mobile agent has some means of marking the nodes of the graph. As mentioned in the previous section, when the mobile agent is provided with a pebble, map construction can be solved in all connected graphs. A much weaker assumption, that the starting node of the agent is distinctly marked, is also a sufficient condition for mapping an arbitrary graph. An algorithm that achieves this was presented in [8] (see Algorithm 1 below). The idea of the algorithm is the following. Let the starting node (which is distinctly marked) be \( r \). The agent can perform a breadth-first traversal building a BFS-tree \( T \) rooted at \( r \). During the traversal, whenever the agent explores a new edge and reaches a node \( v \), the agent checks whether \( v \) has been visited before (i.e. whether node \( v \) is the same as some node \( u \) in its tree). This can be done by successively applying the label-sequences for the reverse paths from each node \( u \in T \) to the root \( r \), and checking if one of these paths hits the marked node. To this end, the algorithm maintains a data structure, called \text{ROOT_PATHS}, that stores the label-sequence for each path \( P \) in
$T$ going from any node $u \in T$ to the root $r$. For such a stored path $P$, $\text{Start}(P)$ refers to the node $u$. During the process of building the BFS-tree $T$, the algorithm also constructs a map that contains all the cross-edges discovered by the agent, in addition to the tree edges belonging to $T$. The algorithm completes the map construction in $O(n^3\Delta)$ moves.

### 4 Exploration with Small Memory

As mentioned before, a mobile agent requires $\Omega(\log n)$ bits of memory in order to explore all graphs of $n$ nodes. It was an open question for a long time whether exploration of arbitrary graphs can be performed by an agent having logarithmic memory, until Reingold [31] gave a positive answer. Note that an agent having $\Theta(\log n)$ can remember only a few node identifiers in a labelled network. Thus, the conventional DFS or BFS algorithms cannot be used for exploration. It is possible to use a UXS to perform the exploration, due to the results of Reingold [31] who gave a log-space constructible UXS. This however, requires the prior knowledge of an upper bound on $n$. When such knowledge is not available, we can use the following strategy for exploration. The agent has enough memory to remember the identifier of the starting location (which we call the source). The idea is that the agent guesses a value $N$ as an upper bound for $n$ and performs a traversal using UXS for graphs of size $N$. During this traversal whenever it reaches a vertex $v$, the agent performs a check operation and if this operation returns false, then the agent aborts the current traversal, doubles the value of $N$, and starts the whole procedure again using the new value of $N$ and the current vertex as source. The checking operation at a vertex $v$, simply visits each neighbor $w$ of $v$, remembers the identifier of $w$, returns to the source and performs another traversal to check if the traversal visits $w$. If not, then the checking operation returns false. Otherwise it continues to check the other neighbors of $v$ and finally return true. If the agent completes the traversal without having to increase $N$ anymore then this traversal has visited all nodes of $G$. So the agent can stop. Thus the algorithm performs exploration with stop, without the need for any prior knowledge of the network size.

In case of anonymous networks, some upper bound on the value of $n$ is always necessary for exploration, unless the network is a tree. For anonymous tree networks, Gasieniec et al.[23] gave an algorithm for an agent with $O(\log n)$ memory to explore the network and return to its starting location, without any prior knowledge of the network size. Earlier Diks et al. [17] showed that $\Omega(\log \log \log n)$ memory is necessary for tree exploration with stop and $\Omega(\log n)$ memory is required if the agent needs to return to the starting location. Note that in a tree network, exploration without stop can be performed even by an agent having no
memory, using the simple strategy of always leaving a node by the next port (in cyclic order) from the one through which it arrived. This is often called the right-hand-on-the-wall strategy.

The exploration of arbitrary graphs with agents having O(1) memory (i.e. finite state automata), has also been well investigated. In this case there must be some additional mechanism to help the agent perform its task of exploration. For example the nodes or edges of the network may be labelled in such a way as to allow the finite state agent to complete the exploration. Dobrev et al. [18] considered the problem of assigning port numbers to the edge of the graph in such a way that an agent with no memory performing a right-hand-on-the-wall walk can periodically visit all nodes of the graph. The objective is to obtain a small period of traversal and the above paper achieved a period of 10\(n\) for graphs of size \(n\). Eventually this was improved to a period of \((4 + 1/3)n − 4\) in a more recent paper by Czyzowicz et al. [12]. For an agent with \(O(1)\) bits of memory, the authors presented an algorithm and a corresponding port assignment that allows periodic traversal with a period of at most 3.5\(n\). In contrast, there exists a trivial lower bound of \(2^n − 2\) for the period of traversal by any agent (irrespective of memory size). For the case of the agent with no memory, a stronger lower bound of 2.8\(n\) was shown in the same paper. Thus there still exists a small gap between the best known lower and upper bounds for the problem.

5 Network Exploration by Multiple Agents

We have so far considered the exploration problem for a single agent. When there are multiple agents starting from the same node of the network, they may collaborate with each-other to explore the network collectively. Note that if the agents are all identical and follow the same deterministic algorithm, then all the agents would move together and we would not gain anything from having multiple agents instead of one. Thus, there must be some means of breaking symmetry between the agents. For example, if the agents have distinct names then each agent can follow a different path and thus explore different parts of the network concurrently.

Another possible scenario is when multiple agents start from distinct nodes of the network. In this case each agent may independently explore the network using the techniques discussed in the previous sections. However if the agents mark the nodes during the exploration then there needs to be some agreement between the agents so that an agent does not get confused when it encounters a node marked by some other agent. For example, Algorithm 1 presented in the previous section, assumes a uniquely marked homebase; this algorithm would fail when there are multiple agents starting from distinct nodes. The problem
of distributed exploration with multiple agents initially dispersed in the graph, is considered in Section 5.2.

5.1 Collaborative Exploration

The collaborative exploration of a network by a team of collocated mobile agents was studied by Fraigniaud et al. [20]. They considered tree networks where a team of \( k \) synchronous mobile agents labelled as 1, 2, \ldots \( k \) located at the root of the tree, need to explore all nodes and return to the starting node. The paper provided an algorithm that takes \( O(D + n/\log k) \) time for the exploration by \( k \) agents, thus providing an improvement by a factor of \( \log k \) over single agent exploration. The agents do not know the exact topology or, even the size of the network. The main idea of the algorithm is the following. At any stage of the algorithm, the agents available at node \( v \) are distributed among its subtrees in such a way that the number of agents in any two (unexplored) subtrees does not differ by more than one. Whenever a subtree has been explored completely all agents in that subtree move to its parent. The authors show how to implement this strategy in a distributed manner, where the communication between agents is achieved by reading and writing information on whiteboards available at each node.

Note that any algorithm for collaborative exploration by \( k \) agents must require \( \Omega(D + n/k) \) time. Thus there is an overhead of \( k/\log k \) in the above algorithm and it is not known whether this overhead can be reduced. A lower bound of \( 2 - 1/k \) was shown for the overhead of any collaborative exploration algorithm using \( k \) agents.

Collaborative exploration of arbitrary graphs has been considered in the context of the black hole search problem [11]. The objective, in this case, is to locate harmful nodes (called black holes) in the network and any agent arriving at such a node dies and cannot continue with the exploration. The above paper provided an exploration algorithm for \( k \) agents that takes \( O(n/k) \) time when the number of black holes is at most \( k/2 \) and \( k = O(\sqrt{n}) \). However this algorithm assumes that the network topology is known in advance by the mobile agents.

5.2 Distributed Exploration

When there are multiple mobile agents initially dispersed among the nodes of an anonymous network it is possible to collectively explore the network if the agents can communicate by leaving marks. In particular we will assume the whiteboard model of communication where agents can read and write information on public whiteboards at each node that the agent visits. If the agents have distinct identities, each agent can individually explore the complete network by performing a DFS marking each visited node with its identifier. On the other hand if the agents are
identical (i.e. no distinct names) then the marks made by an agent would not be distinguishable from those of another agent. In this case some cooperation between the agents seems necessary. A distributed exploration of the network can be performed by the following algorithm [13], which we call a distributed depth-first search (DDFS).

**Procedure DDFS:** An agent \( A \) starts from its homebase a depth-first search traversal marking the nodes that it visits (unless they are already marked) and labelling them with numbers 1, 2, 3, and so on. Each node marked by the agent and the edge used to reach it are added to a tree data structure stored in the memory of the agent. If the agent reaches an already marked node, it backtracks to the previous node and tries the other edges incident to the node. The agent stops when there are no unexplored edges incident to the nodes of its tree. The tree obtained at the end of the traversal is called the territory \( T_A \) of the agent.

It can be easily shown that the territories obtained by the agents in the above process, forms a spanning forest of the graph \( G \). Thus, each node of the agent is visited by some agent and the agents together complete the exploration of the graph. The exploration requires \( O(m) \) moves in total. If the agents are required to perform periodic traversals of the graph, then any subsequent traversal of \( G \) can be performed using only \((2n - 2k)\) moves in total (by restricting each agent to traverse its own territory).

The above technique was extended in [14] to perform map construction in those cases when \( G \) is covering-minimal (i.e. when it is possible to construct a map of \( G \)). The map construction algorithm proceeds in two phases. In the first phase, the agents simply perform procedure DDFS. At the end of this procedure there is exactly one agent in each tree in the forest and each agent \( a \) has a map of the tree \( T_a \) that it belongs to. The second phase of the algorithm is a competition between neighboring agents, during which each losing agent merges its territory with the corresponding winning agent. This process is repeated with the objective of eventually forming a single tree spanning the graph \( G \). As a final step, nodes of the tree are assigned unique labels and then all non-tree edges are added to obtain a complete map of \( G \). The main complication in the algorithm in the competition phase which proceeds in several rounds (at most \( k \) rounds when there are \( k \) agents). Overall, the algorithm has a cost of \( O(mk) \) moves in total.

### 6 Conclusions

We considered the exploration of unknown networks (graphs) by mobile agents that can autonomously move along the edges of the graph. Graph exploration is a basic subtask in most mobile agent algorithms and efficient techniques for
exploration can help to speed up the computation by mobile agents. We also studied techniques for constructing a map of an initially unknown network. Map construction algorithms help us to execute algorithms for known topologies in unknown environments (or, in dynamic environments where it is not possible to have a-priori knowledge of the network topology).

The scope of this article is restricted in many ways. We consider mainly the exploration of undirected graphs instead of the more difficult task of exploring strongly connected directed graphs. In fact, it is not always possible to explore arbitrary digraphs if the nodes are not uniquely labelled and there is no marking device. Secondly, the algorithms presented in this survey are for the fault-free scenario and it is possible to consider the exploration problem in the presence of various failures, such as node failures, agent crashes, and the failure of marking devices. Finally, this article excludes many interesting results on randomized algorithms for exploration e.g. based on random walks by mobile agents.

References


The Logic in Computer Science Column

by

Yuri Gurevich

Microsoft Research
One Microsoft Way, Redmond WA 98052, USA
gurevich@microsoft.com

Streams are Forever*

Jörg Endrullis  Dimitri Hendriks  Jan Willem Klop
VU University Amsterdam

Abstract

Just like diamonds, streams are forever. They are also ubiquitous, arising in theoretical computer science (formal languages and functional programming), mathematics (number theory), and engineering (signal processing). Also in business computing applications streams are important. Here one can think of the streams of data queries flowing into search engines, or streams of financial data processed by financial organizations.

In this paper, we describe a number of aspects of streams that we have encountered and studied during the last years.

1 Introduction

Infinite streams, also called infinite sequences, infinite words, or \( \omega \)-words, are the subject of study in several disciplines, ranging from the foundations of mathematics (choice sequences in intuitionistic logic), to engineering applications in signal

*This research has been funded by the Netherlands Organization for Scientific Research (NWO) under grant numbers 639.021.020 and 612.000.934.
processing. In this paper we will not be concerned with deep philosophical or logical aspects of infinite sequences. We will be interested mostly in specifications of streams, their classification, and their comparison.

A landmark was the work [54] of Axel Thue, who devised in 1906 infinite sequences of symbols avoiding certain simple patterns such as squares \(ww\) or cubes \(www\) where \(w\) is a finite word. He introduced the cubefree sequence \(M = 01101001100110100\cdots\), now known as the Thue-Morse sequence. This sequence turned out to be ubiquitous indeed, see [2], and was rediscovered by Marston Morse in 1921 in the mathematical context of dynamical systems and ergodic theory [45]. The Thue-Morse sequence is also known to be an automatic sequence (see [3]), and in particular it is a (purely) morphic sequence or D0L sequence. In the terminology of [48] the sequence is obtained by a ‘substitution’, another word for morphism. Later we will encounter several definitions of the Thue-Morse sequence, and also explain some of the key words just used denoting families of streams.

So infinite streams (we will often call them just streams) arise in theoretical computer science, in particular the areas of formal languages and combinatorics, and also in mathematics, with applications in dynamical systems and number theory. They also appear in the more practical side of computer science where functional programming languages reside (see e.g. [52]).

A word on methodology. Streams can be approached from various directions, originating from deep and established mathematical areas such as the theory of numeration systems (there have been several workshops devoted to this theme in recent years) concerned with methods to denote numbers, and also from relatively young developments in theoretical computer science, specifically infinitary term rewriting and coalgebra. The first is an outgrowth of the classical lambda calculus, while the second saw the light with the emergence of theories about infinite processes with communication, based on non-wellfounded set theory. Our point of view is that of infinitary term rewriting, to which we devote a nutshell introduction at the end of the paper.

Let us now give a short survey of the topics we aim to address. In Section 2 we start with presenting some of the well-known families of streams, exemplified by some famous streams. Thus we meet morphic sequences, automatic sequences, Toeplitz words, sturmian sequences, and sequences obtained by periodically iterated morphisms.

In Section 3 we briefly discuss the well-known relation between streams and fractals via turtle graphics. Even though it is well-known that some streams give rise via some turtles to some well-defined fractals, the complete correspondence between streams and fractals is not at all completely clear; at the end of the paper we devote a further question about such a correspondence.

The next point to visit in our guided tour is a discussion of possible ways
to compare streams as to their complexity, Section 4. Here we do not mean the well-known logical complexity in terms of the arithmetical or analytical hierarchy. But there are two notions of comparing complexity of streams that spring to mind, namely Kolmogorov complexity and subword complexity. We argue that both have their drawbacks, and propose a way of comparing streams that is simple but did not yet receive attention. This comparison is analogous to comparing the intrinsic difficulty of sets of natural numbers (which are just streams over \{0, 1\}), by means of Turing degrees. There two objects are equivalent iff they can be transformed back and forth into each other by Turing Machines. For the purpose of comparing streams, a similar procedure can be adopted, except that Turing Machines now are much too powerful; all interesting streams are computable, and thus equivalent with respect to transforming them into one another via Turing Machines. The appropriate transformation tool, generalizing several well-known transformation notions between streams, seems to be that of a finite state transducer (FST), a generalization of a Mealy machine. A Mealy machine transforms an input stream in a letter-to-letter fashion. FSTs are more general in that they transform letters to words. In this way we obtain an interesting hierarchy of degrees of streams, presenting us with a plethora of challenging questions.

After having discussed these degrees of streams, in Section 5 we turn to a specific question, which was raised by Larry Moss. It is concerned with a case study, namely how to decide the equivalence of streams that can be defined using very restricted means, only using the stream operator known as ‘zip’. The famous Thue-Morse stream can be defined this way. Solving this question (see [32]), provided an additional bonus; it yielded an alternative characterization of automatic sequences. Moreover, a slight generalisation of the zip specification format resulted in some quite challenging open questions, where we find ourselves on the sharp edge of the decidability property.

In Section 6 we consider infinite words generated by periodically iterated morphisms, and a question raised by Lepistö and Karhumäki, asking for the subword complexity of streams obtainable by iterating morphisms. The upshot is that this way of specifying streams yields maximally complex streams in that their subword complexity is exponential. The technique to establish this is interesting, using Conway’s intriguing programming language called Fractran of which a short exposition is given.

In Section 7 we discuss a problem that we have postponed somewhat, but that actually confronts us in the very beginning of studying streams, at least if we adopt the mind set of functional programming applications. This is the well-known problem of productivity, put on the map by Dijkstra, studied by his student Sijtsma [50], and approached forcefully by the functional programming community in the two seminal papers [33] and [52]. We outline the problem, and sketch progress made by the present authors.
We conclude our guided tour along streams with a sample of questions in Section 8.

2 Families of Streams

In the landscape of streams there are many families, characterised by shared principles or formats that are used in stream definitions or in the generation of streams. We give a brief listing of the main ones. An overview of some inclusion relations between these families is displayed in Figure 1.

![Figure 1: Some inclusions between stream families.](image)

We use standard terminology and notation; for example, see Allouche and Shallit [3] or Lothaire [43]. We use \( \mathbb{N} = \{0, 1, 2, \ldots \} \) for the set of natural numbers. Let \( \Sigma \) be a finite alphabet. We denote by \( \Sigma^* \) the set of all finite words over \( \Sigma \), and by \( \varepsilon \) the empty word. The set of infinite words over \( \Sigma \) is \( \Sigma^\omega = \{ x \mid x : \mathbb{N} \rightarrow \Sigma \} \). On the set of all words \( \Sigma^\infty = \Sigma^* \cup \Sigma^\omega \) we define the metric \( d \) for all \( u, v \in \Sigma^\infty \) by \( d(u, v) = 2^{-n} \), where \( n \) is the length of the longest common prefix of \( u \) and \( v \). For a word \( w \in \Sigma^\infty \) and \( n \in \mathbb{N} \), we write \( w(n) \) for the \( n \)-th letter of \( w \) (counting from zero). We write \( |x| \) for the length of a word \( x \in \Sigma^* \). We call a word \( v \in \Sigma^* \) a factor or subword of \( x \in \Sigma^\infty \) if \( x = uvy \) for some \( u \in \Sigma^* \) and \( y \in \Sigma^\infty \).

2.1 Morphic or CD0L Sequences

A widely studied family is that of the morphic sequences. Given alphabets \( \Sigma \) and \( \Gamma \), a morphism is a map \( h : \Sigma^* \rightarrow \Gamma^* \) such that \( h(\varepsilon) = \varepsilon \) and \( h(uv) = h(u)h(v) \) for all words \( u, v \in \Sigma^* \). For \( k \in \mathbb{N} \), a morphism \( h \) is called \( k \)-uniform if \( |h(a)| = k \) for all \( a \in \Sigma \); \( h \) is a coding if it is 1-uniform.

Let \( s \in \Sigma^* \), \( h : \Sigma^* \rightarrow \Sigma^* \) a morphism, and \( c : \Sigma \rightarrow \Sigma \) a coding. If in the metric space \( \langle \Sigma^\infty, d \rangle \) the limit \( h^\omega(s) = \lim_{t \rightarrow \infty} h^t(s) \) exists, then

(i) \( h^\omega(s) \) is called purely morphic or D0L word, and

(ii) \( c(h^\omega(s)) \) is called a morphic or CD0L word.

For example, the Thue-Morse sequence \( M \) is obtained by iterating the morphism \( h \) defined by \( h(0) = 01 \), \( h(1) = 10 \) on the starting word 0, i.e., \( M = h^\omega(0) \).
2.2 Periodically Iterated Morphisms

Instead of repeatedly applying a single morphism, one may alternate several morphisms from a given (finite) set in a periodic fashion. This gives rise to what are called PD0L sequences [15, 16, 42, 10], which form a generalization of D0L sequences.

Let \( H = \langle h_0, \ldots, h_{p-1} \rangle \) be a tuple of morphisms \( h_i : \Sigma^* \to \Sigma^* \). We define the map \( H : \Sigma^* \to \Sigma^* \) as follows:

\[
H(a_0 a_1 \cdots a_n) = u_0 u_1 \cdots u_n
\]

where \( u_i = h_k(a_i) \), with \( k \equiv i \pmod{p} \) and \( k \in \mathbb{N}_{<p} \).

For \( s \in \Sigma^* \), if in the metric space \( (\Sigma^\infty, d) \) the limit \( H^\omega(s) = \lim_{i \to \infty} H^i(s) \) exists, we call \( H^\omega(s) \) a PD0L word.

A famous example generated by such a procedure is the Kolakoski word [39]

\[
K = 122112122121121121221122121121
\]

which is defined such that \( K(0) = 1 \) and \( K(n) \) equals the length of the \( n \)-th run of \( K \); here by a ‘run’ we mean a maximal subsequence of consecutive identical symbols. The Kolakoski word can be generated by alternating two morphisms on the starting word 12, \( h_0 \) for the even positions and \( h_1 \) for the odd positions, defined as follows:

\[
h_0 : 1 \to 1, \quad 2 \to 11
\]

\[
h_1 : 1 \to 2, \quad 2 \to 22
\]

So for \( H = \langle h_0, h_1 \rangle \), we have \( K = H^\omega(12) \), and the first few iterations are

\[
\begin{align*}
H^0(12) &= 12, \\
H^1(12) &= h_0(1) h_1(2) = 122, \\
H^2(12) &= h_0(1) h_1(2) h_0(2) = 12211, \\
H^3(12) &= h_0(1) h_1(2) h_0(2) h_1(1) h_0(1) = 1221121
\end{align*}
\]

It is known that the Kolakoski word is not a D0L word [16], i.e., cannot be generated by iterating a single morphism. However it is an open problem whether it is a CD0L word, i.e., the image under a coding of a D0L word. Another famous open problem about the Kolakoski word is whether the letter frequency exists and is indeed \( \frac{1}{2} \), as computer experiments seem to support [44].

Some other open problems concerning PD0L words were recently solved by the first two authors, see further Section 6.
2.3 Toeplitz Words

A subclass of the class of PDOL sequences is formed by Toeplitz words [35]. A Toeplitz word $T_x$ over an alphabet $\Sigma$ is generated by a seed word $x \in \Sigma(\Sigma \cup \{?\})^*$ with $? \notin \Sigma$, as follows. For $u \in \Sigma^\omega$ let $x^\omega[u]$ denote the sequence obtained by replacing the subsequence of $?$’s in $x^\omega$ by $u$. Then $T_x$ is the unique solution for $u$ in the equation

$$u = x^\omega[u].$$

So in order to construct $T_x$, we start with the periodic $x^\omega$ and then replace its subsequence of $?$’s by the sequence $T_x$ under construction. For example, if $x = 101?$, then


$$T_x = 10110101101101101101101101\cdots$$

where the underlined subsequence is identical to the whole sequence. This sequence $T_x$ is known as the period doubling sequence [3, Example 6.4.3], which also is the sequence of first differences (modulo 2) of the Thue-Morse sequence $M$.

2.4 Automatic Sequences

An important subfamily of the morphic sequences is that of the automatic sequences, to which the beautiful monograph [3] is devoted. One way to characterize automatic sequences is that they can be obtained by iterating a uniform morphism, and apply a coding afterwards.

The standard definition of automatic sequences is via deterministic finite automata with output (DFAOs) that produce an element of a sequence when fed the index of the element as input. As an example we consider again the Thue-Morse sequence $M$. The $n$-th element $M(n)$ is the parity of the number of 1’s in $(n)_2$, the binary representation of $n$. This is realized by the automaton displayed in Figure 2.

![Figure 2: DFAO generating the Thue-Morse sequence.](image)

The automaton has states $\{q_0, q_1\}$, initial state $q_0$, input alphabet $\{0, 1\}$ and output alphabet $\{0, 1\}$. The output letter assigned to $q_0$ is 0 and to $q_1$ is 1 (indicated by state/output in the states of the automaton). The automaton generates the
Thue-Morse sequence 0110100110010110\cdots as follows. The $n$-th letter of the sequence is the output of the automaton when reading $(n)_2$, the base-2 expansion of $n$. For example, for input $(3)_2 = 11$ the automaton ends in state $q_0$ with output 0, and for input $(4)_2 = 100$ in state $q_1$ with output 1.

The automaton of Figure 2 is called a deterministic finite state automaton with output (DFAO). For $k \geq 2$, a $k$-DFAO is an automaton over the input alphabet $\mathbb{N}_{<k} = \{0, 1, \ldots, k-1\}$. An infinite sequence $w \in \Delta^\omega$ is called $k$-automatic if there exists a $k$-DFAO such that for every $n \in \mathbb{N}$ the output of the automaton when reading the word $(n)_k \in \mathbb{N}_{<k}^\ast$ is $w(n)$, with $(n)_k$ the base-$k$ expansion of $n$.

2.5 Sturmian Sequences

Much studied in mathematics, for its implications for number theory, is the family of sequences known as sturmian sequences. Sturmian sequences can be obtained in a well-known direct geometrical way (‘rotation sequences’ or ‘cutting sequences’), namely by intersections with the unit grid in the plane and a straight line from the origin. The most famous example here is the Fibonacci word $F$, 01001010\cdots, obtained from the straight line from the origin with slope $\frac{1}{\varphi} = \varphi - 1$, with $\varphi = \frac{1 + \sqrt{5}}{2}$ the golden ratio, see Figure 3.

We note that the Fibonacci word is also a morphic sequence, $F = h^\omega(0)$ with the morphism $h$ given by $0 \rightarrow 01$, $1 \rightarrow 0$.

3 Streams and Fractals

We will now go back some thirty or forty years in the history, to the time that Seymour Papert developed his educational enterprise ‘turtle graphics’ or ‘turtle geometry’ together with the Logo programming language. It was meant to facilitate Papert’s daughter with her experiments relating art and mathematics. We will use turtles in the form of finite state transducers (see Section 4), with the stipulation that the output alphabet consists of graphical instructions such as ‘draw
one straight line unit’, ‘turn write head over \( \frac{\pi}{3} \)’, etc. We call this a ‘smart’ turtle as it has some memory being a finite state transducer. Now we can transform a stream into a fractal. To do this properly, we have to re-scale the drawn figure when it grows too large for the screen or page, and apply the Hausdorff metric on these successive approximations. For instance the fractal in Figure 4 (left) is

Figure 4: A turtle trajectory for the streams Mephisto Waltz (left), Kolakoski (middle) and Fibonacci (right).

generated from the Mephisto Waltz sequence 001001110001001110 … (obtained by iterating the morphism 0 → 001, 1 → 110 on starting letter 0). An interesting surprise in 2005 by Ma and Holdener was their discovery that the Thue-Morse sequence yields when drawn by a suitable turtle, the snowflake of Helge von Koch (also from 1906). As pointed out by J.-P. Allouche, this discovery was actually anticipated in the early 1980’s by M. Dekking in the mathematical framework of iterated exponential sums. Not all streams and all smart turtles yield a ‘decent’ fractal. Some patterns are chaotic, as for example the notoriously difficult Kolakoski stream 1 22 11 2 122 1 12 11 2 11 … which is the sequence of its own run-lengths (see also Section 2). Other streams yield aesthetically pleasing patterns such as the Fibonacci word \( F = 0100101001001 \cdots \). We are interested in the relation of the hierarchy of streams with fractals, and in particular the question whether properties of fractals can be employed to distinguish ‘degrees of streams’, a notion that we introduce in Section 4 below.

An amusing puzzle is to derive a sequence by looking at a fractal curve. In Figure 5 we have displayed the initial approximations of the Sierpiński arrowhead

Figure 5: Initial approximations of the Sierpiński arrowhead curve [47].
curve [47]. The question arises: what is the sequence behind this fractal curve? In other words, interpreting 0 and 1 as turtle drawing instructions e.g. as follows:

0: move forward one unit length and turn to the left \( \pi/3 \) radials, and
1: move forward one unit length and turn to the right \( \pi/3 \) radials,

the search is for the sequence which generates the curve, in the limit using the Hausdorff metric.

To construct the sequence, we consider Figure 5. The first iteration of the construction, the arrowhead shape, corresponds to the word \( w_1 = 00111100 \). The second iteration is obtained from

\[
w_2 = w_1 0 \overline{w_1} 0 w_1 1 \overline{w_1} 1 w_1 1 \overline{w_1} 1 w_1 0 \overline{w_1} 0
\]

where \( \overline{w_1} = 11000011 \), the mirrored arrowhead. Note that \( w_1 \) and \( \overline{w_1} \) alternate, and the word filled in-between is \( w_1 \) itself. This construction clearly resembles the construction of Toeplitz words, as introduced in Section 2.3, and we find that the Toeplitz word generated by the pattern

\[
x = 00111100 ? 11000011 ?
\]

is the desired sequence \( T_x = 001111000110000110001111001110000111 \cdots \), which we call the Sierpiński sequence \( S = T_x \).

4 Degrees of Streams

We now consider a novel approach [29] to comparing the complexity of streams, namely in terms of reducibility by finite state transducers (FSTs). This gives rise to a hierarchy of stream ‘degrees’ somewhat analogous to the recursion-theoretic degrees of unsolvability (see further Shoenfield [49]). It is the structure and properties of this partial order of degrees that we are interested in. As we shall see, this hierarchy exhibits a variety of unique properties that set it apart from the usual complexity measures for streams.

We explain the notion of finite state transducers, and introduce the partial order of stream degrees. We then discuss and motivate this hierarchy, and compare it to the common complexity measures for streams. Then we sketch a few initial results, and collect some open questions.

**Finite State Transducers.** A finite state transducer (FST) is a finite automaton which reads the input stream letter by letter, in each step producing an output word and changing its state. An example of an FST is depicted in Figure 6, where we write ‘a|w’ along the transitions to indicate that the input letter is \( a \) and the
output word is \( w \). This FST computes the first difference of the input stream. For example, it reduces the Thue-Morse sequence \( M \) to \( T = 101101010111011 \cdots \), the period doubling sequence (see Section 2.3).

![Finite state transducer](image)

*Figure 6: A finite state transducer realizing the difference \( \Delta \) of consecutive bits modulo 2: \( \Delta(abw) = (b - a \,(\text{mod} \, 2)) \Delta(bw) \) for all \( a, b \in \{0, 1\} \) and \( w \in \{0, 1\}^\omega \).*

The working of an FST on a stream is intuitively clear and it is easy to render it formally. At the end of the paper, in our nutshell introduction to infinitary rewriting, we show how the working of an FST on a stream can alternatively be phrased as infinitary rewriting to the output normal form.

### 4.1 A Hierarchy of Streams

Finite state transducers transform streams to streams (or finite words\(^1\)). Thereby transducers give rise to a preorder \( \triangleright \) on the set of all streams: for streams \( u \) and \( v \), we define \( u \triangleright v \), \( u \) is reducible to \( v \), by:

\[
u \triangleright v \iff \text{there is an FST that transforms } u \text{ into } v\]

We write \( u \triangleleft\triangleright v \) if both a forth and a back transformation is possible, that is, \( \triangleleft\triangleright = \triangleleft \cap \triangleright \). It is easily checked that \( \triangleleft\triangleright \) forms an equivalence relation, and we refer to the equivalence classes of \( \triangleleft\triangleright \) as *degrees*. The reducibility relation \( \triangleright \) induces a partial order on the degrees.

Figure 7 sketches a few initial results (obtained in [29]) and a few open questions. The preliminary results are: the hierarchy is not dense, not well-founded, there exist no maximal degrees, and a set of degrees has an upper bound if and only if the set is countable. The morphic degrees, and the computable degrees form interesting subhierarchies. The subhierarchy of computable degrees turns out to have a maximum degree. An interesting notion is that of a ‘prime degree’, an indivisible or minimal degree, see below.

---

\(^1\)The result of the transformation is finite if the transducer outputs the empty word \( \varepsilon \) for almost all letters of the input stream. We are interested in streams only since the set of finite words would merely entail two spurious extra sub-bottom degrees of our hierarchy, one for finite non-empty words and one for the empty word.
The bottom degree $0$ is formed by the ultimately periodic streams, that is, all streams $\sigma$ of the form $\sigma = \tau \gamma \gamma \ldots$ for finite $\tau, \gamma$. Every stream can be reduced to any ultimately periodic stream $\sigma = \tau \gamma \gamma \ldots$ by an FST of the form displayed in Figure 8 consisting of just two states.

Figure 8: An FST reducing any stream to the ultimately periodic stream $\tau \gamma \gamma \ldots$. 

Remark 4.1. We emphasize that it is important that the output given by a state transition is allowed to be a word over the output alphabet, and not just a single letter or the empty word $\varepsilon$, although that may also be the case. Finite state transducers generalize the class of Mealy machines; the latter are restricted to output of precisely one letter in each step. For instance, the transducer shown in Figure 6 is not a Mealy machine, and there exists no Mealy machine implementing this transformation. Belov [8] independently studied the hierarchy arising from Mealy machines; this hierarchy however, does not have the nice properties that we envisage. In particular, the equivalence induced by Mealy machine transduction (forth and back) is not invariant under insertion or removal of finite (possibly scattered) subwords of a stream.
Every pair of degrees $\sigma$ and $\tau$ has an upper bound. This upper bound can be constructed as follows: pick streams $u \in \sigma$ and $v \in \tau$. Let the stream $\text{zip}(u, v)$ be obtained by alternatingly interleaving the elements of the streams $u$ and $v$, that is:

$$\text{zip}(u(0), u(1), u(2), \ldots, v(0), v(1), v(2), \ldots) = \text{u}(0) \text{v}(0) \text{u}(1) \text{v}(1) \text{u}(2) \text{v}(2) \ldots$$

Then the stream $\text{zip}(u, v)$ that can be reduced to both $u$ and $v$:

$$\text{zip}(u, v) \triangleright u \quad \text{zip}(u, v) \triangleright v$$

by the FSTs shown in Figures 9 and 10, respectively. As a consequence, the degree of the stream $\text{zip}(u, v)$ is an upper bound for the degrees $\sigma$ and $\tau$ (the degrees of $u$ and $v$).

As the set of all FSTs is countable, it follows that every degree is countable, and that every degree can only have countably many degrees below it. Let $\{w_0, w_1, w_2, \ldots\}$ be a countable set of streams. Then we can obtain an upper bound by interleaving the streams in the following way:

$$\text{zip}(w_0, \text{zip}(w_1, \text{zip}(w_2, \ldots)))$$

see also Figure 11. As an immediate consequence we obtain that a set of degrees

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sigma_0(0)$</td>
<td>$\sigma_1(0)$</td>
<td>$\sigma_0(1)$</td>
<td>$\sigma_2(0)$</td>
<td>$\sigma_0(2)$</td>
<td>$\sigma_1(1)$</td>
<td>$\sigma_0(3)$</td>
<td>$\sigma_3(0)$</td>
<td>$\sigma_0(4)$</td>
<td>$\sigma_1(2)$</td>
<td>$\sigma_0(5)$</td>
<td>$\sigma_2(1)$</td>
</tr>
</tbody>
</table>

has an upper bound if and only if the set is countable.
There are no maximum degrees, i.e., for each degree a strictly larger one can be constructed. First note that the set of all degrees is uncountable as every degree is countable but there are uncountably many streams. Let $\sigma$ be a degree. We show that $\sigma$ is not maximal. Since there are uncountably many degrees, we can find a degree $\tau$ such that $\sigma \nless \tau$, and then their upper bound is higher than $\sigma$. This establishes a non-constructive argument that every degree start an infinite ascending chain. A constructive example is the following [29]:

$$\ldots$$

$\upg A_3 = 1(10)^3 1(100)^3 1(10000)^3 1(100000000)^3 \ldots$

$\upg A_2 = 1(10)^2 1(100)^2 1(10000)^2 1(100000000)^2 \ldots$

$\upg A_1 = 110 1100 110000 110000000 \ldots$

$\upg A_0 = 111111 \ldots$

forms an infinite ascending chain. For an infinite descending chain, consider:

$$D_0 = 10^{20} 10^{21} 10^{22} 10^{23} 10^{24} 10^{25} 10^{26} \ldots$$

$$\downarrow D_1 = 10^{20} 10^{22} 10^{24} 10^{26} 10^{28} 10^{210} 10^{212} \ldots$$

$$\downarrow D_2 = 10^{20} 10^{24} 10^{28} 10^{212} 10^{216} 10^{220} 10^{224} \ldots$$

$$\downarrow \ldots$$

Note that $A_1 \nless D_0$ and so their degree has an infinite ascending chain as well as an infinite descending chain.

**Prime Degrees.** An interesting notion that suggests itself is that of a prime degree: a stream $\sigma$ is prime if there exists no stream $\tau$ whose degree is strictly intermediate between that of $\sigma$ and the bottom degree $0$. Thus the prime degrees reduce only to $0$ or themselves. An example of a stream of prime degree is the following sequence called *rarified ones* (see [29]):

$$\Pi = 1101001000100001000001 \ldots$$

which can be obtained as the image of the fixed point of the morphism

$$a \mapsto a1 \quad 1 \mapsto 01 \quad 0 \mapsto 0 \quad (1)$$

on the starting word $a$, under the coding $a \mapsto 1$, $0 \mapsto 0$, $1 \mapsto 1$; thus $\Pi$ is a morphic stream. Intuitively, the information content of the stream $\Pi$ is ‘indivisible’: whatever FST is applied on this stream, either the result is eventually periodic (the structure is entirely destroyed), or there is enough structure left for an FST to reconstruct the original stream.
4.2 Motivation

Finite state automata and finite state transducers are ubiquitous in computer science and computational linguistics. Surprisingly, very little is known about the reducibility relation that finite state transducers induce on streams: Given streams $u$ and $v$, is there an FST reducing $u$ to $v$? This is a challenging question, especially for the case of proving non-reducibility, for example of morphic streams. At present there are no methods available for this problem.

Except for the importance of finite state transducers, the study of the FST-hierarchy is intriguing due to its unique characteristics that set it apart from all existing approaches to measuring stream complexity. The hierarchy is robust and fine grained at the same time:

(i) It is robust under the insertion and removal of arbitrary finite amount of elements in the streams (for example, cutting-off of prefixes), and change of encoding. In some sense, the FST-hierarchy classifies streams by their invariant infinite patterns.

(ii) Thus on the one hand we have a fairly general notion of transformation, but on the other hand it is not too strong, leading to a fine grained structure of degrees. By contrast, allowing Turing Machines for the transformations would be far too strong. Most of the interesting streams are computable, and with Turing Machines any pair of such computable streams could be transformed into each other, and our endeavour would trivialize. The hierarchy of degrees arising from Turing Machines is the well-known recursion theoretic degrees of unsolvability that classify the intrinsic ‘difficulty’ of a set of natural numbers (Shoenfield [49]).

4.3 Comparison with Standard Complexity Measures

The hierarchy of stream degrees is very different from the common approaches for measuring the complexity of streams: (i) the recursion-theoretic hierarchy identifies all computable streams, (ii) Kolmogorov complexity can be increased arbitrarily by the insertion of elements, and (iii) subword complexity can be trivial (linear) even for non-computable streams. The subword complexity of a stream $\sigma$ is a function $\xi : \mathbb{N} \to \mathbb{N}$ such that $\xi(n)$ is the number of subwords of length $n$ occurring in $\sigma$. The Kolmogorov complexity $K(\sigma) \in \mathbb{N}$ of a computable stream $\sigma$ is the length of the shortest program computing $\sigma$. 

83
4.4 Open Questions

The study of the hierarchy concerns the very core of finite state transducers: the transducibility relation they induce on streams. Surprisingly, although finite automata and finite state transducers are ubiquitous in computer science, only little is known about this transducibility relation. Given streams $\sigma$ and $\tau$, there are hardly any methods available to determine whether $\sigma \triangleright \tau$, that is, can $\sigma$ be reduced to $\tau$ via some FST? This is challenging especially for the case of proving that $\sigma$ cannot be transduced to $\tau$. In special cases, pumping lemma arguments may work, but they fail for basically all streams that we are interested in (e.g. morphic streams). If $\sigma$ is morphic and $\tau$ not, then we can make use of a famous theorem by Dekking stating that morphic streams are closed under FST transduction. In all other cases, we are lost.

We mention a few intriguing open questions:

(i) What is the structure of the partial order of degrees?

(ii) How many prime degrees exist?

(iii) Is the degree of the Thue-Morse sequence $M$ prime?

(iv) How do the degrees of some well-known streams compare? For example, are $M$ and $S$ of the same degree? Here $S$ is the Sierpiński stream, see Figure 5.

(v) How to prove non-reducibility $\sigma \ntriangleleft \tau$ (e.g. for morphic streams $\sigma, \tau$)?

(vi) Do the structures displayed in Figure 12 exist?

![Figure 12: Possible structures in the FST-hierarchy: a diamond, and a line. The arrows $S \rightarrow T$ mean $S \triangleright T$. Using transitivity of $\triangleright$ we leave some arrows implicit. Moreover, we assume that if $S$ is a degree and $S \triangleright T$, then $T$ is depicted as well. In particular there are no intermediate degrees between two displayed nodes connected by an arrow.](image)

(vii) What is the structure of the partial order of degrees restricted to different families of streams? E.g. the families of computable or morphic streams?
5 Zip Goes a Million

Elias Howe, the inventor of the sewing machine, patented in 1851 an ‘automatic, continuous clothing closure’, later known as zip or zipper. The name is an onomatopoeia, a sound imitation. Apart from its use as a coding device for American postal services around 1950, ‘zip’ made its entrance also in the world of light culture, giving rise in 1919 to the Broadway play *Zip! goes a million*, and a 1954 remake as a London musical *Zip goes a million*, the former unsuccessful, but the latter quite successful. The plot was based on the book *Brewster’s millions* from 1902, describing the problem how to lose $1000 000 in order to gain $7000 000.

In this section we are concerned with the more serious culture of stream specifications, and we will endeavour to describe how the stream operator ‘zip’, in the literature known as *perfect shuffle*, can profitably be used to give an elegant alternative definition of automatic sequences, and how its use suggests some further challenging questions. First, we will consider zip-specifications, to be followed by an excursion to mix-automatic sequences.

5.1 Destructing Automatic Sequences

For $i, k \in \mathbb{N}$, we define projection functions $\pi_{i,k} : \Delta^\omega \rightarrow \Delta^\omega$ by the equations:

$$
\pi_{0,k}(x : \sigma) = x : \pi_{k-1,k}(\sigma), \quad \pi_{i+1,k}(x : \sigma) = \pi_{i,k}(\sigma)
$$

(2)

So $\pi_{i,k}(\sigma)$ is an arithmetic subsequence of $\sigma$:

$$
\pi_{i,k}(\sigma) = \sigma(i) \sigma(i + k) \sigma(i + 2k) \cdots
$$

The functions $\pi_{0,k}, \ldots, \pi_{k-1,k}$ are the destructors of $\text{zip}_k$, that is, we have

$$
\pi_{i,k}(\text{zip}_k(\sigma_0, \ldots, \sigma_{k-1})) = \sigma_i \quad (0 \leq i < k)
$$

(3)

**Definition 5.1.** Let $\Phi$ be a set of stream functions $\Delta^\omega \rightarrow \Delta^\omega$, and $\sigma \in \Delta^\omega$ a stream. The $\Phi$-derivatives of $\sigma$ are the smallest set $D \subseteq \Delta^\omega$ such that: $\sigma \in D$, and $\phi(\tau) \in D$ whenever $\tau \in D$ and $\phi \in \Phi$.

The derivatives provide an elegant, iterative way of defining the $k$-kernels [3]. The $k$-kernel of a sequence $\sigma$ is the set of $\{\pi_{0,k}, \ldots, \pi_{k-1,k}\}$-derivatives of $\sigma$. Then the well-known characterization of $k$-automatic sequences via finite $k$-kernels can be phrased as follows:

**Theorem 5.1.** Let $k \in \mathbb{N}$. A stream $\sigma \in \Delta^\omega$, is $k$-automatic if and only if the set of $\{\pi_{0,k}, \ldots, \pi_{k-1,k}\}$-derivatives of $\sigma$ is finite.
This characterization can be generalized to other sets of derivatives:

**Theorem 5.2.** Let $k > 1$. Let $\Phi$ be a finite set of projection functions $\pi_{i,n}$ with $i, n \in \mathbb{N}$ and $n \geq 1$ such that the set $\mathbb{N} \setminus \{a + nb \mid \pi_{a,b} \in \Phi, \ n \in \mathbb{N}\}$ is finite. Then $\sigma \in \Delta^\omega$ is $k$-automatic if and only if the set of $\Phi$-derivatives of $\sigma$ is finite.

For example, the set of $\{\pi_{1,k}, \ldots, \pi_{k,k}\}$-derivatives is finite if and only if the sequence is $k$-automatic, see further [27]. The increased flexibility in choosing the projection functions $\Phi$ can help to simplify proofs and disproofs of $k$-automaticity. In the terminology of [40], the derivatives $\{\pi_{0,k}, \ldots, \pi_{k-1,k}\}$ are a complete set of cooperations, forming (together with head) a cobasis for streams. This cobasis gives rise to a final coalgebra for automatic sequences, see [41, 32].

### 5.2 Automatic Sequences via Zip-Specifications

Automatic sequences can be defined via term rewriting, or equational specifications in a restricted format: zip-specifications, see further [32]. For example, the Thue–Morse sequence is obtained by the succinct zip-specification

$$M = 0 : X \quad X = 1 : \text{zip}(X, Y) \quad Y = 0 : \text{zip}(Y, X)$$

(We obtain a term rewriting system by orienting the equations from left to right.) Here the function zip interleaves the elements of two streams alternatingly, also known as perfect shuffle:

$$\text{zip}(\sigma_0 : \sigma_1 : \sigma_2 : \ldots, \tau_0 : \tau_1 : \tau_2 : \ldots) = \sigma_0 : \tau_0 : \sigma_1 : \tau_1 : \sigma_2 : \tau_2 : \ldots$$

As a term rewriting rule, ‘zip’ can be defined as follows:

$$\text{zip}(x : \sigma, \tau) = x : \text{zip}(\tau, \sigma)$$

Naturally, the function zip can be generalized to interleaving $k$ arguments:

$$\text{zip}_k(x_1 : s_1, s_2, \ldots, s_k) = x_1 : \text{zip}_k(s_2, \ldots, s_k, s_1)$$

A zip-$k$ specification consists of recursion equations $X_1 = r_1, \ldots, X_n = r_n$ such that $X_1, \ldots, X_n$ are recursion variables, and the right-hand sides $r_1, \ldots, r_n$ are terms inductively generated by the following grammar:

$$G_k ::= X_i \mid a : G_k \mid \text{zip}_k(G_k, \ldots, G_k) \quad (1 \leq i \leq n, a \in \Sigma)$$

That is, the right-hand sides of the equations are built from recursion variables, prefixing a symbol at the head of a stream, and the stream function zip$_k$. 
It turns out, that the class of streams definable by zip-
\( k \)-specifications is precisely the class of \( k \)-automatic sequences [32]. Thus zip-specifications provide a term rewriting syntax for automatic sequences.

Let us explain why zip-specifications give rise to automatic sequences at the example of the specification (4). We show that the sequence is 2-automatic by computing the 2-kernel, the set of \( \{ \pi_{0,2}, \pi_{1,2} \} \)-derivatives. To this end, we compute the derivatives symbolically using the equations (2) and (3), and unfolding stream constants defined by (4) whenever necessary. For better readability we write even for \( \pi_{0,2} \) and odd for \( \pi_{1,2} \). We obtain:

\[
\begin{align*}
even(M) &= \even(0 : X) = 0 : \odd(X) = 0 : \odd(1 : \zip(X, Y)) = 0 : \even(\zip(X, Y)) = 0 : X = M \\
\odd(M) &= \odd(0 : X) = \even(X) = \even(1 : \zip(X, Y)) = 1 : \odd(\zip(X, Y)) = 1 : Y = 1 : Y
\end{align*}
\]

Likewise \( \even(1 : Y) = 1 : Y \) and \( \odd(1 : Y) = M \). Thus the set of derivatives is finite, namely \( \{ M, 1 : Y \} \), and by Theorem 5.1, the stream \( M \) is 2-automatic.

The derivatives of a stream specification can be illustrated using a graph. The nodes of the graph are the \{even, odd\}-derivatives. The edges represent the derivative relation and are labeled with even and odd, correspondingly. The graph corresponding to the \{even, odd\}-derivatives of \( M \) is shown in Figure 13. If we replace in this graph ‘even‘ by 0 and ‘odd’ by 1, we obtain a 2-DFAO that generates the Thue–Morse sequence \( M \) as automatic sequence.

\[ \text{Figure 13: The } \langle \text{head, even, odd}\rangle\text{-coalgebra induced by } M. \]

5.3 Mix-Automatic Sequences

The correspondence of \( k \)-automatic sequences with zip-\( k \) specifications, leads to the natural question: What class of sequences is obtained when allowing zips of different arities in the same specification? For example:

\[
Z = \zip_2(0 : Z, Y) \quad Y = 1 : \zip_3(Z, Y, 0 : Z)
\]
We call the arising class of sequences \textit{mix-automatic}. It forms a proper extension of the class of automatic sequences [32], and is in contrast to automatic sequences, closed under zipping (perfect shuffle). For example, zipping a 2-automatic and a 3-automatic sequence, both not ultimately periodic, yields a non-automatic (yet mix-automatic) sequence. While automatic sequences have at most linear subword complexity, and morphic sequences at most quadratic, the subword complexity of mix-automatic sequences can exceed an arbitrary polynomial, see [22].

Mix-automatic sequences can be defined via a generalization of \(k\)-DFAOs allowing that the input alphabet depends on the current state. We call these automata \textit{mix-DFAOs}. Let us consider the example of a mix-DFAO shown in Figure 14.

![Figure 14: An example of a mix-DFAO.](image)

The state \(q_0\) has two outgoing edges, reflecting the input alphabet \(\{0, 1\}\), while \(q_1\) has three outgoing edges, reflecting the input alphabet \(\{0, 1, 2\}\).

**Dynamic Radix Numeration Systems.** The numeration systems these automata operate on are no longer the standard base-\(k\) representation. The corresponding number systems are called \textit{dynamic radix numeration systems}, a generalization of \textit{mixed radix} numeration systems [38]. In this number representation, the base for each digit is determined depending on the lower-significance digits. Thus we let the automata read from the least to the most significant digit (from right to left). We write \((n)_M\) for the number representation of \(n\) that serves as input for the automaton \(M\). For \(M\) the automaton from Figure 14, the representations of the first eight numbers are

\[
\begin{align*}
(0)_M &= \varepsilon & (2)_M &= 1_20_2 & (4)_M &= 1_20_20_2 & (6)_M &= 1_31_20_2 \\
(1)_M &= 1_2 & (3)_M &= 1_31_2 & (5)_M &= 2_31_2 & (7)_M &= 1_30_31_2
\end{align*}
\]

where a subscript \(b\) (not part of the number representation) in \(d_b\) indicates the base employed for \(d\). Let us explain this at the example \((17)_M = 1_20_22_31_2\). Knowing the base for each digit, we can reconstruct the value of the representation as follows: \(17 = 1 \cdot 2 \cdot 3 \cdot 2 + 0 \cdot 3 \cdot 2 + 2 \cdot 2 + 1\) where each digit is multiplied with the product of the bases of the lower digits. Given just the representation 1021, the base of each of the digits is determined by the input alphabet of the state of the automaton reading the digit. The states \(q_0\) and \(q_1\) of \(M\) have input alphabets \(\{0, 1\}\) and \(\{0, 1, 2\}\) and thus expect the input in base 2 and 3, respectively. When reading 1021 (right to left) the automaton \(M\) visits the states \(q_0, q_1, q_0, q_0\) and \(q_1\). Annotating the
input digits with the state of the automaton when reading the digit, we obtain
\[1_{q_0}, 0_{q_0}, 2_{q_1}, 1_{q_0},\] and mapping states to their expected base yields \(1_20_22_31_2\).

Let \(M\) be a mix-DFAO. Then every \(n \in \mathbb{N}\) has a unique representation \((n)_M = d_m \cdots d_0\) (without leading zeros). This representation can be computed as follows. Assume that we have determined the value of the digits \(d_{i-1} \cdots d_0\) with corresponding bases \(b_{i-1} \cdots b_0\). The base \(b_i\) of digit \(d_i\) is determined by the input alphabet of the state of the automaton after reading \(d_{i-1} \cdots d_0\) (right to left), and digit \(d_i\) is the remainder of the division of \(n - \sum_{0 \leq j < i} d_j (b_{j-1} \cdots b_1 \cdot b_0)\) by \(b_i\).

A mix-DFAO \(M\) gives rise to a mix-automatic sequence \(w \in \Delta^\omega\) as follows: for every \(n \in \mathbb{N}\), \(w(n)\) is the output of \(M\) when reading \((n)_M\).

### 5.4 Research Questions

The concept of mix-automatic sequences and dynamic-radix numeration systems are very recent, and many interesting questions remain. We highlight three particularly intriguing, and challenging questions:

1. (J.-P. Allouche) Characterize the intersection of mix-automatic and morphic sequences. (Note that at least all automatic sequences are in.)

2. Is the following problem decidable: Given two mix-DFAOs, do they generate the same sequence?

3. Can Cobham’s Theorem (below) be generalized to mix-automatic sequences?

**Theorem 5.3** (Cobham’s Theorem [11]). Let \(k, \ell \geq 2\) be multiplicatively independent (i.e., \(k^a \neq \ell^b\), for all \(a, b > 0\)), and let \(w \in \Delta^\omega\) be both \(k\)- and \(\ell\)-automatic. Then \(w\) is ultimately periodic.

For more details, we refer to [32, 22].

### 6 Periodically Iterated Morphisms

Infinite words obtained from periodically iterating multiple morphisms, so-called \(PD0L\) words, have been studied in [15, 16, 42, 10]. In particular, Lepistö [42] shows that for all \(r \in \mathbb{R}\) there is a \(PD0L\) word whose subword complexity is in \(\Omega(n^r)\). Subword complexity [31, 1, 3] is a natural characteristic of streams. The subword complexity of a stream \(u\) is a function \(\mathbb{N} \rightarrow \mathbb{N}\) mapping \(n\) to the number of \(n\)-length words that occur in \(u\). It is well-known that \(CD0L\) words can have at most quadratic subword complexity [19]. Hence from Lepistö’s result [42] it follows that there are \(PD0L\) words that are not \(CD0L\). Cassaigne and Karhumäki [10]
show that all Toeplitz words are PD0L words, and that some of them have subword complexity in $\Omega(n^r)$ for $r > 2$, thus forming an alternative proof of what was established in [42]. We note that, conversely, the existence of CD0L words that are not PD0L words was shown in [15].

Recently, the first two authors [27] have shown that PD0L words can even exhibit exponential subword complexity, answering a question raised by Lepestö [42], Cassaigne and Karhumäki [10] on the existence of such words. Another open problem concerned the decidability of the first-order theories of PD0L words [46]; from [27] it follows that is already undecidable whether a certain letter occurs in a PD0L word. This stands in contrast to the situation for D0L words (purely morphic words), which are known to have at most quadratic subword complexity, and for which the monadic theory is decidable.

Actually, [27] shows a stronger result, which can be paraphrased by

**Periodically iterated morphisms are Turing-complete.**

The proof of this result is based on an encoding of Fractran [12, 13] programs as PD0L systems. Fractran is a very simple, yet Turing complete programming language invented by John Horton Conway. We will here describe the essence of this encoding. In particular we show how the halting problem of Fractran programs is translated to the productivity problem for erasing PD0L systems, i.e., systems where morphisms are allowed to map letters to the empty word $\varepsilon$. A PD0L system is called *productive* if it generates an infinite word. Productivity is discussed in the wider context of term rewriting in Section 7. In [27] this construction is extended to non-erasing PD0L systems that also record the output of the Fractran program, so that any computable stream can be ‘embedded’ in a PD0L word. Let us first give a nutshell introduction to Fractran.

### 6.1 Fractran

A Fractran program $F$ is a finite list of fractions

$$F = \frac{n_1}{d_1}, \ldots, \frac{n_k}{d_k}$$

with $n_i, d_i$ positive integers. Let $f_i = \frac{n_i}{d_i}$. The action of $F$ on an input integer $N \geq 1$ is to multiply $N$ by the first ‘applicable’ fraction $f_i$, that is, the fraction $f_i$ with $i$ the least index such that the product $N' = N \cdot f_i$ is an integer again, and then to continue with $N'$. The program halts if there is no applicable fraction for the current integer $N$. For example, we consider the program

$$F = \frac{5}{2 \cdot 3}, \frac{1}{2}, \frac{1}{3}$$
and the run of $F$ on input $N = 2^3 3^5$:

$$2^3 3^5 \rightarrow 2^2 3^4 5^1 \rightarrow 2^1 3^3 5^2 \rightarrow 2^0 3^2 5^3 \rightarrow 2^0 3^1 5^3 \rightarrow 2^0 3^0 5^3.$$  

Each multiplication by $\frac{5}{6}$ decrements the exponents of 2 and 3 while incrementing the exponent of 5. Once $\frac{5}{6}$ is no longer applicable, i.e., when one of the exponents of 2 and 3 in the prime factorization of the current integer $N$ equals 0, the other is set to 0 as well. Hence, executing $F$ on $N = 2^a 3^b$ halts after $\max(a, b)$ steps with $5^{\min(a,b)}$.

Thus the prime numbers that occur as factors in the numerators and denominators of a Fractran program can be regarded as registers, and if the current working integer is $N = 2^a 3^b 5^c \ldots$ we can say that register 2 holds $a$, register 3 holds $b$, and so on.

The real power of Fractran comes from the use of prime exponents as states. To explain this, we temporarily let programs consist of multiple lines of the form

$$\alpha : \frac{n_1}{d_1} \rightarrow \alpha_1, \frac{n_2}{d_2} \rightarrow \alpha_2, \ldots, \frac{n_m}{d_m} \rightarrow \alpha_m$$

(6)

forming the instructions for the program in state $\alpha$: multiply $N$ with the first applicable fraction $\frac{n_i}{d_i}$ and proceed in state $\alpha_i$, or terminate if no fraction is applicable. We call the states $\alpha_1, \ldots, \alpha_m$ in (6) the successors of $\alpha$, and we say a state is looping if it is its own successor.

The program $P_{\text{add}}$ given by the lines

$$\alpha : \frac{2 \cdot 5}{3} \rightarrow \alpha, \frac{1}{1} \rightarrow \beta$$ and $$\beta : \frac{3}{5} \rightarrow \beta$$

realizes addition; running $P_{\text{add}}$ in state $\alpha$ on $N = 2^a 3^b$ terminates in state $\beta$ with $2^{a+b} 3^b$.

A program with $n$ lines is called a Fractran-$n$ program. A flat list of fractions $f_1, \ldots, f_k$ now is a shorthand for the Fractran-1 program $\alpha : f_1 \rightarrow \alpha, f_2 \rightarrow \alpha, \ldots, f_k \rightarrow \alpha$. Conway [13] explains how every Fractran-$n$ program ($n \geq 2$) can be compiled into a Fractran-1 program, using the following steps:

(i) For every looping state $\alpha$, introduce a ‘mirror’ state $\varnothing$, substitute $\varnothing$ for all occurrences of $\alpha$ in the right-hand sides of its program line, and add the line

$$\varnothing : \frac{1}{1} \rightarrow \alpha$$

(ii) Replace state identifiers $\alpha$ by ‘fresh’ prime numbers.
(iii) For every line of the form (6) append the following fractions:

\[
\frac{n_1 \cdot \alpha_1}{d_1 \cdot \alpha}, \frac{n_2 \cdot \alpha_2}{d_2 \cdot \alpha}, \ldots, \frac{n_k \cdot \alpha_m}{d_m \cdot \alpha}
\]

(preserving the order) to the list of fractions constructed so far.

We explain these steps on the program \(P_{\text{add}}\). Step (i) of splitting loops, results in

\[
\begin{align*}
\alpha : & \quad \frac{2 \cdot 5}{3} \rightarrow \alpha, \quad \frac{1}{1} \rightarrow \beta \\
\beta : & \quad \frac{3}{5} \rightarrow \beta \\
\alpha & \quad : \quad \frac{1}{1} \rightarrow \alpha \\
\beta & \quad : \quad \frac{1}{1} \rightarrow \beta.
\end{align*}
\]

In step (ii), we introduce ‘fresh’ primes to serve as state indicators; for example, \(\langle \alpha, \alpha, \beta, \beta \rangle = \langle 7, 11, 13, 17 \rangle\). Third, we replace lines by fractions, to obtain the program

\[
F_{\text{add}} = \frac{2 \cdot 5 \cdot \alpha}{3 \cdot \alpha}, \quad \frac{\alpha}{\alpha}, \quad \frac{\beta}{\alpha}, \quad \frac{3 \cdot \beta}{5 \cdot \beta}, \quad \frac{\beta}{\beta}.
\]

Then indeed the run of \(F_{\text{add}}\) on \(2^a 3^b \alpha\) ends in \(2^{a+b} 3^b \beta\).

For ‘sensible’ programs any state indicator has value 0 (‘off’) or 1 (‘on’), and the program is always in exactly one state at a time. Hence, if a program \(F\) uses primes \(r_1, \ldots, r_p\) for storage, and primes \(\alpha_1, \ldots, \alpha_q\) for control, at any instant the entire configuration of \(F\) (= register contents + state) is uniquely represented by the current working integer \(N\)

\[
N = r_1^{e_1} r_2^{e_2} \cdots r_p^{e_p} \alpha_j
\]

for some integers \(e_1, \ldots, e_p \geq 0\) and \(1 \leq j \leq q\).

The reason to employ two state indicators \(\alpha\) and \(\alpha\) to break self-loops in step (i), is that each state indicator is consumed whenever it is tested, and so we need a secondary indicator \(\alpha\) to say “continue in the current state”. This secondary indicator \(\alpha\) is swapped back to the primary indicator \(\alpha\) in the next instruction, and the loop continues.

The halting problem for Fractran programs is undecidable (implicit in [13], explicit in [21, 32]).

**Proposition 6.1.** The input-2 halting problem for Fractran programs, that is, deciding whether a program halts for the starting integer \(N = 2\), is \(\Sigma_1^0\)-complete.
6.2 Encoding Fractran Programs as PD0L Systems

Definition 6.1. Let \( H = \langle h_0, \ldots, h_{p-1} \rangle \) be a tuple of morphisms \( h_i : \Sigma^* \rightarrow \Sigma^* \). We define the map \( H : \Sigma^* \rightarrow \Sigma^* \) as follows:
\[
H(a_0 a_1 \cdots a_n) = u_0 u_1 \cdots u_n
\]
where \( u_i = h_k(a_i) \), with \( k \equiv i \pmod{p} \) and \( k \in \mathbb{N}_{<p} \).

If \( s \in \Sigma^* \) is such that \( s \preceq H(s) \), then the triple \( \mathcal{H} = \langle \Sigma, H, s \rangle \) is called a PD0L system. Then in the metric space \( \langle \Sigma^\infty, d \rangle \) the limit
\[
H^\omega(s) = \lim_{i \to \infty} H^i(s)
\]
exists, and we call \( H^\omega(s) \) the PD0L word generated by \( \mathcal{H} \). We say that \( \mathcal{H} \) is productive if \( H^\omega(s) \) is infinite, and \( \mathcal{H} \) is erasing if some of its morphisms \( h_i \) are erasing. If \( x \) is a PD0L word generated by \( p \) morphisms, and \( x = uvy \) for some \( u, v \in \Sigma^* \) and \( y \in \Sigma^\infty \), we say that the factor \( v \) of \( x \) occurs at morphism index \( i \) when \( i \in \mathbb{N}_{<p} \) and \( i \equiv |u| \pmod{p} \).

Example 6.1. Cassaigne and Karhumäki [10] show that all Toeplitz words are PD0L words. Consider for example the Toeplitz word \( T_y = 121211221112221 \cdots \) generated by the seed word \( y = 12??? \) (see Section 2.3). Then we have \( T_y = H^\omega(1) \) where \( H = \langle h_0, h_1, h_2 \rangle \) and \( h_0(a) = 12a \) and \( h_1(a) = h_2(a) = a \) for all \( a \in \{1, 2\} \). Moreover, from [10, Theorem 5] it follows that the subword complexity of \( T_y \) is in \( \Theta(n^r) \) with \( r = \frac{\log 5}{\log 5 - \log 3} \approx 3.15066 \), thus forming an alternative proof (since morphic words have quadratic subword complexity at most) of what was established by Lepistö [42]: there are PD0L words that are not morphic.

In [27] it is shown that the problem of deciding productivity of erasing PD0L systems is undecidable. The idea is to encode a given Fractran program \( F \) as a PD0L system \( \mathcal{H}_F = \langle \Sigma, H, s \rangle \) such that \( H^\omega(s) \) is infinite if and only if \( F \) does not terminate on input 2.

Definition 6.2. Let \( F = \frac{n_1}{d}, \ldots, \frac{n_d}{d} \) be a Fractran program (every program can be brought into this form by taking \( d \) the least common denominator of the fractions). We define the PD0L system \( \mathcal{H}_F = \langle \Gamma, H, s \rangle \) with
\[
\Gamma = \{ s, \omega, a, A, b, B \} \quad \text{and} \quad H = \langle h_0, \ldots, h_{d-1} \rangle,
\]
where for every \( i \) with \( 0 \leq i < d \) the morphism \( h_i : \Gamma^* \rightarrow \Gamma^* \) is defined by
\[
h_i(s) = s \omega^{d-1} \ a \ a \ b \ \omega^{d-1}
\]
\[
h_i(\omega) = \varepsilon
\]
\[
\begin{align*}
  h_i(a) &= \begin{cases} 
    A \cdot d^{-1} & \text{if } i = d - 1 \\
    \varepsilon & \text{otherwise}
  \end{cases} \\
  h_i(b) &= B \cdot d^{-1-i} \\
  h_i(A) &= \begin{cases} 
    a^{n_{\psi}(i)} & \text{if } \psi(i) \text{ is defined} \\
    \varepsilon & \text{otherwise}
  \end{cases} \\
  h_i(B) &= \begin{cases} 
    a^{\frac{n_{\psi}(i)}{d}} \cdot b \cdot d^{-1} & \text{if } \psi(i) \text{ is defined} \\
    \varepsilon & \text{otherwise}
  \end{cases}
\end{align*}
\]

In [27] it is shown that productivity of the PD0L system \( H_F \) coincides with \( F \) running forever on input 2. Here we give some intuitive explanation, and illustrate the working of \( H_F \) on an example program \( F \).

Let \( F \) be a Fractran program with common denominator \( d \), and (finite or infinite) run \( N_0, N_1, N_2, \ldots \). Let \( q_i \in \mathbb{N} \) and \( r_i \in \mathbb{N}_{<d} \) such that \( N_i = q_i d + r_i \), for all \( i \geq 0 \). We let \( x_n \) be the ‘contribution’ of the iteration \( H^{n+1} \), i.e., \( x_n \) is such that \( H^{n+1}(s) = H^n(s) x_n \). Then \( H^\omega(s) = s \cdot x_0 \cdot x_1 \cdot x_2 \cdot \ldots \). We will display \( H^\omega(s) \) in separate lines each corresponding to an \( x_n \). The computation of the word \( H^\omega(s) \) proceeds in two alternating phases: the transition from even to odd lines corresponds to division by \( d \), and the transition from odd to even lines corresponds to multiplication by the currently applicable fraction \( \frac{n_{\psi}(N_i)}{d} \). These phases are indicated by the use of lower- and uppercase letters, that is, \( x_{2n} \in \{\text{\textit{\textordbar} \textordbar}, \text{\textit{a}, \textit{b}} \}^* \) and \( x_{2n+1} \in \{\text{\textit{\textordbar} \textordbar}, \text{\textit{A}, \textit{B}} \}^* \), as can be seen from the definition of the morphisms. Now the intuition behind the alphabet symbols (in view of the defining rules of the morphisms) can be described as follows. We use \( s \) as the starting symbol, and the symbol \( \text{\textordbar} \textordbar \) is used to shift the morphism index of subsequent letters.

In every even line \( x_{2i} \)

(i) there is precisely one block of \( \text{\textit{a}} \)'s; this block occurs at morphism index 0 and is of length \( N_i \), representing the current value \( N_i \) in the run of \( F \);

(ii) \( \text{\textit{b}} \) is a special marker for the end of a block of \( \text{\textit{a}} \)'s, and so is positioned at morphism index \( r_i \), the remainder of dividing \( N_i \) by \( d \).

In every odd line \( x_{2i+1} \)

(iii) the number of \( \text{\textit{A}} \)'s corresponds to the quotient \( q_i \), and every occurrence of \( \text{\textit{A}} \) is positioned at morphism index \( r_i \);

(iv) \( \text{\textit{B}} \) (also at morphism index \( r_i \)) takes care of the multiplication of the remainder \( r_i \) with \( \frac{n_{\psi}(N_i)}{d} \). Then \( \psi(N_i) = \psi(r_i) \) ensures that the morphism can select the right fraction to multiply with.
We illustrate the encoding by means of an example.

**Example 6.2.** Consider the Fractran program \( \frac{9}{2}, \frac{5}{3} \), or equivalently \( F = \frac{27}{6}, \frac{10}{6} \) and its finite run 2, 9, 15, 25. Following Definition 6.2 we construct the PD0L system \( \mathcal{H}_F = \langle \Gamma, H, s \rangle \) with \( H = \langle h_0, \ldots, h_5 \rangle \) and

\[
\begin{align*}
  h_1(s) &= s \cdot 5 \cdot a \cdot b \cdot 5 \\
  h_i(\omega) &= \varepsilon & h_i(b) &= B \cdot 5^{-i} \\
  h_0(a) &= \ldots = h_4(a) = \varepsilon & h_0(B) &= b \cdot 5 \\
  h_5(a) &= A \cdot 5 & h_2(B) &= a^9 \cdot b \cdot 5 \\
  h_0(A) &= h_2(A) = h_4(A) = a^{27} & h_4(B) &= a^{18} \cdot b \cdot 5 \\
  h_3(A) &= a^{10} & h_3(B) &= a^5 \cdot b \cdot 5 \\
  h_1(A) &= h_5(A) = \varepsilon & h_1(B) &= h_5(B) = \varepsilon
\end{align*}
\]

for \( i \in \mathbb{N}_{<6} \). Then \( H^\omega(s) \) is finite and the stepwise computation of this fixed point can be displayed as follows. To ease reading, we write below each letter its morphism index. Let \( z_n \) denote the morphism index of \( x_n \). The word \( H^\omega(s) = s \cdot x_0 \cdot x_1 \cdots \) is broken into lines in such a way that every line \( x_{n+1} \) is the image of the previous line \( x_n \) under \( H_{z_n} \) (except for the line \( x_0 \), which is the tail of the image of \( s \) under \( H_0 \)).

\[
\begin{align*}
  x_0 &= a \cdot 5 \cdot a \cdot a \cdot b \cdot 5 \\
  x_1 &= B \cdot 3 \\
  x_2 &= a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot b \cdot 5 \\
  x_3 &= A \cdot 5 \cdot B \cdot 2 \\
  x_4 &= a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot b \cdot 5 \\
  x_5 &= A \cdot 5 \cdot A \cdot 5 \cdot B \cdot 2 \\
  x_6 &= a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot b \cdot 5 \\
  x_7 &= A \cdot 5 \cdot A \cdot 5 \cdot A \cdot 5 \cdot B \cdot 4 \\
  x_8 &= \varepsilon
\end{align*}
\]

**Lemma 6.2 ([27]).** For all Fractran programs \( F \), the PD0L system \( \mathcal{H}_F \) is productive if and only if \( F \) does not terminate on input 2.
Hence, by Lemma 6.2 and Proposition 6.1, deciding productivity of PD0L systems is undecidable.

**Theorem 6.3** ([27]). The following problem is $\Pi_1^0$-complete: Given a PD0L system $\mathcal{H}$, is $\mathcal{H}$ is productive? □

## 7 The Productivity Question

So far, we have met in our guided tour through the landscape of streams some of the important families, some famous members of these families, and we have considered some important properties of streams, at the same time outlining some tough questions that confront us. There is one basic issue that we still have been silent about (although it is in fact already present in the previous section), but is of paramount importance in stream specifications, in particular when we view the stream landscape from the point of view of functional programming applications. The issue was considered by Dijkstra, who also coined the word ‘productivity’ of a stream definition. Two important papers gave initial approaches to this important problem of productivity: Hughes, Pareto and Sabry [33], and Telford and Turner [52]. In this section we explain the problem, and briefly mention the realization of solving this problem in an automated way for a restricted yet extensive class of stream specifications.

An important aspect of program correctness is termination. When dealing with programs that construct or process infinite objects, one cannot require termination; instead such programs need to be **productive**. The concept of productivity captures the intuitive notion of unlimited progress, of programs producing values indefinitely, programs immune to deadlock. For instance, control programs, such as an operating system, are not supposed to be terminating, but we want them to be productive. In other words we want the system to keep responding to user input and produce output, thus generating an infinite stream of outputs. Thus productivity guarantees **deadlock-** and **lifelock-freeness**. In lazy functional programming languages such as Miranda, Clean or Haskell, usage of infinite structures such as streams, is common practice. For the correctness of programs dealing with such structures one must guarantee that every finite part of the infinite structure can be evaluated; that is, the specification of the infinite structure must be productive. This holds both for terminating programs that lazily evaluate only finite parts of the infinite structures, as well as for non-terminating programs that directly construct or process infinite objects (like streams of sensor data or user inputs). Since productivity is undecidable in general, the challenge is to find ever stronger methods for proving productivity.

We study termination and productivity in the framework of term rewriting. For
example, the Thue-Morse sequence can be defined as follows:

\[
\begin{align*}
M &= 0 : \text{zip}(\text{inv}(M), \text{tail}(M)) \\
\text{tail}(x : \sigma) &= \sigma \\
\text{inv}(0 : \sigma) &= 1 : \text{inv}(\sigma) \\
\text{flip}(x : \sigma, \tau) &= x : \text{flip}(\tau, \sigma) \\
\text{inv}(1 : \sigma) &= 0 : \text{inv}(\sigma)
\end{align*}
\]  

(7)

A stream specification is called **productive** if not only can the specification be evaluated continually to build up a unique infinite normal form, but the resulting infinite expression is also meaningful in the sense that it is a **constructor normal form** which allows to read off consecutively individual elements of the stream. We emphasize that productivity is not just unique solvability, but also the potential to obtain the solution by evaluation. For example, the specification \( Z = z(Z) \) with \( z(x : \sigma) = 0 : z(\sigma) \), has a unique solution (the stream of zeros), but it cannot be evaluated to obtain this solution.

For specification (7) of the Thue-Morse sequence, it is relatively easy to convince oneself that the definition is productive. Productivity of an arbitrary stream specification is however non-trivial, even undecidable. To get a feeling of the non-triviality, consider the stream specification

\[
J = 0 : 1 : \text{even}(J) \\
\text{even}(x : \sigma) = x : \text{odd}(\sigma) \\
\text{odd}(x : \sigma) = \text{even}(\sigma)
\]  

(8)

This specification ‘deadlocks’; it produces only 4 entries and then starts an infinite idling sequence of function calls: \( J = 0 : 1 : 0 : 0 : \text{even(even(even(...)))} \).

The notion of productivity was first mentioned by Dijkstra [18]. Since then several papers [55, 50, 14, 33, 51, 9] have been devoted to criteria ensuring productivity, and more recently [23, 24, 20, 21, 56, 57, 25, 17, 26, 34]. Technically, the common essence of most of these approaches is a **data-oblivious** analysis, that is, a quantitative analysis of productivity where the concrete values of the elements in the stream are ignored. We have adopted and elaborated this approach in [23, 24, 20, 21]. The recent trend is towards a **data-aware** analysis of productivity via a transformation to termination of term rewriting systems, see further [25, 56, 57, 26]. Let us highlight two of the papers, namely [20] and [26].

In [20] we develop a **data-oblivious** method for proving productivity. This method is provably optimal among all data-oblivious approaches, meaning that no data-oblivious approach can recognize more specifications as productive. Here ‘data-oblivious’ refers to a quantitative analysis of productivity where the concrete values of the elements in the stream are ignored. Previously, all techniques for recognizing productivity have been data-oblivious.

In [26] we have devised a sound and complete transformation of productivity to context-sensitive termination, thereby making the power of termination provers available for proving **data-aware** productivity. For proving termination of TRSs
\[ [M] = \mu M \cdot \text{zip}(\text{inv}(M), \text{tail}(M)) \]
\[ = \mu M \cdot \inf(\pm\pm(\pm\pm(M)), \pm\pm(\pm\pm\pm(M))) \rightarrow_R \mu M \cdot \inf(\pm\pm(M), \pm\pm\pm(M)) \]
\[ \rightarrow_R \mu M \cdot \inf(\pm\pm(M), \pm\pm\pm(M)) \rightarrow_R \mu M \cdot \inf(\pm\pm\pm(M), \pm\pm\pm(\pm\pm(M))) \rightarrow_R \mu M \cdot \inf(\pm\pm\pm(M), \pm\pm\pm\pm(M)) \rightarrow_R \inf(\mu M, \mu M) \rightarrow_R \inf(\text{src}(\infty), \text{src}(\infty)) \rightarrow_R \text{src}(\infty) \]

Figure 15: Output of our productivity decision tool, establishing productivity of Definition (7). Here \( \mu \) symbolizes recursion, the pebble \( \bullet \) an abstract element, \( \inf \) the infimum, \( \{-, +\} \) code input and output, and overlining means periodic iteration. The actual computation uses a confluent and terminating rewrite relation \( \rightarrow_R \).

\[ [J] = \mu J \cdot (\pm\pm(\pm\pm(J))) \rightarrow_R \mu J \cdot \pm\pm(\pm\pm(\pm\pm(\pm\pm(J)))) \rightarrow_R \mu J \cdot \pm\pm(\pm\pm(\pm\pm(J))) \rightarrow_R \mu J \cdot \pm\pm(\pm\pm(J)) \rightarrow_R \mu J \cdot \pm\pm(\pm\pm(J)) \rightarrow_R \text{src}(4) \]

Figure 16: For the specification (8) we obtain that \( J \) is not productive; only 4 elements can be evaluated.

automatically, many powerful techniques and tools have been developed, to wit AProVE, Jambox, Matchbox, MuTerm, Torpa, TTT, etc.

We have implemented our data-oblivious techniques [23, 24, 20] in the productivity prover ProPro (http://infinity.few.vu.nl/productivity/). Figure 15 gives a script of the tool’s computation for specification (7) of the Thue-Morse sequence, testifying that its definition outputs \( \infty \) entries, and therefore is productive. Applied on (8) the answer is 4 (see Figure 16), hence not productive.

8 More Questions

(i) The productivity question for stream definitions has everything to do with the input-output behavior of the various operations on streams (like zip). Thus a stream specification is reminiscent to a system of communicating
Figure 17: Comparing the ‘fingerprints’ $\Delta^\omega(S)$ and $\Delta^\omega(W)$ of the Sierpiński sequence $S$ (left), and the Mephisto Waltz $W$. We find that $\Delta^2(S) = \Delta^3(W)$!

processes. There are tools (like $\mu$-CRL) that analyze deadlock-freeness of process specifications. Can the stream productivity question profit from the tools technology for process communication? We expect that it can.

(ii) Connections with $\lambda$-calculus and infinitary $\lambda$-calculus are interesting. It is clear that stream operations like zip, even, etc., as expressible in term rewriting, are definable in $\lambda$-calculus. The resulting $\lambda$-term $M$ that is the translation of the stream specification then has a Böhm Tree $BT(M)$ that is precisely the stream to be defined. It is total (without bottoms $\bot$) iff the stream specification is productive. In fact, we may call $\lambda$-term $M$ with total $BT(M)$ productive. The study of productive $\lambda$-terms seems worthwhile in itself.

(iii) What streams give rise to fractals? As we have seen, some streams give rise, with suitable ‘turtle’ instructions, to well-defined fractals such as the snowflake, the arrowhead curve, the dragon curve, the Cesàro fractal, and so on. But other streams, like Kolakoski, seem to generated chaotic patterns that do not converge (in the Hausdorff metric, after re-scaling successive approximations) to a well-defined fractal curve. Are there criteria (and definitions for what constitutes a ‘well-defined fractal’) that guarantee the convergence
via some smart turtle to a fractal?

(iv) FSTs are highly useful in many fields. An interesting question is whether they can be decomposed in some elementary, ‘prime’, FSTs (with respect to the composition of FSTs, which is intuitively clear, feeding the output of the first FST as input of the second; this is a wreath product). A theory that was discovered in the sixties by Rhodes and Krohn seems to indicate that indeed there is a prime factorization theorem for FSTs. For Mealy Machines a prime factorization was obtained by Rhodes and Krohn, see [4]. Generalizing this to the more general FSTs should be a corollary which we expect to be straightforward (but not entirely trivial).

(v) Fingerprints of streams over \{0, 1\}. We wonder what information about the complexity of streams can be derived from their ‘fingerprints’. What we record in the fingerprint is the stream of first differences (as realized by the FST \(\Delta\) depicted in Figure 6), and this repeated ad infinitum. Being twodimensional, the black and white patterns that arise in this way, are much easier ‘processed’ than the ‘patterns’ in the original stream, the top row. In this way, we observed [28, 29] the relation between the Sierpiński stream \(S\) and the Mephisto Waltz \(W\) (both sequences are defined in Section 3), see Figure 17. For the Fibonacci word \(F\), a sturmian stream, the fingerprint reveals patterns of a rather ‘quiet nature’. Is this always so for sturmian streams? Note that looking ‘deep’ in the fingerprint, amounts to looking far away to the right in the stream.

9 Background: Infinitary Term Rewriting

We give a brief introduction to infinitary rewriting. Actually, what we present here is only a prefix of a more extensive theory in which reductions can have any countable ordinal length. Here we will be satisfied with reductions up to length \(\omega\). For the full framework of infinitary rewriting we refer to [37, 53, 7, 30], for an introduction to finitary rewriting to [36, 53, 5, 6].

A signature \(\Sigma\) is a set of symbols \(f\) each having a fixed arity \(#f\in \mathbb{N}\). (Earlier we used \(\Sigma\) to denote an alphabet of letters; we now overload the use of \(\Sigma\) somewhat, but this will not cause confusion.) Let \(X\) be an infinite set of variables such that \(X\cap \Sigma = \emptyset\). Then the set of finite terms \(\text{Ter}(\Sigma, X)\) over \(\Sigma\) and \(X\) is inductively defined by the grammar:

\[
T ::= x \mid f(T, \ldots, T)_{\text{#f times}} \quad (x \in X, f \in \Sigma)
\]  

(9)
We obtain the set of (finite and) infinite terms \( \text{Ter}^\infty(\Sigma, X) \) by interpreting this grammar coinductively. That is, \( \text{Ter}^\infty(\Sigma, X) \) is the largest set \( T \) such that every term \( t \in T \) is either a variable \( t \in X \), or \( t = f(t_1, \ldots, t_n) \) with \( n = \#f \) and \( t_1, \ldots, t_n \in T \).

The equality on infinite terms is bisimilarity \( \Rightarrow \subseteq \text{Ter}^\infty(\Sigma, X) \times \text{Ter}^\infty(\Sigma, X) \) which is defined as the largest relation \( R \) such that \( s \ R \ t \) implies that \( s = t \in X \), or \( s = f(s_1, \ldots, s_n) \) and \( t = f(t_1, \ldots, t_n) \) such that \( s_1 \ R \ t_1, \ldots, s_n \ R \ t_n \). We consider bisimilar terms \( s \ Leftrightarrow t \) as identical.

**Remark 9.1.** Alternatively, the infinite terms arise from the set of finite terms, \( \text{Ter}(\Sigma, X) \), by metric completion, using the well-known distance function \( d \) such that for \( t, s \in \text{Ter}(\Sigma, X) \), \( d(t, s) = 2^{-n} \) if the \( n \)-th level of the terms \( t, s \) (viewed as labeled trees) is the first level where a difference appears, in case \( t \) and \( s \) are not identical; furthermore, \( d(t, t) = 0 \). It is standard that this construction yields \( \langle \text{Ter}(\Sigma, X), d \rangle \) as a metric space. Now infinite terms are obtained by taking the completion of this metric space, and they are represented by infinite trees. We will refer to the complete metric space arising in this way as \( \langle \text{Ter}^\infty(\Sigma, X), d \rangle \), where \( \text{Ter}^\infty(\Sigma, X) \) is the set of finite and infinite terms over \( \Sigma \).

Let \( t \in \text{Ter}^\infty(\Sigma, X) \) be a finite or infinite term. The set of variables \( \text{Var}(t) \subseteq X \) of \( t \), and the set of positions \( \text{Pos}(t) \subseteq \mathbb{N}^* \) of \( t \) are defined coinductively by:

\[
\begin{align*}
\text{Var}(x) &= \{x\} & \text{Var}(f(t_1, \ldots, t_n)) &= \text{Var}(t_1) \cup \ldots \cup \text{Var}(t_n) \\
\text{Pos}(x) &= \{\epsilon\} & \text{Pos}(f(t_1, \ldots, t_n)) &= \{\epsilon\} \cup \{i \epsilon \mid 1 \leq i \leq n, \ p \in \text{Pos}(t_i)\}
\end{align*}
\]

For \( p \in \text{Pos}(t) \), the subterm \( t|_p \) of \( t \) at position \( p \) is defined by:

\[
|_p = t \quad f(t_1, \ldots, t_n)|_p = t_i|_p
\]

A substitution \( \sigma \) is a map \( \sigma : X \rightarrow \text{Ter}^\infty(\Sigma, X) \). We extend the domain of substitutions \( \sigma \) to terms \( \text{Ter}^\infty(\Sigma, X) \) as follows:

\[
\sigma(f(t_1, \ldots, t_n)) = f(\sigma(t_1), \ldots, \sigma(t_n))
\]

We write \( \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\} \) for the substitution \( \sigma \) defined by \( \sigma(x_1) = t_1, \ldots, \sigma(x_n) = t_n \) and \( \sigma(y) = y \) for every \( y \in X \setminus \{x_1, \ldots, x_n\} \).

A context \( C \) is a term \( \text{Ter}^\infty(\Sigma, X \cup \{\square\}) \) containing precisely one occurrence of \( \square \), that is, there is precisely one position \( p \in \text{Pos}(C) \) such that \( C|_p = \square \). For a context \( C \) and a term \( t \), we write \( C[t] \) for the term \( \{\square \mapsto t\}(C) \).

A rewrite rule \( \ell \rightarrow r \) over \( \Sigma \) and \( X \) is a pair \( (\ell, r) \in \text{Ter}(\Sigma, X) \times \text{Ter}(\Sigma, X) \) of finite terms such that the left-hand side \( \ell \) is not a variable \( (\ell \notin X) \), and all variables in the right-hand side \( r \) occur in \( \ell \) \((\text{Var}(r) \subseteq \text{Var}(\ell))\). A term rewrite system (TRS) \( \mathcal{R} \) over \( \Sigma \) and \( X \) is a set of rewrite rules over \( \Sigma \) and \( X \). We define a binary relation \( \rightarrow_{\mathcal{R}} \), the rewrite steps, to consist of all pairs

\[
C[\sigma(\ell)] \rightarrow_{\mathcal{R}} C[\sigma(r)]
\]
for contexts $C$, rule $\ell \rightarrow r \in \mathcal{R}$, and substitution $\sigma : X \rightarrow \text{Ter}^\omega(\Sigma, X)$. Furthermore, we write $\rightarrow_{\mathcal{R}, p}$ whenever additionally $C|_p = \Box$. We drop the subscript $\mathcal{R}$ in $\rightarrow_{\mathcal{R}}$ and $\rightarrow_{\mathcal{R}, p}$ whenever $\mathcal{R}$ is clear from the context. The notion of normal form, which now may be an infinite term, is unproblematic: it is a term without a redex occurrence. A finite rewrite sequence from $s$ to $t$, denoted $s \rightarrow t$, is a sequence $s = s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_n = t$.

An infinitary rewrite sequence from $s$ to $t$, denoted $s \rightarrow\rightarrow t$, is either a finite rewrite sequence $s \rightarrow t$, or an infinite sequence $s_0 \rightarrow p_0 \Rightarrow s_1 \rightarrow p_1 \Rightarrow s_2 \Rightarrow p_2 \ldots$ of rewrite steps such that the following conditions hold:

(i) the distance $d(s_i, t)$ tends to 0 for $i \rightarrow \infty$ and, moreover,

(ii) the depth of the rewrite action, that is, the length of the position $p_i$, tends to infinity for $i \rightarrow \infty$.

Note that item (i) requires Cauchy convergence of the sequence of terms. The requirement (ii) is strong convergence, which in addition requires that the depth of the redexes contracted in the successive steps tends to infinity. So this rules out the possibility that the action of redex contraction stays confined at the top, or stagnates at some finite level of depth.

Example 9.2. Let us give a simple example of infinitary term rewriting that moreover explains the working of an FST on a stream. Consider the FST in Figure 6. We ‘translate’ this FST in the TRS with rules:

\[
\begin{align*}
q(0 : s) & \rightarrow q_0(s) \\
q(1 : s) & \rightarrow q_1(s) \\
q_0(0 : s) & \rightarrow 0 : q_0(s) \\
q_0(1 : s) & \rightarrow 1 : q_1(s) \\
q_1(0 : s) & \rightarrow 1 : q_0(s) \\
q_1(1 : s) & \rightarrow 0 : q_1(s)
\end{align*}
\]

Here 0, 1 are 0-ary constants, $q, q_0, q_1$ are unary function symbols, ‘:’ is a binary stream constructor, $s$ is a variable (for streams).

Now the transformation of the Thue-Morse sequence $M$ to the Toeplitz word $T = T_{101?}$ (see Section 2.3) proceeds by the following infinite reduction sequence, where we omit the infix ‘:’ symbols:

\[
q(M) = q(0110100110010110\ldots) \\
\rightarrow q_0(110100110010110\ldots) \\
\rightarrow 1 q_1(10100110010110\ldots) \\
\rightarrow 10 q_1(0100110010110\ldots) \\
\rightarrow 101 q_0(100110010110\ldots) \\
\rightarrow 1011 q_1(00110010110\ldots) \\
\rightarrow\rightarrow T
\]

The term $T$ is an infinite normal form. Note that indeed the depth of the contracted redexes tends to infinity during the reduction.
It is not hard to see that we can easily extend such rewrite sequences to length beyond ω; e.g. \( q(q(M)) \rightarrow q(T) \rightarrow 11001 \ldots \), which yields an infinite normal form after \( ω \cdot 2 \) steps. In Figure 17 we already considered the ‘fingerprint’ of a stream \( s \), being the matrix with rows \( q^n(s) \), or in our earlier notation \( Δ^n(s) \).

**Remark 9.3.** Without wanting to go deep in the theory of infinitary rewriting, let us mention some facts. There are infinitary counterparts \( \text{CR}^∞ \), \( \text{UN}^∞ \), \( \text{WN}^∞ \) and \( \text{SN}^∞ \) for the well-known finitary properties \( \text{CR} \) (confluence or Church–Rosser property), \( \text{UN} \) (unique normal form property), \( \text{WN} \) (weak normalization), and \( \text{SN} \) (strong normalization). In general \( \text{CR}^∞ \) does not hold for first-order orthogonal term rewriting systems, but for \( \text{TRSs} \) as in the example, without collapsing rules, it does. Moreover, \( \text{UN}^∞ \) always holds for first-order orthogonal \( \text{TRSs} \). For the example \( \text{TRS} \) also \( \text{WN}^∞ \), \( \text{SN}^∞ \) and \( \text{UN}^∞ \) hold.

**References**


The design of efficient algorithms for planar graphs, as a field of research, is over forty year old and continues to be an exciting area. There are several new efficient algorithms for a variety of graph optimization problems that exploit planarity and, in general, the structure of low genus graphs and graphs with excluded minors.

Efficient space-bounded computations, on the other hand, involve a different genre of algorithmic ideas and exploit the planar graph structure somewhat differently. There has been a flurry of interesting research on this topic in last decade. In this timely and well-written article, Samir Datta and Raghav Kuakarni focus on planar graphs in the context of space complexity. They discuss several properties of planar graphs and their precise role in the design of space efficient algorithms.
1 Introduction

The purpose of this article is to survey several useful properties of planar graphs that can be exploited specifically in the context of space bounded computation to obtain efficient algorithms. For completeness we also point out some situations where planar restrictions remain computationally as hard as general graphs.

The classes we will encounter in this survey are mostly variants of deterministic logarithmic space, $L$. They include non-deterministic logarithmic space, $NL$, Unambiguous Log-space $UL$, Stoic Probabilistic Log-space $SPL$ and counting classes like $\text{GapL}$, $\oplus \text{L}$. For a definition of these classes see e.g. [32]. We will also have occasion to compute functions in Log-space. We say that a function is computable in Log-space if there is a Log-space transducer, i.e. a deterministic logarithmic space machine with a write-only one-way output tape on which it writes the function value.

**Definition 1.** A graph is said to be planar if there is a mapping of the vertices on the plane such that the edges can be drawn as curves in the plane, which do not intersect except at their end points.

We are going to make repeated use of the following algorithmic version of planarity:

**Theorem 2** (Allender,Mahajan[3],see also [15]). There is a Log-space algorithm for constructing an embedding of a planar graph on the plane.

We will also make use of the notion of grid embedding of a graph. This is a map which identifies a graph with a subgraph of a complete rectangular grid (i.e. a rectangular array of nodes in which a node is connected to the vertices immediately next to it horizontally and vertically). This is done by a degree reduction...
step, (by replacing a vertex with a binary tree or a cycle) followed by embedding the vertices so that that the edges of the degree reduced graph get elongated into paths. The crucial point is that this preserves some property of the digraph or graph e.g. reachability [2] or number of perfect matchings [10] and runs in Log-space.

In the rest of the paper, in every section, we mention a property (or a collection of properties) of planar graphs and then give a brief description of how the property helps us develop a bounded space algorithm or hardness result. We should mention at the outset that this survey does not aim at completeness of any sort - it aims at providing a taste of the planarity magic through a number of examples drawn according to our preferences and experience.

2 Unique Embedding

The first property of planar graphs that we will exploit is the property of unique embeddability of 3-connected planar graphs, in the plane. This remarkable property which was first shown by Whitney, way back in 1933, allows us to give a bounded space algorithm canonical description for such highly connected planar graphs. We should point out it is notoriously difficult to prove good upper bounds or lower bounds for the general graph isomorphism problem. In contrast, planarity enables us to prove matching upper and lower space complexity bounds.

The first bounded space algorithm that made use of Whitney’s theorem for planar isomorphism was by Thierauf, Wagner [29] based on distance computation in planar graphs. This was soon followed up by an algorithm [13] (which we describe below) that achieved the optimal Log-space bound by replacing the distance computation by an exploration from Reingold’s seminal paper[25]. The result was later extended [14] to arbitrary planar graphs through a careful application of Lindell’s Log-space Tree Canonization algorithm to the tree decomposition of the given planar graph into its 2-connected and 3-connected components.

Whitney’s Unique Embeddability Theorem

We will make crucial use of the following theorem by Whitney

**Theorem 3.** (Whitney[33]) A 3-connected\(^1\) planar graph has exactly one combinatorial embedding on the plane (up to reflection).

The theorem can be rephrased as: in any planar drawing - the cyclic order of the edges leaving a vertex in, say, the clockwise order, is identical (except,  

\(^1\)removing any two vertices from the graph does not disconnect it
of course that we may reflect the graph and get another one in which the order of edges is reversed). As an example consider the 3-connected planar graph Figure 2 (a) with its two embeddings. Figure 2 (b) illustrates the necessity of 3-connectivity for unique embeddability.

**Unique Embedding and 3-connected Planar Graph Isomorphism**

Given a planar graph, we will compute, using a Log-space transducer, a binary string that is independent of the presentation of the graph (in terms of the vertex and edge names) and uniquely characterizes the graph. Thus, if \( \text{can}(G) \) denotes the canonical string for \( G \), \( G_1 \cong G_2 \) if and only if, \( \text{can}(G_1) = \text{can}(G_2) \). Testing for isomorphism is then easy - just go through the canonical descriptions bit by bit and verify that they match.

The rough idea behind the algorithm is to construct a canonical ordered spanning tree - a rooted tree that does not depend on the presentation of the graph and where the children of a node are ordered. Now it is easy to describe the graph in terms of the tree. First the vertices of the graph can be canonically labeled by the pre-order numbering of the tree. Next the edges can be listed in lexicographic order of the numbers of their endpoints (with, say, the smaller endpoint first).

Thus we have to show how to produce a canonical spanning tree. We show something slightly weaker, that is, we produce a collection of spanning trees which depends only on the choice of the “root edge” and the choice of one of the two embeddings. We can “canonize” the graph relative to each of the trees as described above and then pick the minimum description as our final canon.

For simplicity assume that the graph has degree exactly 3 - this is easy to ensure by a simple degree reduction step.

We will need to introduce the concept of universal exploration sequence UXS to describe how to obtain the tree above. A UXS \( \sigma_n \) is a string (of length polynomial in \( n \)) from \( \{0, 1, 2\}^* \) which encodes a way to explore a cubic graph which is
equipped with a cyclic ordering on the neighbours of each vertex. We will explore
the graph one edge at a time. At the $i^{th}$-step we will be sitting on an (directed)
edge $(u, v)$, having just visited $u$ and being about to visit $v$. The next edge will be
some $(v, w)$ depending on the letter $\sigma_n[i]$ which tells us the offset of the next edge
relative to the current one: an offset of zero means $w = u$ and offsets 1, 2 indicate
the first/second neighbours of $v$ after $u$ in the prescribed cyclic order. Some-
what mysteriously the universality of $\sigma_n$ implies that given any cubic graph of size
$n$ with an ordering on its vertices, an exploration guided by $\sigma_n$ will lead us to visit
every vertex in the graph. It is easy to construct UXS's using randomization. The
magic of Reingold’s construction [25] is that

- a UXS can in fact be constructed in Log-space!

A planar embedding of a cubic graph naturally induces a cyclic ordering on the
neighbours of a vertex - say the clockwise order. Whitney’s theorem tells us there
are exactly two such cyclic orders. The magic of planarity meets that of Reingold,
at this point. Thus, given a fixed UXS churned out by Reingold’s algorithm and
two choices:

- the directed edge to start the exploration from, and

- one of the two possible embeddings,
there is a fixed way to visit each vertex in the graph. By considering the edge of first visit to a vertex we can produce a spanning tree from this exploration. It is important to note that:

- the tree produced is canonical relative to the two choices above (and a fixed UXS)
- it can be computed in Log-space since a transducer can output the parent edge of each vertex by repeating the exploration multiple times

This is all that is needed to prove the following:

**Theorem 4.** *Isomorphism in 3-connected planar graphs can be tested in \( L \).*

## 3 Duality and Flows in Planar Graphs

The next property which we will consider is duality. For this we need to define the dual of a graph \( G \). Informally, the nodes of the dual consist of the faces of \( G \) and two nodes are joined by an edge whenever the corresponding faces share an edge. We use this correspondence crucially in our next application viz. computing flows in planar networks.

We consider the problem of determining if there is a feasible flow in a network where there are both demands at nodes (i.e. the amount of flow that the vertex needs - this may be positive or negative) and capacity constraints at edges (i.e. the maximum amount of flow possible through an edge). Notice that the graph is bidirected i.e. there is an edge in either direction for every undirected edge and each of these edges have capacities of their own.

Many graph theory problems can be recast in this framework, e.g. bipartite matching. We will of course restrict our attention to the case when the network is planar. The approach we are going to delineate is based on a paper by Miller and Naor [24] which proposes a parallel algorithm for the problem.

**Cut-Cycle duality in Planar Graphs and Miller-Naor’s Algorithm**

We first reduce the problem with edge capacities and vertex demands to one in which the vertex demands are all zero and edge capacities are appropriately modified. Zero demands imply that the flow across any cut is zero. Thus if the sum of capacities of edges in any directed cut are strictly negative we have a contradiction to the actual flow being zero i.e. the flow is infeasible. It is not too hard to prove that this sufficient condition for infeasibility is also necessary. But finding
if a graph has negative cuts seems to be a hard problem. We can of course reduce the problem to finding a minimal cut\(^2\) that is negative.

This is where the planarity magic comes into play:

- Minimal cuts in the primal graph correspond to cycles in the dual.

Thus with appropriately weighed dual edges, the problem reduces to detecting a negative cycle in the dual. But there exists a negative cycle in a graph if and only if there exists a negative closed walk. And detecting the existence of a negative closed walk is easy in NL - just start from a vertex keeping non-deterministically walk along the graph keeping track of only the current vertex and the sum of weights encountered so far (along with the initial vertex). If the initial vertex is reached with a negative sum within polynomial number of steps then accept else reject.

The idea can be refined to find the flow if indeed the dual graph has no negative cycles. The intuitive idea being that since there are no negative cycles shortest distance between any pair of dual nodes is well defined. Miller and Naor [24] show these dual shortest distances can in fact be converted to a primal feasible flow.

**Space Complexity of Miller-Naor’s algorithm**

The basic ingredients needed in the proof are:

- a planar embedding of the graph
- a spanning tree algorithm (for reduction to the zero-demand case)
- an algorithm to detect a negative cycle in a (planar) digraph
- a directed shortest path algorithm in a (planar) digraph with no negative cycles

The first two are known to be in deterministic Log-space through non-trivial algorithms [3, 15, 25] while the latter two are easily seen to be reducible to reachability i.e. in NL.

### 4 Duality and Deterministic Isolation in Planar Graphs

Mulmuley, Vazirani and Vazirani’s isolation lemma informally states that a uniform random assignment of small weights to the elements of a universe makes the

\(^2\)a minimal set of edges removing which disconnects the graph
weight of the minimum weight set of any given set system, unique. Derandomizing the lemma in general is not possible [1] but for several set systems arising from planar graphs, there exists a simple deterministic weighing scheme that isolates the min weight set [2, 6, 12]. This deterministic isolation can then be translated to a space efficient algorithm for finding the min-weight set using known extraction procedures like [26, 4]. Our focus in this article will be on the isolation rather than the extraction part of the algorithm, since it is the former that makes crucial use of planarity.

In describing these algorithms it is often convenient for the purposes of visualization to first embed the graph in a 2-d rectangular grid, after a degree reduction step. This is how the algorithms [6, 12] historically came about. But it is possible to work directly on the graph (and its dual) rather than invoke grid embedding [28, 8]. In this survey we will stick to grid embedding for the ease of exposition.

**Deterministic Isolation in Planar Structures**

We will consider two important set systems in this article:

- the system of directed \( s,t \)-paths in a grid-digraph \( G \), where \( s,t \) are vertices of \( G \)
- the system of perfect matchings in a grid-graph \( G \).

A path or matching will be regarded as a set of edges.

We want to prescribe a weighing scheme for the grid edges such that the min-weight set is unique. For technical reasons we will insist that the weighing scheme in the first case is skew-symmetric i.e. reversing an edge just flips the sign of the weight.

By definition, if we have two distinct min-weight sets \( S_1, S_2 \) then their weights must be equal. Let us first consider the case of \( s,t \)-reachability. If we reverse the edges of, say \( S_2 \) we get a closed walk of zero weight. An easy substitution argument shows that every simple cycle \( C \) contained in this closed walk must have zero total weight.

We follow a similar argument in the case of perfect matchings. The symmetric difference \( S_1 \oplus S_2 \), of two perfect matchings is exactly a collection of disjoint cycles. Thus an easy substitution argument again shows that the difference \( w(S_1 \cap C) - w(S_2 \cap C) = 0 \) for any of these cycles \( C \). Because there is strict alternation between edges of \( S_1, S_2 \) in \( C \) this is equivalent to saying that the alternating sum of each cycle \( C \) is zero.

As a notational convenience we have the following definitions of *circulation* for reachability and matching. Let circulation of a simple cycle be
The Bulletin of the EATCS

- the sum of weights of the (directed) edges of the cycle in case of reachability
- the alternating sum (which is well-defined up to its sign) of weights of an undirected simple cycle in the case of matching

Thus the two paragraphs prior to the definition can be summarized as: distinct min-weight sets imply zero circulation for some simple cycle. Contrapositively, if we can produce a weighing scheme such that all simple cycles in the complete grid have non-zero circulations then the min-weight set for any grid graph will be unique. We now proceed to construct such a weighing scheme.

In order to do so we again invoke planarity magic and ensure that:

- the circulation of a (simple) cycle equal in magnitude to the area enclosed by the cycle.

This is easy to argue for reachability, suppose we could assign skew symmetric weights to the edges of a unit square such that they sum to 1 in the clockwise orientation. Then if we consider two adjacent unit squares both oriented clockwise, skew symmetry ensures that their circulations adds up to 2 since the common edge cancels out. Inductively, it is easy to see that any clockwise cycle will get a circulation equal to its area (and counter-clockwise cycle will have a circulation equal to the negative of its area).

By a somewhat more complicated argument we can ensure that the (matching) circulation of a cycle in the grid equals the signed sum of circulations of the squares enclosed by it. The sign of square being a quantity that flips between adjacent squares and the bipartiteness of the dual grid permits us such a function. Now we need to make sure that the circulation of every square equals its sign which ensures that the circulation of a simple cycle equals its signed area.

Two simple sets of linear equations reveal what assignments of weights will work - we illustrate this for perfect matchings in Figure 4.

Isolation and Extraction in Planar Graphs

We have sketched the proof of the following result by Bourke, Tewari and Vinodchandran [6]: Let $G$ be a sub-digraph of the $n \times n$ square grid. There is a constant $c$ and a Log-space computable weighing function $w : E \to [-cn, cn]$ under which the min-weight $s, t$-path is unique (if it exists) for any two vertices $s, t \in V(G)$. As a consequence, due to the extraction procedure in Reinhardt and Allender [26], $s, t$-reachability in planar digraphs is in $UL \cap \text{co-UL}$.

The following result by Datta, Kulkarni and Roy [12] mirrors the previous one for planar bipartite perfect matchings: Let $G$ be a sub-graph of the $n \times n$ square grid. There is a constant $c'$ Log-space computable weighing function $w'$:
Figure 3: The weighing scheme and signs for isolating perfect matching in grid graphs

\[ E \to [-c'n, c'n] \] under which the min-weight perfect matching of \( G \) is unique (if it exists). As a consequence, due to the extraction procedure in Allender, Reinhardt and Zhou [4], finding a perfect matching in planar bipartite graphs is in SPL.

5 Bounded Local Tree Width

The concept of tree-width of graphs has turned out to be very useful for designing efficient algorithms. See for instance [17]. Although the tree-width of planar graphs could be arbitrarily large, they enjoy a weaker property that is equally useful, namely that of having bounded local tree width, i.e., if we do breadth first search (BFS) on a planar graph starting from any given vertex then the tree-width of any constant number of BFS layers is bounded above by a constant. This weaker property suffices to obtain fast algorithms for several problems.

Baker’s Algorithm

In a well-known paper [5], Baker first exploited the bounded local tree width property of planar graphs to give an approximation scheme \((1 + \epsilon)\) approximation algorithm for every \(\epsilon > 0\) for a variety of problems (for instance Max-Cut) that are hard to approximate in general graphs. The basic idea of Baker’s algorithm is as follows:

- (Step 1) Remove some edges from the graph so that:
  - the optimum solution is not affected by much: say it is affected only by \(\epsilon\) fraction
The rest of the edges can be partitioned into disjoint union of graphs whose tree-width is a constant that depends on the approximation guarantee, i.e., $\epsilon$.

- (Step 2) Solve the problem on the bounded tree width graphs using dynamic programming and combine the solution on the disjoint pieces to obtain a solution in the original graph.

**Space Complexity of Baker’s Algorithm**

In fact, for Step 1, Baker observes that the vertices at any fixed distance from the root form an outer-planar graph. Thus any $k$ consecutive layers constitute $k$-outer-planar graphs, which happen to have tree-width at most $O(k)$.

Thus: the efficiency of Baker’s algorithm relies mainly on the following two components:

- (Component 1) distance computation in planar graphs,
- (Component 2) exact solution on bounded tree-width graphs.

The first component is crucial for obtaining the BFS layers, which in turn help in obtaining the desired decomposition of the planar graph into graphs of bounded tree width. The second component gives an efficient algorithm for solving the problem exactly on the bounded tree width graphs. This is achieved via a dynamic programming approach. The bounded tree-width property guarantees the efficiency of the dynamic programming.

The exact solution on bounded tree-width graphs more or less directly combine to give an approximate solution in the original graph. In the end, one has to argue that the solution obtained after Step 2 is good enough, i.e., the removed edges do not affect the optimal solution by much. This is guaranteed by the delicate choice of edges in Step 1.

In the context of the space complexity, Datta and Kulkarni [9] observe that both these components are less expensive than their counterparts in general graphs!

For the first component, we can use an elegant result by Bourke, Tewari, and Vinodchandran [6] showing that the directed planar reachability is in $\mathsf{UL} \cap \mathsf{co-UL}$ as opposed to the directed reachability in general graphs being $\mathsf{NL}$-complete. As a consequence, BFS in planar graphs can be performed in $\mathsf{UL} \cap \mathsf{co-UL}$. The exact space complexity of BFS in planar graphs is still not known. It is still possible that in fact BFS in planar graphs could be performed as efficiently as Log-space!

The current best known upper bound in this context is (deterministic) $O(\log^2 n)$ space, which follows from the famous Savitch’s Theorem [27] that the class $\mathsf{NL}$ is...
contained in $O(\log^3 n)$ space. This means that the decomposition of planar graphs into bounded tree width graphs can performed in a space efficient way.

As for the second component in Baker’s algorithm, a recent theorem of Elberfeld et. al. [17] comes to our help. They show that any optimization problem expressible in monadic second order logic can be solved in Log-space on bounded tree width graphs. It turns out that many optimization problems (e.g. Max-Cut) can be expressed in MSO. Thus the second component of Baker’s algorithm can be made space efficient for problems like Max-Cut.

This simple combination gives space efficient algorithms for some optimization problems in planar graphs. The best bound on space complexity that one can currently obtain via this approach is $O(\log^2 n)$ space. It is an interesting open question whether this space bound can be further improved possibly using some other type of decomposition. For instance: does Max-Cut in planar graph have a Log-space approximation scheme?

## 6 Algebraic Properties

The adjacency matrix of an undirected graph $G$ on $n$ vertices is an $n \times n$ symmetric matrix $A$, i.e., $A(i, j) = A(j, i)$ and whose entries are $A(i, j) = 1$ if and only if $(i, j)$ is an edge in $G$; $A(i, j) = 0$ otherwise. Similarly one can define the adjacency matrix of a weighted as well as directed graphs. Let $D$ denote the diagonal matrix whose $i^{th}$ diagonal entry is the degree of the vertex $i$. The Laplacian matrix of an undirected graph is $L := D - A$.

Suppose we have an undirected graph $G$ together with a fixed orientation of its edges, i.e., every undirected edge $(i, j)$ of $G$ is oriented either $(i, j)$ or $(j, i)$. The oriented matrix $M$ of $G$ with respect to a fixed orientation is a skew symmetric matrix, i.e., $M(i, j) = -M(j, i)$ such that $M(i, j) = 0$ if $(i, j)$ is not an edge in $G$; otherwise it is equal to $1$ if $(i, j)$ is the orientation of the edge; otherwise it is equal to $-1$. The matrices $A, L,$ and $M$ hold some crucial information about graphs. In the context of planar graphs the algebraic properties of these matrices can be exploited to obtain space efficient algorithms.

### Counting Perfect Matchings in Planar Graphs

An orientation of an undirected graph is called Pfaffian if the determinant of the skew symmetric matrix $M$ with respect to the orientation is exactly equal to the square of the number of perfect matchings in the graph. Thus immediately suggests the following approach for counting perfect matchings in a graph that has a Pfaffian Orientation.

- obtain a Pfaffian Orientation via an efficient algorithm
compute the determinant of the skew symmetric matrix $M$ associated to the Pfaffian Orientation; the square root of this determinant is the # perfect matchings.

In this context, planar graphs have been extremely lucky. Around 1965, Kaste lyn [21] discovered that the planar graphs are endowed with an amazing property:

Theorem 5 (Kastelyn). Every planar graph has a Pfaffian Orientation.

In fact, Kastelyn discovered a simple and elegant criteria for a planar graph to have a Pfaffian Orientation. This criteria has lead to efficient algorithms for obtaining a Pfaffian Orientation. We need a definition for further explanation. We call a cycle oddly oriented if we traverse the cycle in the clockwise direction then we encounter odd number of edges oriented in the direction of traversal. Kastelyn showed the following:

- an orientation of a planar graph is Pfaffian if every simple cycle is oddly oriented.
- furthermore: there is an oddly oriented simple cycle if and only if there is an oddly oriented face of the planar graph, i.e., facial cycle

Thus to construct a Pfaffian orientation it suffices to give the orientation to the edges so that every face is oddly oriented. This local criteria for constructing the Pfaffian Orientation turns out to be useful in the context of space complexity. In fact, Mahajan and Vinay [23] observe that using parity tree evaluation techniques one can obtain a Log-space algorithm for computing such an orientation! A consequence of having a Log-space computable Pfaffian Orientation is that one can compute the number of perfect matchings in planar graph in the class NC, and hence in poly-logarithmic space. To note the contrast, recall that the problem of counting perfect matchings in general graphs is known to be #P-hard by the classic result of Valiant [30].

Counting Spanning Trees modulo $2^k$ in Planar Graphs

Another setting where planarity works like magic is that of modular counting of spanning trees. In particular, Braverman, Kulkarni, and Roy [7] show that for any fixed $k$, computing the number of spanning trees modulo $2^k$ in planar graphs can be done in Log-space! To show the contrast, we note that same problem in general graphs is known to be $\oplus L$-complete [7], i.e., as hard as computing the determinant of an integer matrix modulo 2.

This result uses two interesting connections.
connection of certain type of knots to the co-rank of Laplacian matrix (mod 2)

the Matrix-Tree Theorem that connects # spanning trees to the determinant of a minor of the Laplacian

First we give some explanation on the first connection: There are certain type of knots that are called left-right walks (cf. Godsil and Royle [19]) that one can associate to a planar graph. These knots can be constructed in Log-space. The number of these knots is exactly equal to the co-rank of the Laplacian matrix (modulo 2) of the planar graph. Counting the number of these knots can be done by a simple reduction to undirected connectivity problem, which is known to be in Log-space; thanks to the celebrated result of Reingold [25]. Now the magic is handed over to the Kirchhoff’s Matrix-Tree Theorem:

**Theorem 6** (Kirchhoff). Let $L$ be the Laplacian matrix of graph $G$ and let $L'$ denote the matrix obtained from $L$ by deleting the first row and the first column. Then: determinant of $L'$ is exactly equal to the number of spanning trees in $G$.

Thus the number of spanning trees is odd if and only if determinant of $L'$ modulo 2 is odd. This happens if and only if the co-rank of $L$ is at most 1. This immediately gives a Log-space algorithm for counting (modulo 2) the spanning trees in planar graphs. In fact, Braverman, Kulkarni, and Roy extend this result to mod $2^k$ setting for any fixed $k$ using somewhat sophisticated techniques from Algebraic Topology.

7 Planarizing Gadgets

Every graph can be drawn on plane. One can represent the nodes of the graph by some points in the plane and one can represent the edges of the graph by curves joining the points. We say that two edges cross if the curves representing the two edges intersect in a point other than the end points of the edges. A graph is planar if and only if one can draw the graph on plane so that no two edges cross. Hence the only (and the big) difference between planar graphs and arbitrary graphs is that planar graph has no crossings while arbitrary graph might have many crossings.

A planarizing gadgets is loosely speaking a local replacement for crossings to obtain a planar graph. The replacement is done in order to maintain certain properties. Such gadgets can be used in two ways:

- to obtain an efficient algorithm for a problem in general graphs
- to obtain a hardness result for problems planar graphs.

Below we illustrate the two with some examples.
Upper Bounds using Planarizing Gadgets

A perfect example of the power of planarizing gadgets for obtaining upper bounds is given by the framework of so called Holographic Algorithms. This framework was introduced by Valiant [31] and was further developed by Cai et. al. [34]. It exploits the fact that perfect matchings in planar graphs can be computed efficiently, to obtain non-trivial polynomial time algorithms for several seemingly hard problems. The key to this framework is the construction of so called Match-Gates, which are the planarizing gadgets satisfying certain properties. It is a beautiful piece of art to construct such Match-gates, see for instance a survey by Cai [34].

The framework works as follows:

- model a counting problem in general graphs in terms of Match-Gate circuits
- design appropriate Match-Gates and replace the crossings by these Match-Gates

In fact the only computationally expensive step in these algorithms is the computation of determinant. This step can be performed in NC, and hence in poly-logarithmic space. So all these algorithms can be considered space efficient.
Hardness Results using Planarizing Gadgets

One of the main uses of planarizing gadgets is to obtain hardness results. The basic strategy for proving hardness results for problems in planar graphs is as follows:

- start with a hard instance of the problem in general graphs
- replace each crossing by a suitable planar graph such that the solution is preserved under this replacement

Classically several hardness results are known for planar graphs via such gadgets: for instance Hamiltonian Cycle in planar graph is NP-hard [18]. Often times the gadgets are simple once they are discovered but could be challenging to construct. In the context of space bounded computation (and this article) we would like to mention the following:

Datta, Kulkarni, Limaye, and Mahajan [10] study the Determinant and Permanent when restricted to adjacency matrices of planar graphs. They show that planar restrictions of Determinant and Permanent remain as hard as general graph.

Using an interesting use of the gadgets developed by Datta, Kulkarni, Limaye, and Mahajan together with some additional gadgets, Kulkarni [22] shows that the extension of Datta, Kulkarni, and Roy’s [11] method of isolation from bipartite to non-bipartite planar graphs is hard; in the sense that such an extension would imply several strong results such as Bipartite Matching is in NC, NL ⊆ ⊕L, and NP ⊆ ⊕P.

Using the result of Kulkarni [22], recently Datta and Kulkarni [9] show that Max-Cut in planar graphs is NL hard.

Recently there have been some results on proving that certain type of planarizing gadgets do not exist (see for instance [20]).

8 Conclusion and Open Ends

We have shown several examples where studying the space complexity of problems in planar graphs could be fruitful. Several useful properties of planar graphs naturally lead to space efficient algorithms. We list some open questions that could be further explored:

Space Complexity of Recognizing Bounded Genus Graphs

It is known that one can recognize planar graph and in fact construct a combinatorial embedding of it in Log-space [3]. It is still open whether or not one can generalize it for constant-genus graphs. The best known bound is NC [16].
Space Complexity of Reachability in Planar Graphs

Is reachability in directed planar graphs in Log-space? A related question is whether or not the distance computation in undirected planar graphs is in Log-space?

Isolation in Non-bipartite Planar Graphs

Can one extend the method of non-vanishing circulation of Datta, Kulkarni, and Roy [11] to non-bipartite planar graphs? Such an extension would have non-trivial consequences such as $\text{NL} \subseteq \oplus L$ [22].

Space Complexity of Max-Cut in Planar Graphs

We have observed that Max-Cut in planar graphs admits $UL \cap co - UL$ approximation scheme. Furthermore, we have shown that Max-Cut in planar graphs is NL hard. A natural question is whether or not there is a Log-space approximation scheme for Max-Cut in planar graphs.

Space Complexity of Matching in Planar Graphs

Perfect Matching in bipartite planar graphs is known to have space efficient algorithms [11] but non-bipartite case remains open for long time. Kulkarni [22] shows that Minimum Weight Perfect Matching in planar graphs is NL hard even when the weights are 0 or 1. Can one show NL-hardness for Perfect Matching or Maximum Matching in planar graphs? Does Perfect Matching in planar graphs have poly-log space algorithm?

References


The Bulletin of the EATCS