Pointer Machines and Pointer Algorithms: an Annotated Bibliography

Amir M. Ben-Amram

September 4, 1995

Abstract

The term *pointer machine* has been used ambiguously in Computer Science literature. In this report we give precise definitions of several models that have been referred to as "pointer machines" and list references for each of these models under the appropriate heading.

1 Introduction

In a 1992 paper by Galil and the author we referred to a "pointer machine" model of computation. A subsequent survey of related literature has produced over thirty references to papers having to do with "pointer machines", naturally containing a large number of cross-references. These papers address a range of subjects that range from the model considered in the above paper to some other ones which are barely comparable. The fact that such different notions have been discussed under the heading of "pointer machines" has produced the regrettable effect that cross references are sometimes found to be misleading. Clearly, it is easy for a reader who does not follow a paper carefully to misinterpret its claims when a term that is so ill-defined is used.

This report is an attempt to rectify the situation. It includes a presentation of the different notions that have appeared under the heading of "pointer machines," meant to help in creating a common terminology which allows the necessary distinctions. Related references are listed following the presentation of each subject considered. We use the following conventions: within each of these lists, alphabetical order is used. The names of the authors are underlined for emphasis in some publications, which survey and discuss a collection of results about a subject, and thus may be of particular interest to the reader.

2 Abstract Machines: Atomistic Models

Abstract machines were introduced to Computer Science, at the first place, as a means of formalizing the notion of "algorithm". The main example is the Turing Machine; another model introduced with this goal in mind is the Kolmogorov-Uspenskii machine, to be described in more detail below. Such machines have "input/output media," conventionally described as tapes on which symbols from some finite alphabet are written: thus algorithms in this class compute functions from Σ^*, where Σ is a finite alphabet, into Σ^*. The fact that this is common to all models has the following important consequences. First, it is possible to prove that all of them compute the same class of functions, thus giving supportive evidence to Church's Thesis. Secondly, it is possible to compare the *power* of different models by studying the relationship between the complexity of problems on the different models, or the complexity of *simulating* one model by another.

The differences between the models are expressed in their *internals,* which are typically specified by a *storage structure* and a set of operations that can be put together to form a *program.* These operations include input/output operations, operations that modify or inspect the internal storage etc.

Much as the input/output language of these machines is the language of strings over a finite alphabet, so is it often desired that the internal operations of the machines have a "discrete" nature, since this seems to support the claim that they represent effective algorithms: this has been argued by Kolmogorov and Uspenskii (1958) as follows.

> The mathematical notion of Algorithm has to preserve two properties...
>
> 1. The computational operations are carried out in discrete steps, where every step only uses a bounded part of the results of all preceding operations.
> 2. The unboundedness of memory is only quantitative: i.e., we allow an unbounded number of elements to be accumulated, but they are drawn from a finite set of types, and the relations that connect them have limited complexity.

Schönhage (1980) referred to abstract machines that have these properties as *atomistic.*

The form of storage suggested by Kolmogorov and Uspenskii as well as by Schönhage is a collection of *nodes* that are interconnected by *pointers* (these are Schönhage's terms, which agree with programming practice). All machine models that have this type of storage structure can be described as *atomistic pointer machines.*

2.1 Kolmogorov-Uspenskii Machines

The Kolmogorov-Uspenskii model represents storage as an undirected finite graph, in which every edge is has one of a finite set of *labels*, and edges incident to the same node must have different labels. This naturally implies a finite bound on the degree of the graph. At any moment, one specific node is designated as *the active node*. The neighborhood of the active node of some fixed radius is called *the active zone*. In the original formulation, each step of the machine consists of applying a fixed transformation that maps every possible form of the active zone into a (generally different) subgraph that has the same "boundary" (so the rest of the storage graph remains accessible). For example, the active zone can be contracted, with the effect that nodes that were previously too far are pulled inside for inspection.

A more conventional programming formulation, which serves to bring this model under the common framework described above, defines a program to be a sequence of instructions that include the following types: unconditional branch, input, output, conditional branch and storage modification. The conditional branch instruction specifies two strings of labels, not longer than the radius of the active zone; the destination of the branch is determined by whether the two paths starting at the active node, and specified by the given labels, lead to the same node. A storage modification instruction can add a node, remove one, add or remove an edge.

The following papers introduce the model:

A. N. Kolmogorov, "On the notion of algorithm," *Uspehi Mat. Nauk* 8 (1953), 175–176.

A. N. Kolmogorov and V. A. Uspenskii, "On the definition of an algorithm," *Uspehi Mat. Nauk* 13 (1958), 3–28. English translation in *AMS transl.* II Vol. 29 (1963), 217–245.

The following publications address the KUM specifically. Publications which address both the KUM and another model are listed later in this section.

D. Ju. Grigor'ev, "Kolmogorv's algorithms are stronger than Turing machines," Seminar Notes of the Steklov Mathematical Institute 60 (1975), 29–37, Nauka, Leningrad. (in Russian).

D. Ju. Grigor'ev, "Imbedding theorems for Turing machines of different dumensions and Kolmogorv's algorithms," Dokl. Akad. Nauk 234:1 (1977), English transl. in Soviet Mathematics Dokladi 18:3 (1977).

2.2 Storage Modification Machines

The SMM model, introduced by Schönhage, differs from the KUM by representing storage as a directed finite graph; apart from that the description in the last paragraph applies. Schönhage did not include the requirement of a fixed radius for the active zone, but mentioned that this change may give a "more precise and realistic" measure of running time.

In contrast with KUMs, here the finite size of the set of labels only restricts the *out-degree* of nodes; an unbounded number of pointers may lead to a single node.

Observe that using a set of two distinct labels, i.e., nodes of out-degree two, any algorithm using a larger out-degree can be simulated with at most a constant factor loss in running time as well as in the number of nodes in storage. This is true for both the SMM and the KUM.

The following papers introduce the model:

A. Schönhage, "Universelle Turing Speicherung," in *Automatentheorie und Formale Sprachen*, Dörr and Hotz (eds.), Bibliogr. Institut, Mannheim, 1970, 369–383.

A. Schönhage, "Storage modification machines," *SIAM J. Comput.* 9:3 (1980), 490–508.

The following publications address the SMM specifically.

J. Y. Halpern, M. C. Loui, A. R. Meyer and D. Weise, "On time versus space III," Mathematical Systems Theory 19 (1986) 13–28.

D. R. Luginbuhl and M. C. Loui, "Hierarchies and space measures for pointer machines," *Information and Control* 104:2 (1993) 253–270.

A. Schönhage, "Tapes versus Pointers, a Study in Implementing Fast Algorithms," *Bull. of the EATCS* 30 (1986), 23–32.

P. van Emde Boas, "Machine models and simulations," in: *Handbook of Theoretical Computer Science*, Vol. A, Elsevier, 1990, 1–66.

The following publications address both the KUM and the SMM.

Y. Gurevich, "Kolmogorov machines and related issues: the column on Logic in Computer Science," *Bull. of the EATCS* 35 (1988), 71–82.

Y. Gurevich and S. Shelah, "Nearly-linear time," *Proceedings, Logic at Botik '89*, Lecture Notes in Computer Science 363, Springer-Verlag (1989), 108–118.

P. van Emde Boas, "Space measures for storage modification machines," *Information Processing Lett.* 30 (1989) 103–110.

See also Section 2.4.

2.3 Knuth's Linking Automaton

The Linking Automaton was defined by Knuth (1968) as a model that may help to understand better the capabilities of algorithms that operate on linked structures. It is defined the same as the SMM except that every node has, in addition to the fixed number of outgoing pointers, also a fixed number of *value fields*. Each value field stores one symbol out of a given alphabet. The program has the capabilities of moving symbols around and of comparing value fields for equality of contents.

D. E. Knuth, *The Art of Computer Programming*, Vol. 1: *Fundamental Algorithms*. Addison-Wesley, Reading, Mass. 1973, 462–463 (first published 1968).

2.4 General Atomistic Pointer Machines

Shvachko (1991) suggested to study machines which operate on graphs, under different specifications of the class of allowable graphs. He mentions the KUM and SMM as particular examples, where the former is obtained by selecting undirected graphs of bounded degree and the latter by selecting directed graphs of bounded out-degree. In addition to these models, he considers "tree pointer machines." These are naturally obtained by restricting the graphs to be bounded-degree trees. We remark that Shvachko applies the original programming style of Kolmogorov and Uspenskii for all the models he considers.

K. V. Shvachko, "Different Modifications of Pointer Machines and their Computational Power," *Proc. Symp. Mathematical Foundations of Computer Science* (MFCS) 1991, LNCS 520, Springer, Berlin, 426–435.

2.5 Jones' `I` Language

An interesting observation on the "abstract machine" way of formalizing algorithms, is that once a programming language is defined, with proper semantics and cost functions (to evaluate complexity), a computational model has been fully specified. In a model thus obtained, the notion of *algorithm* is identified with that of a *program,* and not of *machine.*

Jones (1993) took this approach, and presented two programming languages, `I` and $\mathtt{I}^{su}$, that can be used as general computational models. He used denotational semantics; however, the natural choice of a model for operational semantics is, in the case of $\mathtt{I}^{su}$, an SMM of out-degree two. This makes the $\mathtt{I}^{su}$ model a programming-oriented alternative to the SMM. It only differs by the restriction of the out-degree (which could be easily relaxed), and the fact that the $\mathtt{I}^{su}$ language is structured, while the SMM is programmed using `goto`s.

The `I` language is a restricted version of $\mathtt{I}^{su}$: in this language, the destination of a pointer may only be set when the node it emanates from is created. In programming terms, it has a `cons` instruction but not a `setcar` or `rplaca` instruction (according to your favourite LISP dialect). This results in the memory graph being always acyclic. It is widely believed, that this makes the model weaker in an asymptotic sense, a conjecture that has not yet been proved.

N. D. Jones, "Constant time factors do matter," *Proc. 25th Annual ACM Symp. on Theory of Computing* (1993), 602–611.

3 Abstract Machines: High-Level Models

Recall that machines described in the last section only manipulate a single type of "values," namely symbols from the chosen alphabet; sometimes only pointers are manipulated internally by the program, but in the case of atomistic machines, this does not make a substantial difference. Let us extend the framework by replacing the finite alphabet by an arbitrary set of data (e.g., the integers) which is furthermore structured (i.e., is the domain of some algebraic structure). Let us call this set our *domain* and further equip the model by a set of instructions for effecting various operations that are defined on elements of the domain (e.g., addition and subtraction of integers). All instructions will count as a single step in the calculation of time complexity. Consider the input and output operations too as transporting an arbitrary element of the domain in one step instead of a symbol from a finite set. In accordance with programming-language practice, we refer to the algebraic structure (domain + operations) as a *data type.*

We refer to a computational model thus obtained as a *high-level* machine: clearly, it does not belong to the atomistic class of Section 2, and thus the rich theory of computation developed for these models does not always apply. Moreover, models of different domains compute functions out of different function spaces. It is thus not possible in general to compare such models for efficiency. However, it is quite possible if the models share the data type and only differ on "intrinsic" features such as the style of programming or the structure of memory. In this case, the comparison may tell us something about the role of the features involved.

The main justification for high-level models is that they are closer to the way computer algorithms are described and analyzed in practice. This position was advocated by Ben-Amram and Galil (1992). They also claimed that, to to get closer to this goal, it is worthwhile to use models that reflect the capabilities of useful programming languages. In this vein they introduced two computational models called the $\mathcal{T}$-FLM (Full LISP Machine) and $\mathcal{T}$-PLM (Pure LISP Machine). In both cases, $\mathcal{T}$ stands for the *data type*. The $\mathcal{T}$-FLM is similar to the SMM, with the extensions that pointers point either to storage nodes (which have an out-degree of two, as in LISP structures) or to domain elements, called *atoms*. The $\mathcal{T}$-PLM is a restricted version of the $\mathcal{T}$-FLM. In the PLM, nodes are created by a `cons` instruction, that pairs to given pointers; these pointers cannot be redirected at any later time. This restriction originates from the subset of the LISP language known as *pure* LISP; it has advantages in terms of simple semantics and efficient implementation. However, it is widely conjectured to be weaker than the full model. The name LISP *Machine* (LM) is used in that paper when the distinction between the two models may be neglected.

A. M. Ben-Amram and Z. Galil, "On Pointers versus Addresses," *J. of the ACM* 39:3 (1992).

Some additional results on the relationship of LISP machines to random access machines are reported in

A. M. Ben-Amram, "On the power of random access machines," thesis, Tel-Aviv University, 1994.

4 Parallel Pointer Machiness

As in the case of sequential models, there are different models of parallel computation that have been described as *parallel pointer machines.* On the other hand, it also happened that different names were used for models which are essentially the same. As in the preceding sections, we use, for each model, a unique name, avoiding the ambiguous "parallel pointer machine."

4.1 Atomistic Models

4.1.1 Pulsing Automata

This model can be described as a network of synchronous concurrent automata, that constitutes a directed graph of bounded degree. In every time step, every node ν modifies the destinations of its outgoing pointers, as well as its internal state, according to the states of the automata in its *l-neighborhood*, i.e., nodes reachable from ν by a chain of at most l pointers. The radius of the neighborhood as well as the number of states of every automaton are parameters of the particular network. Note that the network does not change its size: automata cannot be created and do not disappear during computation.

The following publications address this model.

Ya. M. Barzdin', "Universal Pulsing Elements," Dokl. Akad. Nauk 157:2 (1964), 291–294; English transl. in Soviet Physics Dokladi 9:7 (1965) 523–525.

Ya. M. Barzdin', "Capacity of the medium and behavior of automata," Dokladi Akadmii Nauk 160:2 (1965), 302–305; English transl. in Soviet Physics Dokl. 10:1 (1965) 8–11.

4.1.2 Growing Pulsing Automata

This model differs from the previous in that an automaton has the capability of *forking*, i.e., creating a new node, at each step. Thus the network may grow over time.

The following publications all address this model (using three different names: the above, HMM and PPM).

Ya. M. Barzdin', "Universality problems in the theory of growing automata," Dokladi Akadmii Nauk 157:3 (1964), 542–545; English transl. in Soviet Physics Dokl. 9:7 (1965) 535–537.

P. W. Dymond, F. E. Fich, N. Nishimura, P. Ragde and W. L. Ruzzo, "Pointers versus Arithmetic in PRAMs," *Proc. Annual Conference on Structure in Complexity Theory* 1989.

T. W. Lam and W. L. Ruzzo, "The Power of Parallel Pointer Manipulation," *Proc. Annual ACM Symposium on Parallel Algorithms and Architectures* 1989, 92–102.

K. H. Mak, "The Power of Parallel Time," thesis, Univ. of Illinois at Urbana-Champaign report UILU-ENG-95-2206 (1995).

4.1.3 Associative Storage Modification Machines

This model is based on the SMM. It only has one thread of control, that is, one active node, and incorporates parallelism by supporting instructions that affect many nodes in parallel. The idea is that of an *associative content-addressable memory* (hence the name). In such a memory system, it is possible to perform certain operations, in parallel, on all memory cells that contain a certain value. In the ASMM, this is achieved by using *reverse directions.* Recall that in SMM instructions, nodes are specified by a sequence of labels that describes a path from the active node. In the ASMM, certain instructions allow part of the label sequence to be marked "reverse," and this allows us to perform operations such as: "for all nodes C that point to A, set a certain pointer from C to B" or "for all such C, create a new node connected to C." This way an exponential amount of nodes can be created in linear time, and various operations can be implemented in an efficient parallel way. Tromp and van Emde Boas (1993) proved that this model belongs to the so-called *second machine class*, consisting of those models that accept in (deterministic or non-deterministic) polynomial time precisely the languages of the Turing machine class PSPACE.

J. Tromp and P. van Emde Boas, "Associative Storage Modification Machines," in: *Complexity Theory: Current Research*, K. Ambos-Spies, S. Homer and U.Schöning (eds.), Cambridge University Press, 1993.

4.2 High-Level Models

I know of only one parallel model that manipulates linked structures with non-atomic contents. This model is analogous to the PRAM model, except that the shared storage is a directed graph, as in the high-level LISP machines. Processors can access this graph via pointers and perform the standard manipulations. As for the PRAM, variants of the model are obtained by allowing or disallowing common access to a node for changing its contents or just for reading it. Goodrich and Kosaraju give an algorithm for sorting on such a machine. This is a comparison sort, and so their requirements on the data type are rather humble: it has only to admit a comparison operation.

M. T. Goodrich and S.R. Kosaraju, "Sorting on a parallel pointer machine with applications to set expression evaluation," *Proc. 30th FOCS* (1989), 190–195.

5 Pointer Algorithms

5.1 Definition

When a class of problems is too hard to analyse in the most general fashion, one often chooses to concentrate on a particular means of solution that is natural for the problems at hand, and study the capabilities of this means in isolation. A well-known example is the study of comparison algorithms. In this area, we ask for the number of comparison operations that are necessary for solving certain order-related problems (e.g., sorting). We thus *assume* that the data are amenable to pairwise comparisons, and that the algorithm only uses information gained by such comparisons. We *ignore* any other work that a program, implementing this algorithm, might have to perform.

The study of *pointer algorithms* falls into the same framework. Here the problems we are interested at may be generally described as *data-structure problems.* A primary example, and the one first considered in this framework, is the *maintenance of disjoint sets.* In this problem, the data structure represents a collection of disjoint, named subsets of some universe U. Operations include the query: given an element of U, what set contains it (known as *find*) and various update operations, such as merging two sets into one (known as *union*).

A common representation for such structures is a directed graph, including a node for every element and set name, such that it is possible to go from an element to its set name by following directed arcs[1]. This graph could be implemented as a network of records in memory with the arcs represented by pointers from one record to another; however, we are not interested in implementation details. Instead, let us *assume* that our data structure is a graph of the above kind; we can then ask for the complexity of performing data-structure operations as measured by the number of modifications, or accesses, to the graph.

We thus define the class of *pointer algorithms* as algorithms for performing some given data structure operations on a graph representation of the structure. Clearly, the concrete relationship of the graph to the represented structure has to be defined specifically for each problem; in the example of disjoint-set maintenance problems, we require each element as well as set-name to be represented by a node. In general this relationship should be of such nature that the data-structure operations accept pointers to nodes in the graph as input and deliver their output in form of such pointers. For instance, the *find* operation accepts a pointer to a node that represents an element and has to deliver a pointer to the node that represents the appropriate set name. Note that sometimes there is more than

[1]We use the term *arc* for a directed graph while *edge* is used for undirected graphs.

one input pointer (e.g., two pointers are input for a *union*); and sometimes there is more than one output (as in a query that reports all elements satisfying some condition). To measure the *time complexity* of a pointer algorithm we consider the number of arcs followed by the algorithm, i.e., the length of the shortest (possibly branching) paths leading from the set of input nodes up to every member of the set of output nodes. To this we add the number of arcs added to or removed from the graph during the operation. Note that, while a particular algorithm may traverse the same arc more than once, we only count the number of arcs used for the cost of the algorithm. This may underestimate the cost of an algorithm, but such underestimation is inherent in the use of this kind of model, and it turns out that tight lower bounds are nonetheless obtained for various problems. Space complexity is sometimes measured by the number of nodes in the graph.

An even more precise way of defining this model is as an implementation of a given *abstract data structure* (e.g., a union-find structure) by means of the abstract data structure *directed graph*. We impose the restriction that the abstract data types "element" and "set name" must be implemented as the data type "pointer to node". The abstract data structure *graph* provides a well-defined set of operations: add node, add pointer, get pointer and delete pointer. The time complexity of the implementation is defined by the number of these operations, and the space by the number of *add node* operations.

Let us compare the pointer-algorithm model (or more generally, the isolated-means-of-solution approach) with the abstract machine approach to computational complexity of problems. An essential characteristic of the abstract machine approach is the machine-independent representation of the problem, as a function over strings, integers etc. This machine-independence allows us to ask for the complexity of the same problem on different models and in fact to compare different models for their power.

To contrast, in the pointer-algorithm model the problem definition is all but machine independent: in fact, it makes use of *pointers*, an entity which has no external representation. The fact that input and output are given as pointers is essential to the model; for instance, consider a union-find structure representing n singleton sets, so that there are n nodes that represent set names. If a node's name had to be given to the *find* procedure in an external representation, say a string or a number, and the procedure only maintains a finite number of pointer variables, then $\Omega(\log n)$ arcs would have to be followed in most cases just to locate the desired set-name node.

Thus such algorithms are a natural model only for data-structure problems, where we may assume that the procedures we consider are intended for use by a "client" program: this program may obtain a pointer from one procedure call and at a later time pass it to the next. In fact, this "client" may even be the one that supplies the pointers in the first place: think of a program that manages a set of records which, among other uses, have to be organized in a union-find structure. The requirements of this user naturally call for a pointer algorithm.

We finally remark that, except as noted below, it is assumed that the out-degree of the graph is bounded by a constant independent of the problem instance (for example, it cannot be a function of the number of elements). We call this the *fixed degree requirement.*

5.2 Separable Pointer Algorithms

In the context of the disjoint set maintenance problem a special type of pointer algorithms has been considered, known as *separable pointer algorithms.* This class of algorithms is defined by imposing the *separability condition*: after every operation is completed, the graph can be partitioned into disjoint subgraphs that correspond to the current disjoint sets. The subgraph corresponding to a given set contains the node for its name. No edge leads from one subgraph to another.

It turns out that for such algorithms tight lower bounds may be obtained even if the fixed degree requirement is omitted. Therefore, the class of separable pointer algorithms is defined without this requirement.

We remark that it is in this framework that pointer algorithms have been first studied (Tarjan, 1979).

5.3 References

5.3.1 References Involving Disjoint Set Union Problems

Many of the publications on pointer algorithms have to do with the set-union problem in various variations. We group all of these in the following list. In this list, we mark publications with the acronyms PA or SPA to indicate whether they apply to (degree bounded) pointer algorithms or to separable pointer algorithms.

A. Apostolico, G. Gambosi, G. F. Italiano and M. Talamo, "The set union problem with unlimited backtracking," *SIAM J. Comput.* 23:1 (1994), 50–70. (SPA)

N. Blum, "On the single-operation worst-case time complexity of the disjoint set union problem," *SIAM J. Comput.* 15:4 (1986), 1021–1024. (SPA)

N. Blum and H. Rochow, "A lower bound on the single-operation worst-case time complexity of the union-find problem on intervals," manuscript. (SPA)

Z. Galil and G. F. Italiano, "Data structures and algorithms for disjoint set union problems," *ACM Computing Surveys* 23:3 (1991), 319–344. (PA & SPA)

J. A. La Poutré, "Lower bounds for the union-find and the split-find problem on pointer machines," *Proc. 22nd Annual ACM Symp. on Theory of Computing* (1990), 34–44. Full version: technical report RUU-CS-89-21, Department of Computer Science, Utrecht University, 1989. (PA)

K. Mehlhorn, S. Näher and H. Alt, "A lower bound for the complexity of the union-split-find problem," *Proc. 14th ICALP* (1987), 479–488. (PA & SPA)

H. Mannila and E. Ukkonen, "Time parameter and arbitrary deunions in the set union problem," *Proc. 1st Scandinavian Workshop on Algorithm Theory* (SWAT) 1988, LNCS 318, Springer, Berlin, 34–42. (PA & SPA; remark — Section 2 of this paper applies to PA, although the other term is used).

R. E. Tarjan, "A class of algorithms which require nonlinear time to maintain disjoint sets," *J. Comput. System Sci.* 18 (1979), 110–127. (SPA)

J. Westbrook and R. E. Tarjan, "Amortized analysis of algorithms for set union with backtracking," *SIAM J. Comput.* 18 (1989), 1–11. (SPA)

5.3.2 Other References

The following papers consider (non-separable) pointer algorithms in conjunction with other data-structure problems.

B. Chazelle, "Lower bounds for orthogonal range searching," *J. of the ACM* 37 (1990), 299–212.

B. Chazelle and B. Rosenberg, "Lower Bounds on the Complexity of Simplex Range Reporting on a Pointer Machine," *Proc. 19th ICALP* (1992), LNCS 623, 439–449.

D. Harel and R. E. Tarjan, "Fast algorithms for finding nearest common ancestors," *SIAM J. Comput.* 13 (1984), 338–355.

Acknowledgement

The following people have contributed references for this bibliography: Stavros Kolliopoulos, Michael C. Loui.