Report on the Dagstuhl-Seminar on

INCREMENTAL COMPUTATION AND DYNAMIC ALGORITHMS

May 2 - 6, 1994

The purpose of the Seminar was to bring two research communities together that have common interests, but that (to date) have had relatively limited contact: (1) theoretical computer scientists who work in the area of “dynamic algorithms”, and (2) programming-language/systems researchers who work in the area of “incremental computing”. The goal of the Seminar was to stimulate intellectual cross-fertilization between the two groups.
For example, theoreticians were exposed to the concerns that arise when making use of incremental processing in practical systems; the systems people were exposed to new algorithms and analysis techniques developed by theoreticians.

For both groups, an important goal is to find efficient algorithms that make use of the solution to one problem instance to find the solution to a ”nearby” problem instance. In the dynamic algorithms community, several avenues of approach have been followed:

1. The design of dynamic combinatorial algorithms, also referred to as ‘update algorithms’ (especially dealing with search and network problems),
2. On-line algorithms and competitive analysis,
3. Parametric algorithms and sensitivity analysis, and
4. The design of adaptive, on-the-fly, and self-stabilizing algorithms in the context of distributed computing.

¿From the programming languages/systems perspective, the motivations for work on incremental computation and dynamic algorithms include concerns such as:

1. Efficient techniques for dealing with “incremental changes” in order to speed up the response times of interactive systems,
2. The creation of languages and systems that provide support for incremental computation, and

3. The design of environments involving cooperating tools. (In such environments, it is paramount that the tools communicate information about *changes* to each other, which raises the possibility for the processing of changes by individual tools to be performed incrementally.)

The Seminar provided the opportunity to present the state-of-the-art in the relevant fields at the high level and to let the two research communities benefit from other's insights. We are grateful to our colleagues in the Seminar for engaging in the endeavour with us and for making the meeting so successful. In all respects the Seminar has been very stimulating and we hope that lasting bonds have been created between the two fields.

Jan van Leeuwen (Utrecht)
Kurt Mehlhorn (Saarbrücken)
Thomas Reps (Madison)

Lecture Schedule

Monday, May 2^{nd}, 1994

Morning session

9.00 - 9.15	**Tom Reps**, **Jan van Leeuwen**	Opening
9.15 - 10.00	**Giuseppe Italiano**	Maintaining Spanning Trees of Small Diameter
10.05 - 10.50	**Raymond Seidel**	New Proofs for the Clarkson-Shor Framework
11.00 - 12.00	**Aswin van den Berg**	Incremental Higher-Order Attribute Evaluation

Afternoon session

15.00 - 16.00	**Maarten Pennings**	Incremental Evaluation of Higher-Order Attribute Grammars
16.05 - 16.50	**John Field**	A Graph Rewriting Approach to Incremental Term Rewriting
17.15 - 18.00	**Kurt Mehlhorn**	Two Problems on Sets and Sequences

Tuesday, May 3^{rd}, 1994

Morning session

9.00 - 9.20	**Susan Graham**	Interactive Computing in Interactive Software Development Environments
9.20 - 9.40	**Michael Harrison**	Incremental Issues in Multimedia Systems
9.45 - 10.30	**Kannan Muthukkaruppan**	Synthesizer for Practical Incremental Evaluators
10.45 - 11.25	**Bill Maddox**	Incremental Semantic Analyzer for Ensemble
11.30 - 12.00	**Emma van der Meulen**	Incremental Implementation for Algebraic Specifications

Afternoon session

15.00 - 15.30	**Sandra Irani**	On-line Algorithms and Restricted Adversaries
15.35 - 16.10	**Stefano Leonardo**	Serving Requests with On-Line Routing
16.30 - 17.00	**Susanne Albers**	On the Influence of Look-Ahead in Competitive On-Line Algorithms
17.05 - 17.45	**Martin Farach**	Dynamic Dictionary Matching

Evening session

20.00 - 22.00 Discussion Session **Theory ↔ Practice**

Wednesday, May 4$^{\text{th}}$, 1994

Morning session

9.00 - 10.00	**Tom Reps**	From "Incremental Attribute Evaluation" to "Bounded Incremental Computation"
10.05 - 10.45	**Tom Marlowe**	Incrementality for Flow-Sensitive Data Flow Problems
10.55 - 11.25	**Hans Rohnert**	Dynamic Algorithms for Compilers

Evening session

20.00 - 22.00 System Demonstrations by Aswin van den Berg and Robert Paige

Thursday, May 5$^{\text{th}}$, 1994

Morning session

9.00 - 9.45	**Roger Hoover**	Alphonse
10.00 - 11.00	**Annie Liu**	Deriving Incremental Programs
11.15 - 12.00	**Bob Paige**	TBA (Algorithm Derivation and Program Development by Program Transformation

Afternoon session

15.00 - 15.35	**Umberto Nanni**	Two Incremental Problems on Graphs
15.40 - 16.15	**Christos Zaroliagis**	Dynamic Shortest Paths
16.45 - 17.30	**Jeffery Westbrook**	Dynamic Connectivity with Backtracking

Evening session

20.00 - 21.00 **Open Problems** Session

Friday, May 6th, 1994

Morning session

9.00 - 10.00	**Tom Reps**	Incremental Interprocedural Data Flow Analysis
10.15 - 11.00	**Mats Wiren**	Incremental Natural-Language Computing
11.15 - 12.00	**Michiel Smid**	An Efficient Construction of a Bounded-Degree Spanner

Abstracts

Susanne Albers
MPI, Saarbrücken

On the Influence of Look-Ahead in Competitive On-Line Algorithms

In the competitive analysis of on-line problems, an on-line algorithm is presented with a request sequence to be served. Each request must be served without knowledge of any future request. We study the issue of lookahead: What improvement can be achieved in terms of competitiveness if an on-line algorithm sees not only the present request but also some future requests? We introduce two different models of lookahead and investigate problems such as paging, the list update problem, caching, the k-server problem and metrical task systems using these models. We show that in the paging problem and in the list update problem, lookahead can reduce the competitive factors of on-line algorithms. In addition to lower bounds we present a number of on-line algorithms with lookahead. We also show that in more general on-line problems (caching, k-server problem, metrical task systems) lookahead is of only very little help.

Aswin van den Berg
Cornell University

Incremental Evaluation of Higher Order Attribute Grammars

Normal order attribute grammars have a separation between syntactic and semantic levels (between derivation trees and attributes). This restricts AG's to only constant phase transformations. In higher order attribute grammars (HAG's) derivation trees can be defined in terms of attributes (NTA's) and attributes can reference derivation trees directly (syntactic references). This enables the iteration of transformations and annotations to be modeled or implemented.
We discuss HAG's and their efficient incremental implementation. To allow constant time access to attributes we explicitly maintain the attributed tree. Higher order evaluation is reduced to normal order evaluation with side-effects that extend the derivation tree. An algorithm is presented that reuses reusable subtrees of an NTA when reevaluated.
Our methods have been implemented in the current version of the Synthesizer

Generator. HAG functionality is used the implementation of the Polya transformation system that was demonstrated during the Seminar.
(Joint work with David Gries & Sofoklis Efremida.)

Martin Farach
Rutgers University

Dynamic Algoriths for String Matching

We consider two problems of string matching whose static version are classics within the field: dictionary matching and text indexing. In the first, one is interested in finding occurences of a set of patterns in a text; the dynamic version allows the set of patterns to change between text queries. In the second problem, a text is given for preprocessing and queries consist of patterns to be matched for; in the dynamic version, the text may change.
We give data-structures for these problems and give open questions.

John Field
IBM T.J. Watson Research Center

A Graph Reduction Approach to Incremental Term Rewriting

We present an algorithm for performing *incremental reduction* in term rewriting systems (TRS's). Incremental reduction is concerned with normalizing sequences of similar terms that are derived from one another by some set of disjoint subtree replacements. This technique is particularly useful as an implementation technique for any rewriting-based system in which relatively small changes to terms to be reduced are made frequently. In addition, however, it can be viewed as the basis for a general theory of incremental computation.
The algorithm applies to any unconditional TRS $\mathcal{R}$ having (almost) any normalizing reduction strategy $\mathcal{F}$, and relies on two key ideas: First, we systematically transform $\mathcal{R}$ into a new TRS $\mathcal{R}'$ and also derive a new reduction strategy $\mathcal{F}'$. These transformations will enable certain key reduction steps to be "checkpointed" for use in subsequent reductions. Next, we implement the transformed system using a variant of *graph reduction*, which allows the results of certain reduction steps to be shared. Each reduction step in $\mathcal{R}'$ either simulate an $\mathcal{R}$-reduction step, or performs an "administrative" reduction. By extending analysis techniques of Huet, Lévy and Maranget, we can define a notion of *relative optimality*, and show that our technique is relatively optimal. In addition, we show that for each simulated $\mathcal{R}$-reduction step, the number of administrative steps is bounded in the worst case by the number of allowable subtree replacements. In

practice, the overhead is usually much lower.

Susan Graham

University of California, Berkeley

Incremental Computing in Software Development Environments

We first examine the role played by incremental algorithms in software development environments (SDE's), notably the benefits of information about changes. Next, we illustrate the high level ideas behind incremental language analysis and summarize the requirements on the specification languages, their use, and their incremental analysis in SDE's. We then review the key ideas underlying the Colander incremental analyzer created by Ballance for the Pan language based editing system, and outline its strenghts and weaknesses. Finally, we summarize some of the goals of the Ensemble system that is being built at Berkeley to provide language-based services for integrated software and multi-media document preparation. The talks by Harrison, Maddox and Muthukkaruppan discuss various aspects of Ensemble.

Michael A. Harrison

University of California, Berkeley

Presentation Issues in Multimedia Systems

The basic properties of the Ensemble system for the creation, manipulation, browsing and analysis of all software documents are presented. The system services include incremental language analysis, high quality formatting, multimedia, multiple presentations, and compound documents. Key features are multiple representations and the ability to extend the system to new media.

The Ensemble presentation was described in some detail. The presentation specifications are separate from the document. An example of the use of the presentation system to reformat programs was shown and related to attribute propagation.

Examples of other media are discussed. There is a close relationship between dynamic media and some well-known models of formatting. The steps involved in adding a new medium to Ensemble are enumerated.

Multiple presentations of the same graphic scene are shown and relative attribution is used to make simple changes.

This abstract represents joint work with Susan Graham, Bill Maddox, Ethan Munson, and Kannan Muthukkaruppan.

Roger Hoover
IBM T.J. Watson Research Center

Alphonse

Alphonse is a program transformation system that uses dynamic dependency analysis and incremental computation techniques to automatically generate efficient dynamic implementations from simple exhaustive imperative program specifications.

Sandra Irani
University of California, Irvine

On-line Algorithms and Restricted Adversaries

This talk is an introduction to competitive analysis and a survey of recent work in the area. We focus on examples where restrictions to the input sequence can better reflect 'typical' input sequence and allow for enhanced algorithm performance. Examples are taken from virtual memory paging and financial games.

Giuseppe F. Italiano
Universitá di Roma "La Sapienza"

Maintaining Spanning Trees of Small Diameter

Given a graph G with m edges and n nodes, a spanning tree T of G, and an edge e that is being delected from or inserted into G, we give efficient $\mathcal{O}(n)$ algorithms to compute a possible swap for e that minimizes the diameter of the new spanning tree. This problem arises in high-speed networks, particularly in optical networks.
(Joint work with Rajiv Ramaswami.)

Stefano Leonardi
Universitá di Roma "La Sapienza"

Serving Requests with On-Line Routing

In this work we present the problem of a server located at a certain starting time at the origin of a metric space that moves at constant unit speed to serve a sequence of requests coming in on-line fashion. A request is presented at the on-line server at a certain time t and requires to serve the visit of a certain point p in

the metric space. Competitive analyses allows to measure how close the solution given by an on-line algorithm is to the optimal solution. The work presents a lower bound of 2 for the competitive ratio on the real line, a 2.5-competitive non-polynomial algorithm for the euclidean plane, a 3-competitive polynomial on-line algorithm for the euclidean plane, and a 7/3-competitive on-line algorithm for the real line.
(This is a joint work with Giorgio Ausiello, Esteban Feuerstein, Leen Stougie and Maurizio Talamo.)

Y. Annie Liu

Cornell University

Deriving Incremental Programs

An incremental program f' is a program that, after a certain kind of input change to a program f, computes f's new result efficiently based on the previously computed result of f. We present a systematic transformational approach for deriving incremental programs from non-incremental programs written in a standard functional programming language. The basic derivation idea is to expand the computation of f on the new input so that subcomputations whose values can be retrieved from the previously computed result of f are replaced by corresponding retrievals. We exploit various types of static analysis and transformation techniques and domain-specific knowledge, centered around an effective utilization of caching, in order to provide a degree of incrementality not otherwise achievable by a generic incremental evaluator. We also propose to extend the approach to address caching intermediate results and discovering auxiliary information. A prototype system CACHET based on the approach has been implemented using the Synthesizer Generator and used to derive a number of incremental programs including incremental attribute evaluation programs.

William Maddox

University of California, Berkeley

Colander II: An Incremental Analysis Facility for Ensemble

We present an overview of the Colander II system, a generator for incremental static analyzers in our language-based programming environment, Ensemble. The talk focuses on the formalism in which we specify the analysis to be performed, justifying it both from the standpoint of expressiveness and of exposing more of the incrementality inherent in the analysis to the incremental evaluator. We augment attribute grammars of the usual sort (providing terms as attribute values) with relation-valued attributes and attributed objects. Relation-valued attributes provide a way to encode aggregate values such that more of their structure is exposed. For example, incremental view-maintenance strategies based on propagating sets of added and deleted tuples are well-known. In contrast to this eager strategy, we can apply truth-maintenance techniques to the relation tuples (viewed as inferred facts in our Prolog-like logical view of relations). Objects permit remote access to their attributes via object references, i.e. pointers to objects. Our objects are used in much the same way as in Hedin's Door AGs, but serve to model program entities in the program database (via their status as unique identities) as well as serving as an implementation device. We discuss other features of our specification language, most notably the use of a separate program database schema which defines the external interface to the results of the analysis, which are presented in an manner that need not (and should not) depend on a particular client known in advance. For example, the user may perform completely ad-hoc queries against the program database. Our implementation is currently in a rather early stage, and presently supports conventional attribute grammars, a limited form of relational attribution, and schema/implementation separation. We have a comprehensive language description for Fortran 77 written in this subset. Furthermore, examples using the full language have been written and partially analyzed, using the parser and static semantics for the full language which have been implemented.

Tom Marlowe
Seton Hall University

Incrementality, Data Flow Analysis, and Flow-Sensitivity

After an overview of data flow analysis, the standard properties of data flow frameworks, and data flow solution techniques, we define "flow-sensitivity" by considering the function classes needed to represent data flow effects on edges at differing levels of granularity. We indicate why incremental data flow analysis is desirable, and review existing incremental techniques and algorithms for data flow analysis. Finally, we consider how framework properties affect incrementality, argue that flow-sensitivity is the primary obstacle, and show how to get around the problem in some cases by encoding the summary edge flow function computation as the solution of a number of data flow problems on the region. We maintain that effective software development environments will require incremental data flow, which will need a combination of summary functions via associated data-valued problems with the use of approximate problems in which the loss of precision remains acceptable.

Emma van de Meulen
CWI/UvA Amsterdam

Incremental Implementation for Algebraic Specifications

We present a technique for deriving incremental implementations for a subclass of algebraic specifications, namely well-presented primitive recursive schemes with parameters. We introduce a concept adapted from the translation of well-presented primitive recursive schemes to strongly non-circular attribute grammars. We store results of function applications and their parameters as attributes in an abstract syntax tree of the first argument of the function in question. An attribute dependency graph is used to guide incremental evaluation. The evaluation technique is based on a leftmost-innermost rewrite strategy. Next, the uniformity of algebraic specifications allows us to generalize our incremental algorithms to functions on values of auxiliary datatypes, without extending or modifying the specification language. Thus we can obtain finegrain incremental implementations. A finegrain incremental implementation of a table datatype can, for instance, solve the problem caused by aggregated attribute values like table datatypes.

Kannan Muhukkaruppan
University of California, Berkeley

SPINE: Synthesizer for Practical Incremental Evaluation

SPINE is a system for efficiently generating practical efficient incremental evaluators for a strongly non-circular class of attribute grammars. Several interactive language-based software development environments use incremental evaluation of attribute grammars for contextsensitive semantic analysis. Ease of evaluator construction, effective consumption of space applicability to a large class of AG's (SNC), ability to handle multiple subtree replacements and close to optimal performance are the key advantages SPINE offers over other existing incremental AG systems. SPINE is being innovatively used in the Ensemble system to provide advanced incremental formatting of structured documents.

Umberto Nanni
University of L'Aquila

Two Incremental Problems on Graphs

The first considered problem is the "Single-Source Shortest Path" (SSSP)-problem. It is possible to handle a sequence of graph updates in a very efficient way in particular situations: considering only edge insertions and specific classes of graphs,

such as bounded genus graphs (including planar graphs), bounded treewidth graphs, and bounded degree graphs. In such cases the SSSP-tree can be maintained in $\mathcal{O}(|\delta| \log n + |\sigma| + |G|)$ total time for all updates, where $|\delta|$ is the number of times that any node changes its distance from the source, $|G|$ is the size of the graph, and n is the number of its nodes.
The second problem concerns the maintenance of a "depth-first-search" (DFS)-tree in a directed acyclic graph, during a sequence of edge insertions (leaving the graph acyclic). The total time required to update the DFS-tree is $\mathcal{O}(m \cdot n)$, where n is the number of nodes and m is the number of edges in the final graph. This means $\mathcal{O}(n)$ amortized time per edge insertions ???????

Robert Paige

New York University/the University of Copenhagen

Algorithm Derivation and Program Development By Program Transformation

Three transformational tools are discussed. Each one exploits a different aspect of incrementality. The first computes fixed points based on a dominated convergence argument; a solution arises as a result of a slowly converging sequence of values. The second turns costly repeated calculations into efficient incremental ones by finite differencing. The third implements efficient data structures for sets and maps by a real-time simulation of an abstract set machine on a RAM. Taking the DFA minimization problem as an example, we show how these transformational tools can be used to explain, verify, analyze, and implement algorithms.

Robert Paige

New York University/the University of Copenhagen

Towards Increased Productivity of Algorithm Implementation - Demonstration of the APTS System

The preceding transformational tools (see the preceding abstract) were incorporated into two language translators within the APTS transformational programming system. The first language is L1, a functional subset of SETL2 augmented with least and greatest fixed points. It has the property that any valid L1 specification can be compiled into a C-program with worst case running time and space linear in the I/O space. The second language is a Turing-complete low level subset of SETL2. Any program in this language can be simulated (using a C-program) on a sequential RAM in real-time.

Both languages were used in productivity studies to test the feasibility of the

transformation to support the implementation of complex nonnumerical algorithms correctly and efficiently. Productivity was measured in terms of the number of source lines in the C-implementation divided by manual programming time. Comparative benchmarks showed that productivity in C using the low-level subset of SETL2 (and its real-time simulation transformation) can be increased over manual C programming by factors ranging from 5.1 to 9.9. Preliminary results also showed that the running time of C-code produced by this new approach can be as fast as 1.5 times that of tightly coded high-quality Fortran. Experiments with L1 were highly encouraging, but not significant enough to draw credible conclusions.
(Joint work with Jiazhen Cai.)

Maarten Pennings

Utrecht University, The Netherlands

Incremental Attribute Grammar Evaluation

Numerous incremental problems with various specific solutions exist. We want to focus on a class of problems and solve the incremental case for the entire class. Compositional problems are problems defined as homomorphisms on an inductive structure (lists, trees). A homomorphism f is defined on the *basis* $<>$ of the inductive structure $f < i >= \ll i \gg$ and on the "*step*" $\oplus$ of the inductive structure $f(a \oplus b) = f a \oplus\!\oplus f b$ where $\ll \cdot \gg$ is some unary and $\cdot \oplus\!\oplus \cdot$ some binary operation. A change to one of the basic values $< i >$ in a term only requires $\log(n)$ re-applications of $\oplus\!\oplus$, where n is the size of the term. This complexity is simply achieved by memoizing (caching) the so-called visit functions f. In other words, compositional problems are well-suited for incremental evaluation. They form the class we investigate, and as a formalism, we use attribute grammars.

Since the visit functions might be non-strict, which makes them hard to memo, we use Kastens' ordering algorithm to break them up into strict subfunctions. However, since intermediate values computed in one of the subfunctions might be needed in a subsequent one, a special mechanism needs to be incorporated: bindings. Finally, since the tree is an input parameter of the visit functions, and hence a key for the cache, fast tree comparison is essential. This is achieved by memoization of constructors so that trees are shared: tree equality thereby reduces to pointer equality.

The sketched machine implements an incremental attribute evaluator. It has been implemented by Matthijs Kuiper and me; it is known as the LRC processor. Some optimalizations can still be realized, most notably splitting the tree and elimination of copyrule nodes.

Thomas Reps
University of Wisconsin

From “Incremental Attribute Evaluation” to “Bounded Incremental Computation”

In this talk, I describe how the parameter $||\delta||$ - roughly, the size of the *change* in input and output - which has been used in the analysis of algorithms for incremental attribute evaluation, can be used in the analysis of algorithms for certain dynamic graph problems. In particular it is useful in variations on the shortest-path problem with positive edge weights. It can be shown that when the complexity of algorithms for this problem is expressed in terms of the size of the (current) input, there are worst-case inputs for which no incremental algorithm can perform better than the “start-over” algorithm (which ignores all previously computed information and applies the batch algorithm). When the complexity of algorithms is measured in terms of $||\delta||$, there is an algorithm whose running time is $\mathcal{O}(||\delta|| \log ||\delta||)$, whereas the running time of the start-over algorithm is not bounded by any function of $||\delta||$. Thus, the use of $||\delta||$ permits one to distinguish between algorithms that are indistinguishable using the conventional parameter $|input|$.
Joint work with G. Ramalingam.

Thomas Reps
University of Wisconsin

Incremental Interprocedural Dataflow Analysis

This talk presents a framework in which a large class of interprocedural dataflow analysis problem can be solved precisely in polynomial time. It then presents an algorithm for the dynamic version of the problem.
The restrictions of the framework are that the set of dataflow functions be a finite set and that the dataflow functions distribute over the confluence operator (either union or intersection). This class of problems includes - but is not limited to - the classical separable problems, e.g., reaching definitions, available expressions, and live variables. In addition, the class of problems that our techniques handle includes many non-separable problems, including truly-live variables, copy constant propagation, and possibly-uninitialized variables.
A novel aspect of our approach is that an interprocedural dataflow-analysis problem is transformed into a special kind of graph-reachability problem (reachability along *interprocedurally realizable paths*).
(Work on the “batch” version of the problem was carried out in collaboration with Mooly Sagiv and Susan Horwitz.)

Hans Rohnert

Siemens AG, Corporate R & D, Munich

Dynamic Algorithms for Compilers

During the design discussion on the object-oriented programming language Sather and its compiler (both under construction at ICSI, Berkeley, CA) we stumbled over several dynamic problems on graphs, especially the topological sorting problem, transitive closure and reachability from a start node. In detail we describe the solution of the dynamic topological sorting problem. We put it into contrast to the original static problem and an on-the-fly application of topological sorting. Briefly we mention work of others on a competitive analysis of topological sorting in an on-line setting vs an adversary.

The last part of the talk considers the applicability of dynamic graph algorithms for compiler writers. The conclusion was that there are possibilities for these algorithms to be applied but traps and pitfalls like additional overhead and data to be stored to slow disks have to be avoided.

Raimund Seidel

University of California, Berkeley

New Proofs for the Clarkson-Shor Framework

We discuss the "configuration space" framework of Clarkson and Shor in the formulation of Mulmuley, which has proved very valuable in the analysis of randomized geometric algorithms.

A configuration space is specified by a set S of n "objects" and a finite set C of "configurations", where each $c \in C$ has associated with it a set of "triggers" $\mathrm{tr}(c) \subset S$ and a set of "steppers" $\mathrm{st}(c) \subset S$ with $\mathrm{tr}(c) \cap \mathrm{st}(c) = \emptyset$. A configuration c is said to be "active" for $R \subset S$ iff $\mathrm{tr}(c) \subset R$ and $\mathrm{st}(c) \cap R = \emptyset$. It is said to be active during an enumeration π of S iff it is active for some subset of S corresponding to a prefix of π.

For integer $i \geq 0$ define

$$X_i(R) \quad = \quad \sum_{\substack{c \in C: \\ c \text{ active for } R}} |\,\mathrm{st}(c)|^{\underline{i}} \quad , \qquad\qquad R \subset S$$

and

$$Y_i(\pi) \quad = \quad \sum_{\substack{c \in C: \\ c \text{ becomes active during } \pi}} (|\,\mathrm{tr}(c)| + |\,\mathrm{st}(c)|)^{\underline{i}} \quad ,$$

and let $A_i(r) = \mathrm{Ex}[X_i]$ and $B_i = \mathrm{Ex}[Y_i]$, where the expectation is taken over all $R \in \binom{S}{r}$ in the first case and over all $n!$ enumerations π in the second.

We prove the following two theorems:

$$A_i(r) \quad \leq \quad \frac{(d+1)^{\overline{i+1}}}{(r+1)^{\overline{i+1}}}(n-r)^{\underline{i}} \sum_{O\leq j\leq r} f_0(j)$$

$$B_i \quad \leq \quad d^{\underline{i+1}}\, n^{\underline{i}} \sum_{O\leq j\leq n} \frac{f_0(j)}{j^{\underline{i+1}}} \qquad\qquad \text{for } O \leq i < \delta$$

with equality in the second case if $\delta = d$.
Here $d = \max_{c\in C} |\operatorname{tr}(c)|$ and $\delta = \min_{c\in C} |\operatorname{tr}(c)|$, and $f_0(j)$ denotes the expected number of configurations active for a random $J \subset S$ of size j.

Michiel Smid
MPI, Saarbrücken

Efficient Construction of a Bounded Degree Spanner with Low Weight

Let S be a set of n points in $\mathbb{R}^d$ and let $t > 1$ be a real number. A t-spanner for S is a graph on S such that for any pair p and q of points there is a path between them of length at most t times the euclidean distance between p and q.
An efficient implementation of a greedy algorithm is given that constructs a t-spanner having bounded degree such that the total length of all edges is bounded by $\mathcal{O}(\log n)$ times the length of a MST for S. The algorithm has running time $\mathcal{O}(n \log^d n)$.
(Joint work with Sunil Arya.)

Jeff Westbrook
Yale University

Dynamic Graph Connectivity with Backtracking

We consider maintaining the equivalence classes defined by connected components, 2-edge connected components, and two-vertex connected components of a graph. Treating the graph as an abstract datatype, we consider efficient implementations of the query "same-2-edge-component (u, v)", the query "same-2-vertex-component (u, v)", and the update operation "insert edge (u, v)", and "undo". The undo operation undoes the most recent not yet undone edge insertion, and so allows a backtracking search through the space of all possible edge insertions. This has applications to both interactive software & logic programming. By representing connectivity structure in an implied form, we achieve

algorithms running in $\mathcal{O}(\log n)$ worst-case time per operation.
(Joint work with Han La Poutré, Utrecht University.)

Mats Wiren

University of the Saarland, Saarbrücken

Incremental Natural-Language Computing

After giving an overview of the role and potential applications of incremental computation in natural-language processing, I focus on one problem, namely, incrementalizing chart/tabular parsing. Specifically, I take the view that the time that an incremental algorithm uses to process a change ideally should be a polynomial function of the size of the change rather than, say, the size of the entire current input. Based on a definition of "the set of things changed" by a modification, I then show to what extent such a guarantee can be given within a chart-based/tabular parsing framework.

Christos D. Zaroliagis

MPI, Saarbrücken

Dynamic Shortest Paths

The *dynamic* shortest path problem is the following: Given a digraph G (with real-valued edge costs but no negative cycles), build a data structure that will enable fast *on-line* shortest path or distance queries (single-pair and/or single-source ones). In case of dynamic changes in G, update the data structure in an appropriately short time. We describe algorithms for solving the above problem in a planar digraph which exploit the particular topology of the input graph. We support the following dynamic changes: edge cost modifications, edge deletions and the "undo" operation of them. The data structure can be updated in polylogarithmic (worst-case) time after any such dynamic change. We also give the first parallel algorithms for solving the dynamic shortest path problem. Moreover, our algorithms can detect a negative cycle, either if it exists in the initial digraph, or if it is created after an edge cost modification. Our result can be extended to hold for digraphs of genus $o(n)$.
(Joint work with H. Djidjev & G. Pantziou.)

Problem Session

On Thursday evening (May 5th) a plenary session was held aimed at identifying (a selection of) interesting open problems in the field of 'incremental computation and dynamic algorithms'. Each participant was asked/required to state an open problem, in no more than two minutes per problem. The session was moderated and timed by Tom Reps. The summary below is a close reconstruction of the open problems presented in the session, in order of presentation.

1. Develop models of computation for obtaining better lower bounds for the circuit-annotation problem, and give efficient algorithms for the problem.
   (Roger Hoover)

2. Develop a standard terminology and a taxonomy of issues in incremental computing.
   (John Field)

3. Give a good, dynamic algorithm for the ordered-merge problem for function maps.
   (John Field)

4. Give an efficient implementation of lazy incremental attraibute updating.
   Maarten Pennings)

5. Give an efficient algorithm for maintaining the value of a (single) distinguished attribute.
   (Kannan Muthukkaruppan)

6. Develop an efficient incremental algorithm for the dynamic k-shortest path problem.
   (Hans Rohnert)

7. Explore the analogy between 'incremental algorithms' and adaptive distributed algorithms and analyse the message complexity of adaptive distributed algorithms in terms of —AFFECTED—.
   (Jan van Leeuwen)

8. Design a good distributed algorithm for the dynamic snapshot problem.
   (Lefteris Kirousis)

9. Design an efficient distributed priority queue algorithm.
   (Umberto Nanni)

10. Improve the complexity bounds for the semi-dynamic shortest-path problem with insertions; also for the fully dynamic shortest-path problem. Maintain shortest-path information under edge-insertion in o(n).
    (Christos Zaroliagis)

11. Analyze the 'competitiveness' of (a) the randomized list update algorithm and (b) self-adjusting trees.
(Suzanne Albers)

12. Develop 'incremental' algorithms to support efficient audio- and video-*grep*.
(Michael Harrison)

13. Incrementalize language-specific preprocessing. Give algorithm for schema updates.
(Susan Graham)

14. Analyze the 'competitiveness' of attribute update problems. Give a model for incremental attributed term-rewriting.
(Aswin van den Berg)

15. Design a good algorithm for updating minimum spanning trees after a sequence of modifications (i.e., multiple heterogeneous modifications to the tree), measured in term of the number of modifications.
(Giuseppe Italiano)

16. Analyze the competitiveness of eager-evaluation strategies.
(Sandra Irani)

17. Many batch algorithms rely on "preconditioned" input. Now consider that "preconditioning : batch problems = reconditioning : incremental problems". Can "reconditioning" be performed using idle cycles?
(Tom Reps)

18. Deal with the dependency of graph algorithms on cycles (dependency cycles) in measuring incremental complexity.
(Tom Marlowe)

19. Apply grammar "technology" from programming languages in computational linguistics. For example, apply techniques for attribute grammars to definite clause grammars.
(Mats Wiren)

20. Is there a general theory for deriving/constructing incremental algorithms.
(Chritién Aarts)

21. Analyze the use of 'space' in incremental algorithms.
(Annie Liu)

22. Design an algorithm for incremental tree pattern matching. Is there a "simple" tree pattern preprocessing algorithm.
(Robert Paige)

23. Study the competitiveness of garbage collection.
(Jeff Westbrook)

24. Develop parallel dynamic algorithms.
(Jeff Westbrook)

25. Is there a general 'computability theory' of incremental computation (or: what does this mean?)
(Jeff Westbrook)

26. Give methods for assessing the quality of algorithms for (dynamic) scheduling in token rings.
(Stefano Leonardi)

27. What can be said about space-time tradeoffs in concrete algorithms, incremental or otherwise.
(Kurt Mehlhorn)

28. Given a lower-directed triangulation of n points in the plane, can another permutation of the points give the same triangulation.
(Raymond Seidel)

29. Give a dynamic algorithm for the nearest marked ancestor problem.
(Martin Farach)

30. Design fully dynamic transitive closure algorithms.
(Han La Poutré)

31. Characterize update sequences that arise in practice.
(Han La Poutré)

32. Design data structures with "unique representation" properties/equivalence properties of members of data types.
(Bill Maddox)

33. Suppose Unions take $\leq M$, what is the complexity of Finds when $M = \mathrm{o}(\log \mathrm{n})$.
(Michiel Smid)

34. Is finding an incremental version of a function on strings decidable? (Annie Liu)

35. Permit "laziness" in the function-caching approach to attribute-grammar evaluation.
(John Field)