RESPONSE: THE MODELS ARE BROKEN, THE MODELS ARE BROKEN![†]

Allen Newell[*]

I. INTRODUCTION

I was asked to give a computer scientist's perspective on the issue of patentability and algorithms, mainly in response to Professor Chisum's Article.[1] I am almost entirely innocent of matters of patent law. Thirty years ago, at the Rand Corporation, some colleagues and I unsuccessfully tried to patent list processing. I was baffled at the time, and have paid no attention to the patent field ever since. Yet the issue of patenting algorithms is of considerable importance and reading Professor Chisum's Article has stirred in me some thoughts, which may be worth sharing.

Professor Chisum's Article launches a full scale attack on *Gottschalk v. Benson*,[2] which held—erroneously in Chisum's view—that algorithms are not patentable. Chisum may well be right that the *Benson* case has brought on much analytic confusion in the software patent area. He may also be right in supposing that, if *Benson* had only been decided differently, the specific confusion that occurred in its aftermath would not have materialized. It seems to be his view that a different holding in *Benson* would have brought about analytic sweetness and light.

My point is precisely to the contrary. Regardless how the *Benson* case was decided—whether that algorithm or any other was held patentable or not patentable—confusion would have ensued. The confusions that bedevil algorithms and patentability arise from the basic conceptual models that we use to think about algorithms and their use. That is why I have entitled my remarks, "The Models are Broken, the Models are Broken."

I realize that this title makes me sound a little like Paul Revere: "The British are coming, the British are coming." I am sounding a

† © Copyright 1986, Allen Newell.

* U.A. and Helen Whitaker University Professor of Computer Science, Carnegie Mellon University.

1. Chisum, *The Patentability of Algorithms*, 47 U. PITT. L. REV. 959 (1986).
2. 409 U.S. 63 (1972).

call to arms—to do what it takes to banish the confusion and to make a new set of concepts to replace the old.

But then again, it may sound like Chicken Little: "The sky is falling, the sky is falling!" I am unsure of which of these two interpretations I think you ought to put on my remarks—or indeed which one I should either. Be that as it may, this theme is exactly what I want to lay before you. The models we have for understanding the entire arena of the patentability of algorithms are inadequate—not just somewhat inadequate, but fundamentally so. They are broken.

Let me say at the start that I have no opinion about whether algorithms should be patentable—whether they are a patentable subject matter to which notions of invention, novelty, fair reward, and the like, apply. Whatever latent inclinations I might have originally possessed were washed away under the impact of reading Professor Chisum's trenchant analysis. However, as I assimilated his arguments, I gradually perceived difficulty after difficulty with the underlying conceptual groundwork on which he was necessarily forced to build. By now, these models seem to me to be sufficiently broken that arguments sprout on all sides of the question. Thus, all I can attempt is to iterate through the conceptual difficulties.

II. The Nature of Algorithms

Interestingly, the one thing that does not present conceptual difficulties is the notion of an algorithm itself. A standard definition is: An algorithm is an unambiguous specification of a conditional sequence of steps or operations for solving a class of problems. This definition is perfectly reasonable, it is not arcane, and I believe we can all live with it. The confusion, then, is not in the nature of algorithms. It is all the things around it that get confused.

The first confusion is using involvement with numbers as the hallmark for distinguishing mathematics from nonmathematics, as an aid to determining what is an algorithm. This confusion is easily cleared up. Algorithms are certainly mathematical objects. That is an acceptable model. However, mathematics deals with both nonnumerical things and numerical things. Correspondingly, there are both numerical and nonnumerical algorithms. Therefore, any attempt to find a helpful or cutting distinction between mathematics and nonmathematics, as between numerical or nonnumerical, is doomed. Indeed, in the mid-1930s the central argument of a famous proof, by the mathematician Kurt Godel, involved showing that all of logic is a part of number theory. The scheme, which ever after has been called Godel numbering, assigns an integer to each logical expression, such that all the truths in logic become theorems about the integers. Although lawyers need never become acquainted with Godel numbering, they should realize there is an underlying identity between the numerical and the nonnumerical realms that will confound any attempt to create a useful distinction between them.

Next consider algorithms and mental steps. The main line of progress in psychology for the last thirty years (called cognitive psychology) has been to describe human behavior as computational. We model what is going on inside the thinking human brain, as the carrying out of computational steps. Therefore, humans think by means of algorithms. Sequences of mental steps and algorithms are the same thing. Any attempt in the law to make distinctions that depend upon contrasting mental steps versus algorithms is doomed to eventual confusion. It is not important whether you accept this computational view of human thinking. There can be controversy about whether such an approach is the correct one for psychology. What is important is that such a view is a major one in the study of the human mind—that many psychologists see the mind this way and that thousands of technical papers are written from within this view, covering large expanses of psychological phenomena. Any attempt to erect a patent system for algorithms that tries to distinguish algorithms as one sort of thing and mental steps as another, will ultimately end up in a quagmire.

Just avoiding the use of this distinction is only half the story. An identity between algorithms and mental steps leads to such questions as whether you can keep people from thinking patented thoughts. You might attempt to avoid such an untenable position by invoking a doctrine of fair use. Indeed, I found the comments by Professor Chisum on the problem of fair use[3] interesting. But the implications of this identity go much further. We are talking about people who engage in those patented thoughts daily and hourly—even every few seconds—in the pursuit of their business and who make their money and their livelihood by so doing. I expect that any doctrine of fair use would experience substantial strain under such challenges.

3. Chisum, *supra* note 1, at 1018-19.

III. Algorithms and Basic Truths

Let us talk about natural law and mathematical truths. One model underlying the patent system posits the existence of a gap between general scientific discovery and its application to matters of social and economic value. The discovery of a natural law or mathematical truth does not wear its practical application on its sleeve, so to speak. Additional discoveries and inventions must occur. However, no great intrinsic motivation exists for making such practical inventions. (Natural human curiosity and the prospect of fame presumably suffice to keep some of us seeking the general truths of nature.) Thus, economic rewards must be proffered to encourage jumping this gap with the additional inventions. The patent system is designed to provide such rewards.

In light of this model, the natural question is whether algorithms are to be considered either natural laws or mathematical truths, hence not to be encouraged by patents; or whether they are inventions that jump the gap from such laws and truths to application, hence to be encouraged by patents. One view comes from observing how practical algorithms can be. They are directly related to use in specific tasks, tasks that can be of the utmost value. That would seem to place them as devices to jump the gap to application, hence as patentable.

But a more confounding answer flows from the general nature of computer science. *All* of computer science is directly related to use. There is essentially no gap, no matter how pure or basic the science is. With rare exceptions, scientific knowledge in computer science is in the form of means-ends relationships—what to do to obtain something of value. Indeed, this is just the essence of algorithms: what to do to perform a task. But algorithms, far from being an applied part of computer science, are at the center of its basic theoretical structure. For example, we spend our theoretical energies analyzing how to sort a collection of objects, that is, how to arrange them in order. Nothing could be more practical than that. But nothing (for a time) offered more interesting basic theoretical questions. We understand sorting pretty well by now, but the theoretical analysis of other algorithms has taken its place. In sum, it is not possible to do anything in computer science without having it be almost immediately related to use, with only small efforts of the imagination. So where is the gap? Hence, where is the rewardable, risky, inventive effort?

Let us pursue the consequences of patenting algorithms. Would it have been possible to patent the integers? "No," comes the reply—

integers are mathematical truths. Even I know that. Actually, the integers are abstract, mathematical objects, and mathematical truths are things like, "Between every two consecutive odd integers there is exactly one even integer." But let that pass. So the integers cannot be patented. But certainly one might contemplate patenting addition. "Well, that is still not true," comes the reply—addition is defined to be a mapping of pairs of integers (the addends), to integers (the sum). That is not even a truth, that is just a definition. Ah, but if you want actually to *do* addition—that requires doing a sequence of things, not to the integers, which are abstract (so you cannot do things to them anyhow), but to some representation of the integers. Doing addition is accomplished by carrying out an algorithm. If algorithms are patentable, then I can keep you from doing addition with the algorithms invented for it. There would be ever so many things that the poor would not be able to do, such as add up their grocery bill.

However, a further response is possible. Addition is not an algorithm, for there are many methods to do addition. Consequently, you cannot patent addition. Certainly, an essential part of our notion of method is that many methods exist to attain any particular end, each of which has to be discovered or invented. A patent rewards diligence in finding one method, but it leaves open other methods for other inventors to find. Perhaps, then, I can patent some particular method of doing addition. Maybe that seems all right. But we are not yet out of the woods. Addition algorithms work on a given representation. For any such representation, it is plausible that there are only a finite number of relevantly different algorithms that are of any reasonable efficiency. Although an indefinite number of algorithms for addition exist, if the representation is specified and only efficient algorithms are considered, it seems to me quite possible to exhaust the reasonable algorithms. Such a result would have the character of a mathematical truth. Indeed, it is even conceivable that this fact (there is only one more addition method for such and such representation with such and such qualifications) could be proved before we actually discovered the exact last algorithm. So we could find ourselves in the following situation. We have allowed patents on addition algorithms. We discover that there is only one more reasonable algorithm left to be discovered. Continuity with past patent actions indicates we should permit its patenting when found. But to do so permits, not just a monopoly, but a stranglehold on a basic behavior, to wit, being able to add. An odd situation, to say the least.

There is one last way to avoid the swamp that seems to await the patenting of addition methods. We can focus on the primeval character of addition. Yes, it might have been a problem, but not now. Addition cannot be patented, because it is already in the public domain. Moreover, it is special—we will not see its like again in terms of generality and pervasiveness. But this answer will not do. Computer science is nothing but a breeding ground for new algorithms, and computer science is hardly out of its swaddling clothes. Algorithms of immense generality and scope will continue to emerge for as long as the science endures. Examples are easy to come by. The simplex algorithm for doing linear programming was invented by G. Dantzig in 1948. It was, until recently, the only practical algorithm for solving a huge class of management and production problems. The fast fourier transform was invented by J.W. Cooley and J. Tukey in 1965. It essentially created the entire field of digital signal processing. Although considering the patenting of algorithms for addition may seem a bit melodramatic, it is perhaps not entirely unrepresentative.

IV. The Embodiments of Algorithms

Let us turn to the embodiments of algorithms. Computer science understands well the essential feature of a digital computer. First, there is a machine (call it the *operations-machine*) that can perform any of a collection of operations. For standard computers these are mostly operations that modify, store or retrieve data; but many other sorts of operations are possible. This is a real machine, by anyone's standards. However, when turned on, it does not actually do any of its operations, but awaits some signals from the outside to evoke them. Second, there is a specification (the *program*) of what behavior is desired, that is, what operations are to be performed and in what sequence. This specification is essentially a textual document, although it is not encoded as marks on paper, but as marks in some other medium. Lastly, there is an *interpreter*, which is also a real machine by anyone's standards. Except that electrons are very quiet it would clank. If you feed the specification into the interpreter, the interpreter will send signals to the operations-machine to make it carry out the sequence of steps specified. By now there is nothing obscure about this arrangement and I daresay you are all familiar with it.

There may be nothing obscure about the arrangement, but several remarkable consequences flow from it. The one of interest to us can be called the *transfer of creativity*. The operations-machine has a certain scope for action. It is a physical arena in which one can imagine inventing how the machine might do new things and then reducing this to practice. What the interpreter does is to transfer all these possibilities for action to the specification. Going through the interpreter does not lose any of the capabilities of the operations-machine for behavior. Consequently, nothing is added to the nature of creation and invention beyond that which happens in the programming language. (The interpreter makes a number of differences, and there is ample scope for ingenuity in its construction; it just does not restrict the scope of the operations-machine.)

What is true of programs is true of algorithms. An algorithm is just an abstract program, which is to say, just an abstract specification. The abstraction involved in an algorithm concerns relatively routine matters—what exact representation of the data to use, how to modify these representations in detail to accomplish the primitive steps specified by the algorithm, etc. These are important and their choice may make important efficiency differences. But the essential issues of what is to be done are reserved to the algorithm. Indeed, the whole point of writing an algorithm is to convey the essentials of what the operations-machine is to do. Thus, any attempt, for the purposes of locating creativity and invention, to distinguish between the algorithm and any particular embodiment of it turns out to be extremely difficult.

This also explains why algorithms cannot be distinguished from programs. Computer science makes the distinction all right, but to express *degrees* of abstraction. An algorithm is more abstract than a program. Given an algorithm, it is possible to code it up in any programming language. You might think that a program should be something like an algorithm plus implementation details. Thus, you examine the text of a purported algorithm—if you find an implementation detail, you know it is a mere program. But it is not so, at least not in any way useful to the law. For a program is also abstract. It is a general specification for action that is interpreted for a particular occasion by the interpreter. The program is missing certain details, which must be added when the program is to be run (by other programs variously called compilers, assemblers and loaders). The root difficulty is that one man's detail is another man's essential. It all depends on the purposes.

This analysis can be turned around. Since an algorithm determines the sequence of steps to be performed, then there must exist some, perhaps powerful, interpreter than can go directly from the specification of the algorithm to carrying out the steps. The existence of such an interpreter is the acid test of whether an algorithm has really been given. But then the algorithm in this high-level representation looks just like a programming language of a high-level and powerful interpreter. There are certainly matters of degree—algorithms are conventionally more abstract than programs. But there are no separations of kind, even though in the current art we often do not know enough to construct the powerful interpreters. This state of affairs, by the way, is reflected currently in the automatic programming field, where they refuse to draw any hard and fast line between systems to design algorithms and systems to design programs. From their point of view, it is just a continuum of design tasks. The computer science field maintains a distinction (so you can buy textbooks on algorithms and textbooks on programming), but it is more the distinction between what specifications to write to do tasks of interest versus the details of particular schemes for writing the specifications.

These considerations lead to understanding the difficulty with another distinction, namely, algorithms versus methods and processes. Taking the latter to involve transformations of matter, we understand them clearly to be a patentable domain. We also believe firmly that any physical technological domain admits of methods and processes. Now for the rub. It is entirely conceivable that some areas of technology will end up with *all* of the inventive activity in terms of algorithms. That is, with the transfer of creativity, as discussed above, even though a domain involves transformations of matter, yet all inventions of useful transformations occur by the invention of algorithms.

This has not happened yet, but the handwriting is on the wall. Consider, as possible examples, technologies that compose durable physical objects arbitrarily from a continuous medium, such as cast iron, plastics, or powder metal technology. Suppose, now, we get the construction of forms for the medium under suitable computer control. That is, from a data structure in the computer, an automatic, essentially robotic, device constructs the form or mold from which routine art produces the physical objects. This is just a special version of our operations-machine. This is only done currently in crude ways and in part, but it is an active area of research. Actually, it was accomplished a long time ago in the area of numerically controlled machine tools, but as far as I know the dilemmas for patentability have not become serious there.

However, do not bet the foundations of patent law on its not happening for important domains. Such advances imply that the entire realm of invention for a technological domain would come to reside in the computer. To reside in the computer implies that algorithms will be developed that are the direct analogues of methods and processes. For instance, there could be algorithms for taking a data structure that defines a rough casting, and polishing it—which is to say, producing a new data structure that defines a smooth and pleasing shape. The entire evolution of such technological domains may come to reside in the continual discovery and refinement of algorithms. An almost instinctive objection to this arises from a belief that *simulation* cannot be that good—that one cannot capture a big hunk of rich, raw, resplendent reality in a computer, so that all that can be invented in the real world can now be invented in the symbolic world. But many indications to the contrary exist. The advance of computer animation in the movies is just one. The development of CAD systems for dealing interactively with the 3-dimensional geometric world is another. One should be prudent in betting the foundations of the law on instinctive reactions.

With the exception of the last one, my examples all seemed to support the notion that patenting algorithms was fraught with difficulties—that the conservative position was to err on the side of not patenting algorithms. This last example counts on the other side. If methods and processes over large technological domains become an exercise in algorithms, then it is extraordinarily dangerous *not* to patent algorithms. To do so would abandon the whole economic scheme that stands behind patentability as an appropriate energizer of human effort towards desirable social ends.

V. The Form of Algorithms

Let us talk about how to recognize an algorithm when you see one. The definition of algorithm at the beginning says it determines a sequence of steps. Thus, we might expect an algorithm to be in the form of a procedure—do this, and then do this, and then do this, and so on. Anyone can tell a procedure, for it lays out exactly the steps to be taken. Sometimes there is conditionality on the data, so that a procedure has to include something of the form: if the situation is thus and so, then do this, otherwise do that. So that is a model for how to tell an algorithm when you see one. Look at the text and examine its form to see if it is a procedure. Historically, that describes pretty closely what programming languages (Fortran, Ada, Cobol, assembly languages, etc.) and informal notations for algorithms (in textbooks, etc.) are like.

But alas, for our models, the reality of computer science moves on. This reality leads to conceptually richer ground that is highly productive for both theory and application. But it destroys the clean model whereby an algorithm could be recognized by its having a procedural form. Computer science takes an algorithm to be any specification that *determines* the behavior of a system. These specifications can be of any kind whatsoever as long as they actually provide the determination through the interpreter.. Consequently, the form of the specification need no longer be procedural. Sequences of steps must march out after interpretation, but sequences of steps need not march into the interpreter.

This is hardly an idle possibility. We now have languages for writing algorithms that look very different from a sequence of steps. For instance, in some programming systems one simply provides a set of constraints that are to be satisfied by the ultimate actions, and the interpreter (or compiler) determines what actions are needed to satisfy them, and then executes them. A set of constraints does not *look* like a step-by-step procedure, but it is just as good as one, because it determines the steps. Other cases are no longer even esoteric and have moved well beyond the realm of research. The form of many expert systems is simply a collection of if-then rules that provide the knowledge that is needed to perform a task. There is no easy way of seeing such an expert system as a sequence of steps—except that the rule interpreter determines at each moment one rule to fire. Some of you may be familiar with the rhetoric of PROLOG, a prominent expert-system language, "You don't have to say how it's done, you can simply give to the system the knowledge about the task and it will go and do it." The rhetoric is a little overblown, but it illustrates how difficult it will be to detect whether some text is an algorithm from its form alone. If all that counts is the knowledge, then the specifications can look declarative or procedural or any other way.

That these difficulties are upon us can be seen in the Article by Duncan Davidson.[4] He wanted to put the data structure outside his

4. Davidson, *Common Law, Uncommon Software*, 47 U. PITT. L. REV. 1037 (1986).

"black box." As anyone in computer science knows, the boundary between data and program—that is, what is data and what is procedure—is very fluid. In fact, as our discussion of the forms of algorithms indicated, there is no principled distinction in terms of form or representation of which is which. What counts is the total body of knowledge represented somehow in the assembled symbolic expressions. This totality determines the ultimate behavior of the machine. The better the interpreters, the further away the specification can be from an explicit "do this, then do this" form. Because of this, what is outside his black box and what is inside it is open to great manipulation, either to avoid or to create copyright problems, as the case may be.

VI. Increasing the Invention of Algorithms

My final example concerns whether patenting algorithms will lead to more or less innovation in the software field. The standard economic model lying behind the patent system, which was in evidence in several places in Professor Chisum's Article, is that inventors produce inventions that, in turn, produce more of the consumables that society desires. Giving control of the invention to the inventors increases the price of the ultimate products to society, over what would be the case if the inventions were available for all to use. But this cost is more than offset by the increased number of inventions that become available to society, due to the incentive to inventors. These inventions raise the net productivity of the economy.

Consider an alternative model, in which inventions produce, not consumables, but "inventables." That is, suppose the primary effect of every product is to enable additional inventions. There will be consumption as well, but mostly new invention based on ownership of the products. To be clear about the key assumption: New invention comes from the products produced by the original invention, not from the original invention itself. The price increase due to the patent monopoly will restrict the amount of product sold, just as in the original model. But now this implies that the number of new inventions that occur will also be restricted. As long as there is some baseline flow of inventions in society, then it is possible that the loss of inventions to society from the monopoly price restrictions will more than offset the gains in inventions from the financial encouragement. The patent system could discourage invention rather than encourage it.

This may seem like an extreme alternative model. However, some computer communities may approximate it. There, people take each other's programs freely, then enhance them, and then pass them on to others, who do more of the same. Of course, they also use them as well. But consuming the behavior of a program does not consume the program. Furthermore, it is the possession of the previously invented program that permits the new invention to occur. For the new inventor adds, modifies, enhances and reshapes the existing system, mostly in small ways, though occasionally substantially. The capabilities of the system evolve and grow. The motivations for such enhancements are partly that one benefits from the inventions themselves, for one gets to use the enhanced system. But the motivations are also partly those that keep the artist and the mathematician creating—it becomes a medium of expression and a coin of the realm. If patentability implies that mostly what is used is left untouched and unenhanced, then the total improvements in the community's software may well decrease, even though some people are induced to work harder at innovating to capture the rewards from patents. They must do their inventing from a poorer base.

I do not seriously defend this alternative model. My goal is more modest—to point out that the world of algorithms and computers may have a different character than the standard economic model of incentives that underlies the patent law. Even this model may be broken.

VII. Conclusion

So what have I told you? I have told you the models are broken. What I hope you have heard is that we are in a hell of a shape. What is to be done about it? If you think I'm Chicken Little, then the right thing to do is to go your own way. Do not follow me and you will be all right—for that is the lesson of Chicken Little.

If you are more inclined to take me seriously, then you must either throw the models away or you must fix them. The law by itself cannot fix these broken models. The models belong to computer science. However, these models are not broken for computer science's own purposes. They are serving it just fine. Computer science is developing into a pervasive technology, backed up by a deepening scientific understanding, that encompasses all information processing from the most restricted to the most intelligent, and whether by machines or by humans. Computer science is full of promise and positive challenge.

That the models are good for computer science does not automatically make them good for dealing with computers and the law. In particular, computer science can thrive on continued radical change, even when we hardly understand it. The law has other requirements, such as stability of concepts over time and being able to make clear distinctions for the sake of property rights.

I think fixing the models is an important intellectual task. It will be difficult. The concepts that are being jumbled together—methods, processes, mental steps, abstraction, algorithms, procedures, determinism—ramify throughout the social and economic fabric. I am not worried about how new and refurbished models, if we could get them, will get back into the law. They will migrate back by becoming part of legal arguments, or legislation or whatnot. There are many different paths. The task is to get the new models. There is a fertile field to be plowed here, to understand what models might work for the law. It is a job for lawyers and, importantly, theoretical computer scientists. It could also use some philosophers of computation, if we could ever grow some. It is not a job for a committee or a commission. It will require sustained intellectual labor.

And, remember, nothing I have said here tells whether to allow algorithms to be patented or not. To tell that requires fixing the models.