March 18, 2009

On Abstraction and Equivalence in Software Patent Doctrine: A Reply to Bessen, Meurer and Klemens

Andrew Chin

ON ABSTRACTION AND EQUIVALENCE IN SOFTWARE PATENT DOCTRINE: A RESPONSE TO BESSEN, MEURER AND KLEMENS

Andrew Chin[*]

ABSTRACT

Recent books by Professors James Bessen and Michael Meurer and by economist Ben Klemens have argued that software warrants technology-specific treatment in patent doctrine. This article argues that the authors' categorical claims about software are unsupported by computer science, and therefore cannot support their sweeping proposals regarding software patents as a matter of law. Such proposals therefore remain subject to empirical examination and critique as policy choices, and are unlikely to be achieved through judicially developed doctrines.

INTRODUCTION

Two recent monographs currently stand at the center of the decades-old controversy over whether software-related inventions should be considered patentable subject matter under § 101 of the Patent Act, a controversy still unresolved by the Federal Circuit's recent en banc decision in *In re Bilski*.[1] In 2006, Brookings Institution economist Ben Klemens published *Math You*

[*] Associate Professor, University of North Carolina School of Law.

In the course of writing to correct misinterpretations of the Karmarkar algorithm and other results in computer science, the author wishes to note for the record his own erroneous statement (at the age of 19) that Karmarkar had "apparently solved the longstanding 'traveling salesman' problem." *See* Andrew Chin, *Math, At Its Best, Lives On*, THE DAILY TEXAN, Aug. 16, 1985, at 8 (reporting on Michael Saks's plenary lecture on the algorithm at the 1985 Joint Mathematics Meetings in Laramie, Wyo.). Any errors in the present Article are solely the author's responsibility, and he intends to acknowledge them likewise in due course.

[1] In re Bilski, 545 F.3d 943 (Fed. Cir. 2008) (en banc). For a brief discussion of *Bilski*'s failure to resolve the controversy over software patents, *see infra* text accompanying notes 159-170.

Can't Use: Patents, Copyright, and Software,[2] in which he argued that software (and general-purpose computers programmed with software) should not be patentable.[3] Klemens has subsequently clarified and elaborated this argument in a law review article[4] and founded the End Software Patents Project, an organization seeking "to eliminate patents for software and other designs with no physically innovative step."[5]

In the 2008 book *Patent Failure: How Judges, Bureaucrats, and Lawyers Put Innovators at Risk*,[6] Boston University economics professor James Bessen and law professor Michael J. Meurer document the failure of patents to provide effective notice of their scope.[7] Bessen and Meurer single out software and business method patents for special criticism,[8] and conclude that "patent reform will not likely be successful unless these areas are specifically addressed."[9] They argue for "modest" technology-specific changes in patent doctrine;[10] however, if these initial changes "fail to work sufficiently well," they would consider "more aggressive" reforms[11] such as restricting or eliminating the eligibility of software-related inventions.[12]

Both monographs provide detailed accounts of the *symptoms* of software-related patent system dysfunction, including overwhelmed examiners,[13] high litigation costs,[14] and structural distortions of software-related industries.[15] These observations, particularly in the context of Bessen and Meurer's extensive review of empirical law and economics scholarship on the patent system, lend considerable support to the authors' policy arguments. The authors of both books stand on shakier ground, however, in their *diagnoses* of the patent system's difficulties in dealing

[2] BEN KLEMENS, MATH YOU CAN'T USE: PATENTS, COPYRIGHTS, AND SOFTWARE (2006).

[3] *See id.* at 63-64 & 158-60.

[4] Ben Klemens, *The Rise of the Information Processing Patent*, 14 B.U. J. SCI. & TECH. L. 1 (2008).

[5] End Software Patents, *ESP Releases Report on the State of Softpatents*, News Release (Feb. 28, 2008) <http://endsoftpatents.org/28-february-2008:esp-releases-report-on-the-state-of-softpatents> (visited June 15, 2008).

[6] JAMES BESSEN & MICHAEL J. MEURER, PATENT FAILURE: HOW JUDGES, BUREAUCRATS AND LAWYERS PUT INNOVATORS AT RISK (2008).

[7] *See id.* at 46-72.

[8] *See id.* at 187-214.

[9] *See id.* at 247.

[10] *See id.* at 244, 246.

[11] *See id.* at 244.

[12] *See id.* at 245.

[13] *See* BESSEN & MEURER, *supra* note 6, at 192-93; KLEMENS, *supra* note 2, at 84-90.

[14] *See* BESSEN & MEURER, *supra* note 6, at 191-93; KLEMENS, *supra* note 2, at 90-91 & 107; Klemens, supra note 4, at 27-32.

[15] *See* BESSEN & MEURER, *supra* note 6, at 190-91; KLEMENS, *supra* note 2, at 92-107; Klemens, *supra* note 4, at 21-27.

with software-related inventions.

In a section of their book entitled "Why Software Patents Are Different,"[16] Bessen and Meurer argue that "the abstractness of software technology inherently makes it more difficult to place limits on abstract claims in software patents."[17] Given that patent claim drafting is itself largely an exercise in abstraction, however, it is not immediately clear why the abstract nature of software should pose a special problem for the determination of patent scope. In fact, computer scientists and software engineers are accustomed to thinking and communicating precisely about levels of abstraction in software and, as I have indicated previously[18] and will reemphasize herein,[19] this precision can be brought to bear on the problem of defining patent scope. Bessen and Meurer attempt to illustrate the difficulties caused by the "abstractness of software technology" with two examples of algorithms whose equivalents (in some mathematical sense) may be prohibitively difficult to recognize during the examination or term of a patent. Section I of this Article examines these examples and demonstrates that neither of them actually supports Bessen and Meurer's stated concern.

Klemens finds fault with the Federal Circuit's departure from longstanding doctrine that has regarded mathematical formulas as "abstract ideas" to be excluded from patentable subject matter.[20] According to Klemens, a claim to a programmed computer should be unpatentable whenever the program is the only innovative element, because every computer program is "nothing but a mathematical equation."[21] Klemens attempts to support this characterization by loosely paraphrasing a classical result in theoretical computer science, the Church-Turing Thesis,[22] and stating — without proof — sweeping and conclusory propositions that supposedly follow as corollaries from Alonzo Church's and Alan Turing's intricate mathematical theories of recursive functions.[23] The ultimate effect, if not the purpose, of Klemens's appeal to deep theory is to dazzle the "non-

[16] *See* BESSEN & MEURER, *supra* note 6, at 201-14.

[17] *See id.* at 201.

[18] *See* Andrew Chin, *Computational Complexity and the Scope of Software Patents*, 39 JURIMETRICS 17 (1999).

[19] *See infra* text accompanying note 146.

[20] See KLEMENS, *supra* note 2, at 53-69; Klemens, *supra* note 4, at 10-20.

[21] See KLEMENS, *supra* note 2, at 65 (describing a programmed computer as a "state machine").

[22] *See* KLEMENS, *supra* note 1, at 26 (introducing the Church-Turing thesis); Klemens, *supra* note 4, at 9-10 (proceeding to discuss the implications of the thesis without stating the thesis itself).

[23] *See infra* Section II.C.

geeks"[24] rather than to prove any point. Section II of this Article shows that the Church-Turing Thesis actually applies to relatively few software-related inventions and does not speak to Klemens's proposed doctrinal reforms.

In summary, Bessen and Meurer argue, through their examples, that software inventions are inherently too abstract to describe their scope reliably in a patent claim, and in this respect are different enough from other inventions to require technology-specific treatment in patent doctrine. Klemens argues, through theory, that software inventions should be deemed so abstract to be unpatentable as a matter of law. Sections I and II of this Article show that both of these categorical arguments were presented without adequate factual support. These findings imply that the authors' proposals for software technology-specific patent law reform are actually grounded in empirical policy analyses, not categorical distinctions. The proposals therefore remain subject to empirical examination and critique as policy choices, and are unlikely to be achieved through judicially developed doctrines. They also highlight the need for precise language in the ongoing debate over patent reform, in which the meanings of legal, scientific and economic concepts are accurately informed by the understandings of their respective disciplines, rather than intuitions and analogies. Section III of this Article concludes with additional comments and directions for further work.

I. BESSEN AND MEURER

Without singling out any particular area of technology, courts and scholars have long described the ambiguity of claim language as a pervasive impediment to the notice function of patents.[25] Even Bessen and Meurer

[24] KLEMENS, *supra* note 1, at 24.

[25] *See, e.g.*, Autogiro Co. of Am. v. United States, 384 F.2d 391, 396-97 (Ct. Cl. 1967) (describing claim drafting as a "conversion of machine to words [that] allows for unintended idea gaps which cannot be satisfactorily filled"); Gretchen Ann Bender, *Uncertainty and Unpredictability in Patent Litigation: The Time is Ripe for a Consistent Claim Construction Methodology*, 8 J. INTELL. PROP. L. 175, 209 (2001) (arguing that "claim language is often inherently ambiguous"); Michael Risch, *The Failure of Public Notice in Patent Prosecution*, 21 HARV. J.L. & TECH. 179, 192 (2007) (citing 30% appellate reversal rate of district court claim construction rulings); *see also* United Carbon Co. v. Binney & Smith Co., 317 U.S. 228, 236 (1942) ("A zone of uncertainty which enterprise and experimentation may enter only at the risk of infringement would discourage invention only a little less than unequivocal foreclosure of the field."); Merrill v. Yeomans, 94 U.S. 568, 573-74 (1877) ("The public should not be deprived of rights supposed to belong to it, without being clearly told what it is that limits these rights. The genius of the inventor, constantly making improvements in existing patents — a process which gives to the patent system its greatest value — should not be restrained by vague and indefinite

acknowledge that the "problems of abstract patent claims clearly apply to a broad range of technologies in addition to software."[26] Nevertheless, they argue that "software patents are different" in that "the abstractness of software technology inherently makes it more difficult to place limits on abstract claims in software patents."[27]

Specifically, Bessen and Meurer are concerned that computer algorithms have "disparate representations" that may be impossible for even computer scientists to recognize at the time a patent issues, thereby "creat[ing] critically difficult problems for the notice function of the patent system."[28]

To illustrate this difficulty, Bessen and Meurer first discuss an "equivalen[ce]" between two examples of a large class of apparently intractable computational problems known as NP-complete problems.[29] Stated informally, the *traveling-salesman problem* is to find the shortest tour that visits each of a list of cities (in any order), given the known distances between each pair of cities. The *map-coloring problem* is to paint the regions of a given map with a minimal number of colors so that no two adjacent regions are the same color. Bessen and Meurer write:

> [T]he 'traveling-salesman' problem, which is used for routing delivery trucks among other things, is more or less equivalent to the 'map-coloring' problem and a whole range of other problems. This means that an algorithm for solving the traveling-salesman problem is also, if worded broadly enough, an algorithm for doing map coloring.[30]

The authors' concern here is that a patent claim directed specifically to a algorithm for solving one NP-complete problem might eventually be construed more abstractly as covering the "whole range" of algorithms for solving NP-complete problems.

Bessen and Meurer's second illustration concerns a patented linear programming algorithm whose "equivalence" to prior art methods was only discovered by other computer scientists in 1986, two years after the algorithm was published:

> The patent is sometimes cited as an example of what a software patent should be: a highly specific, nontrivial contribution to practical knowledge. Yet serious questions

descriptions of claims in existing patents from the salutary and necessary right of improving on that which has already been invented.").

[26] BESSEN & MEURER, *supra* note 6, at 201.

[27] Id.

[28] Id. at 202.

[29] Id.

[30] Id. at 201-02.

> exist as to the boundaries of even this patent, questions as to whether its claims are truly novel, and whether [the inventor Narendra] Karmarkar actually "possessed" all the technologies claimed. One problem is that Karmarkar's algorithm seemed similar to technologies developed during the 1960s. In 1986, computer scientists demonstrated, in fact, that Karmarkar's algorithm is equivalent to a class of techniques that was known and applied to linear problems during the 1960s. Moreover, after this equivalence was demonstrated, computer scientists began applying algorithms based on these older techniques to linear programming problems — some of these algorithms appeared to work better than the Karmarkar-AT&T approach. . . .
>
> Given these facts, consider the difficulty of determining the boundaries of this patent. Would anyone have seen Karmarkar's algorithm as novel in light of the techniques used in the 1960s? Certainly not after 1986, when their equivalence was proved. But even in 1984, computer scientists might well have had doubts, yet they would have been unable to make a certain comparison. . . . Similarly, would AT&T have been able to assert its patent successfully against people who used linear-programming techniques based on those used in the 1960s? Apparently, AT&T was able to obtain a cross-license from IBM, which had used these older techniques.
>
> The abstractness of the patented algorithm means that these determinations cannot be made with certainty.[31]

Here, the authors' concern is essentially that Karmarkar's claims, being directed to an algorithm, were necessarily drafted in terms that were so abstract that they obscured the relevance of certain prior art techniques to the patentability analysis, thereby resulting in the patenting of an invention of dubious novelty.

The basic problem with Bessen and Meurer's illustrations is that in each case the computational concept of "equivalence" does not correspond to the relevant legal standard of equivalence pertaining to a claimed invention. As the following technical discussion should make clear, it is highly implausible that an algorithm for solving any particular NP-complete problem would be patented under a claim that was only later understood to cover solutions to the general class of NP-complete problems, either literally or by equivalents. It should also become apparent that that the

[31] Id. at 202-03.

aforementioned mathematical programming techniques from the 1960s would not have sufficed as prior art to show that Karmarkar's algorithm was anticipated or obvious in 1984.

A. "Equivalences" Among Algorithms for NP-Complete Problems

The mathematical theory of computational complexity has historically supplied computer science with the rigor necessary to study computational problems and algorithms. One of the most important milestones in this field came in 1971, with the publication by Stephen Cook of a set of results concerning the apparent intractability of a large class of computational problems.[32] From Cook's theory emerged the understanding that many well-known problems, such as the traveling salesman and map coloring problems, are nearly enough equivalent that each is equally resistant to solution by an efficient (i.e., polynomial time) algorithm.[33] To formalize this notion of equivalence, it is necessary to understand three important concepts from computational complexity theory, *polynomial-time algorithms*, *NP-completeness*, and *polynomial-time reductions*.

1. Polynomial-Time Algorithms

The standard basis for measuring the computational complexity of an algorithm is the *Turing machine*, an abstract model of computation. A Turing machine consists of a read-write head, an infinite tape consisting of spaces for symbols that can be read or written, and a finite state control that can move the head one space to the left or right along the tape depending on the machine's state.[34] A program for a Turing machine essentially consists of a transition function that determines the machine's next step (writing, moving and changing state) depending on the machine's current state and the symbol currently being read.[35] The program also specifies two final states, "yes" and "no," for which the machine's next step is simply to halt the computation.[36] For a given program, whether the Turing machine eventually halts in a "yes" state or a "no" state depends on the initial

[32] *See* Stephen A. Cook, *The Complexity of Theorem-Proving Procedures*, PROC. 3RD ANN. ACM SYMP. ON THEORY OF COMPUTING 151 (1971).

[33] *See* MICHAEL R. GAREY & DAVID S. JOHNSON, COMPUTERS AND INTRACTABILITY: A GUIDE TO THE THEORY OF NP-COMPLETENESS 1-14 (1979).

[34] *See id.* at 23.

[35] *See id.*

[36] *See id.* at 23-24.

content of the tape, when read relative to the initial position of the head.[37] (A relatively simple example of a Turing machine program is provided in the Appendix.)

A Turing machine is a relatively weak computational model, but powerful enough to support a stable classification of problems as tractable or intractable.[38] For such a complexity analysis to proceed, the problem in question must be restated as a *decision problem* that can be answered with a "yes" or "no," and there must be a system for *encoding* any instance of the problem as a string of symbols that can be read from a Turing machine tape.[39] A decision problem Π is said to be *tractable* if there exists a *polynomial-time algorithm* for solving it; i.e., there is a polynomial p such that there exists a Turing machine program that halts with the correct decision for each instance of Π in no more than $p(n)$ steps, where n is the size of (i.e., the number of symbols in) the encoded instance.[40] The class of tractable problems is referred to simply as P. Π is said to be *intractable* if there exists no polynomial-time algorithm for solving it.

The class P of tractable problems as defined here turns out to be the same regardless of the underlying computational model,[41] and corresponds to a longstanding consensus among computer scientists about the feasibility of solving increasingly large-scale problems on increasingly powerful real-world machines.[42] This consensus dates back to the 1960s, when papers by computer scientists Alan Cobham[43] and Jack Edmonds[44] famously highlighted the fundamental importance of the distinction between polynomial-time ("good") algorithms and less efficient ("bad") algorithms. Their basic point was that as the processing speed of available computers increases exponentially over time — an empirical observation popularly known as Moore's Law — it is polynomial-time algorithms, and only polynomial-time algorithms, that are capable of harnessing these improvements to solve exponentially larger problem instances.[45] For example, following a 100-factor speedup in processing speed, an algorithm

[37] *See id.*

[38] *See id.* at 7-8.

[39] *See id.* at 9-11.

[40] *See id.* at 26-27.

[41] *See id.* at 10 ("All the realistic models of computers studied so far . . . are equivalent with respect to polynomial time complexity. . . .").

[42] *See id.* at 6-11.

[43] Alan Cobham, *The Intrinsic Computational Difficulty of Functions*, in PROC. 1964 INT'L CONGRESS FOR LOGIC METHODOLOGY AND PHILOSOPHY OF SCIENCE (Y. Bar-Hillel, ed. 1964), at 24.

[44] Jack Edmonds, *Paths, Trees, and Flowers*, 17 CANADIAN J. MATH. 449 (1965).

[45] *See* Andrew Chin, *Computational Complexity and the Scope of Software Patents*, 39 JURIMETRICS 17, 25-26 (1998).

that takes n^2 steps to solve instances of size n will be able to handle instances 10 times as large as before, but an algorithm that takes 2^n steps will only be able to handle instances that are incrementally (i.e., an additional 6.64 input symbols) larger.[46]

2. NP-Completeness

It is relatively straightforward to prove the complexity and correctness of an efficient algorithm for solving a problem, and thereby to show that the problem is tractable (i.e., in P). As is often the case, however, proving the negative is considerably more difficult. The most that can be said about the computational difficulty of solving many problems is that a polynomial-time algorithm is very unlikely to exist.

Even without formal proofs of intractability, computer scientists have managed to show that some computational problems are relatively difficult. They have focused these efforts on the class NP, which consists of those problems for which a polynomial-time algorithm might conceivably exist (whether or not one has already been discovered).[47] The hardest problems in NP, including such familiar examples as the traveling-salesman and graph-coloring problems, are known as *NP-complete* problems.

As illustrated in Figure 1, the class of NP-complete problems has the special property that if any NP-complete problem is tractable, then all problems in NP are tractable (i.e., P=NP). Thus a proof that a problem is NP-complete serves to demonstrate that the problem is intractable, provided that P≠NP. NP-complete problems are sometimes referred to as "equivalent" because of this common property; it is in this sense that Bessen and Meurer's use of the term is apt.[48]

[46] See id. at 8.

[47] In the Turing machine model, the behavior of such a hypothetical polynomial-time algorithm is formally equivalent to a *nondeterministic algorithm* in which a "guessed structure" of polynomial size may be appended to the input to aid the computation, thereby reducing the problem to one of verification. *See* GAREY & JOHNSON, *supra* note 33, at 27-32.

[48] See supra text accompanying note 33.

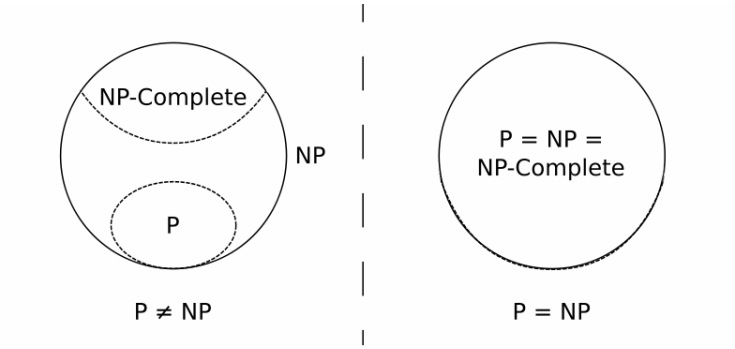

Figure 1. Relationships among the complexity classes P, NP and NP-complete in two alternative states of the world.

It is unknown whether P=NP or P≠NP; in fact, this has become one of the most important open questions in mathematics and computer science.[49] Until it is established that P≠NP, the traveling-salesman and map-coloring problems and thousands of other NP-complete problems will lack an efficient solution, yet will not be known to be intractable.

Failure to establish that P=NP, on the other hand, signifies the failure of the entire scientific community to find a polynomial-time algorithm for solving any one of the thousands of NP-complete problems. Even though computer scientists are certainly well aware that "absence of evidence is not evidence of absence,"[50] many have viewed the absence of an efficient solution to any NP-complete problem as evidence that none can exist (i.e., that P≠NP).[51] This view was expressed in whimsical terms by Garey and Johnson's classic treatise on NP-completeness[52] in 1979, which explained that if tasked with designing an efficient algorithm for some new computational problem, say, the "bandersnatch problem,"

> you might be able to prove that the bandersnatch problem is NP-complete and, hence, that it is equivalent to all these

[49] *See, e.g.*, Michael Sipser, *The History and Status of the P versus NP Question*, in PROC. 24TH ANNUAL ACM SYMP. ON THEORY OF COMPUTING (1992), at 603 (describing it as "one of the most important problems in contemporary mathematics and theoretical computer science"); Clay Mathematics Institute, P vs. NP Problem, <http://www.claymath.org/millennium/P_vs_NP/> (visited July 15, 2008) (describing it as one of seven "Millennium Problems" for which the Institute offered a standing prize of $1 million in 2000).

[50] *See, e.g.*, Hall v. Baxter Healthcare Corp., 947 F. Supp. 1387, 1470-71 (D. Or. 1996) (describing this as "one of the major tenets of science").

[51] *See, e.g.*, William A. Gasarch, *Guest Column: The P=?NP Poll*, 33 SIGACT NEWS 34 (2002).

[52] *See* GAREY & JOHNSON, *supra* note 33.

> other hard problems. Then you could march into your boss's office and announce: "I can't find an efficient algorithm, but neither can all these famous people." At the very least, this would inform your boss that it would do no good to fire you and hire another expert on algorithms.[53]

Three decades later, both the list of "famous people" and the universe of NP-complete problems they have failed to conquer have grown dramatically, further bolstering the case that P≠NP.

In the computer science research community, the view that the edifice of NP-completeness has grown too formidable to collapse is dominant but not universal. In a recent survey of prominent computer scientists, a substantial majority (61%) predicted an eventual proof that P≠NP, while only a small minority (9%) predicted that it will turn out that P=NP.[54] Few (30%) expected the question to be resolved by the year 2029.[55]

The P vs. NP problem appears from the survey to have humbled many of the most accomplished computer scientists of our time. Turing Award winner Richard Karp responded, "My intuitive belief is that P is unequal to NP, but the only supporting arguments I can offer are the failure of all efforts to place specific NP-complete problems in P by constructing polynomial-time algorithms."[56] While taking a contrary view, Senior Whitehead Prize winner Bela Bollobas was equally tentative, describing himself as "on the loony fringe of the mathematical community" in believing "not too strongly" that a proof that P=NP would appear within twenty years.[57] Jim Owings, an emeritus professor at the University of Maryland, was more philosophical about the state of his knowledge: "It is the greatest unsolved problem in mathematics. . . . It is the raison d'etre of abstract computer science, and as long as it remains unsolved, its mystery will ennoble the field."[58]

Even respondents who expected an eventual proof that P=NP expressed doubt that such a result would enable the solution of all NP-complete problems in practice. Donald Knuth, the founder of the modern science of algorithms, wrote that he expects P=NP to be the consequence of an indirect proof, so that "we will never know" the complexity of an NP-complete problem.[59] Other respondents expected any proof of P=NP to result in

[53] Id. at 1-3.

[54] *See* Gasarch, *supra* note 51, at 36.

[55] *See id.*; *but see* id. at 38 (noting John Conway's opinion that "this shouldn't really be a hard problem; it's just that we came late to this theory, and haven't yet developed any techniques for proving computations to be hard.").

[56] Id. at 41.

[57] Id. at 37.

[58] Id. at 43.

[59] See id. at 41.

polynomial time bounds for NP-complete problems whose degrees and/or coefficients are too high to assure the existence of a practical algorithmic solution.[60]

3. Polynomial-Time Reductions

The distinction between problems known and not known to have polynomial-time algorithms has special significance because of Moore's Law and the theory of NP-completeness. Since polynomials with high degrees or coefficients can grow very quickly, however, a problem may be in P yet lack a practical algorithmic solution even for small inputs. Computational complexity theory must therefore also be concerned with achieving the lowest possible upper bounds on the time required to solve tractable problems. An eventual proof that P=NP would imply that all NP-complete problems could be solved by polynomial-time algorithms, but it would not immediately imply the existence of practical algorithms for solving all NP-complete problems. Instead, it would instigate a further program of research into the complexity of individual NP-complete problems.[61]

Much work on the complexity of specific NP-complete problems has already been done. The typical procedure for proving a decision problem $\Pi \in \text{NP}$ to be NP-complete is to show that Π is at least as unlikely to be in P as some other problem Π_0 that has previously been shown to be NP-complete. This involves constructing what is known as a *polynomial-time reduction* from Π_0 to Π, i.e., a polynomial-time computable function f that maps each possible instance x of Π_0 into a corresponding instance $f(x)$ of Π that yields the same yes-or-no decision.[62] The idea is that any polynomial-time algorithm that solves Π could be used as a polynomial-time solution for Π_0: given an input x to problem Π_0, simply calculate the transformed value $f(x)$ in polynomial time, and then solve Π in polynomial time.[63]

Stephen Cook's 1971 article[64] laid the groundwork for this research by identifying and proving the first problem to be NP-complete from first principles. The problem, now known in the literature as SATISFIABILITY

[60] See id. (noting comments of Vladik Kreinovich and Clyde Kruskal).

[61] For example, *see infra* text accompanying notes 85-86 (discussing Karmarkar's improvement of Khachiyan's upper bound for the complexity of linear programming).

[62] *See* GAREY & JOHNSON, *supra* note 33, at 34.

[63] *See id.* at 34-35.

[64] *See* Cook, *supra* note 32.

(or SAT for short), is to determine whether a Boolean formula on n true-or-false variables, given in disjunctive normal form (i.e., an AND of OR-clauses on the n variables and their negations), can be made true by some assignment of values to the variables.[65] Cook's result[66] essentially constructed a polynomial-time reduction from any problem in NP[67] to SATISFIABILITY.[68] Cook's article then went on to show, *inter alia*, a polynomial-time reduction from SATISFIABILITY to a second NP-complete problem, now referred to as SUBGRAPH ISOMORPHISM.[69] Soon thereafter, Richard Karp published an article presenting proofs of the NP-completeness of twenty-one well-known problems in computer science, including 3SAT, a variant of SATISFIABILITY in which each OR-clause consists of exactly three terms.[70]

Over the years, thousands of problems have been added to a growing tree of NP-complete problems, each linked to a previous member of the class by a polynomial-time reduction.[71] Between any two NP-complete problems on the tree, it is possible to trace a chain of polynomial-time reductions that demonstrates their "equivalence," in the sense that the two problems are equally unlikely to be tractable.[72] If used in practice, however, polynomial-time reductions can generate significant overheads, both in the time required to calculate the transformed inputs and in the size of the transformed inputs themselves. Where several polynomial-time reductions are applied in succession, these overheads will be compounded.

To illustrate the overheads that may result from a polynomial-time reduction, consider another of Karp's problems, known as VERTEX COVER. The problem may be stated as follows: Given a graph of N vertices and M edges and an integer $n < N$, is there some subset of n vertices that includes at least one endpoint of every edge in the graph?[73] Garey and Johnson present a proof that VERTEX COVER is NP-complete by presenting a polynomial-time reduction f from 3SAT to VERTEX

[65] *See* GAREY & JOHNSON, *supra* note 33, at 39 (defining SATISFIABILITY).

[66] *See* Cook, *supra* note 32, at 152-53 (proving Theorem 1).

[67] *See supra* note 47.

[68] *See* GAREY & JOHNSON, *supra* note 33, at 44 (restating Cook's result as showing the existence of a polynomial-time reduction f_L from a nondeterministic Turing machine computation recognizing the language L to SAT).

[69] *See* Cook, *supra* note 32, at 153-54 (proving Theorem 2).

[70] *See* Richard M. Karp, *Reducibility Among Combinatorial Problems*, in COMPLEXITY OF COMPUTER COMPUTATIONS 85 (R.E. Miller & J.W. Thatcher eds. 1972).

[71] For early versions of this tree, *see, e.g.*, GAREY & JOHNSON, *supra* note 33, at 47; Karp, *supra* note 70, at 96.

[72] *See supra* text accompanying notes 33 and 48.

[73] *See* Karp, *supra* note 70, at 94 (referring to the problem as NODE COVER); GAREY & JOHNSON, *supra* note 33, at 46.

COVER. Figure 2 illustrates how the reduction f operates to convert the 3SAT instance $(u_1 \vee \neg u_3 \vee \neg u_4) \wedge (\neg u_1 \vee u_2 \vee \neg u_4)$ into an instance of VERTEX COVER with $n = 8$.[74]

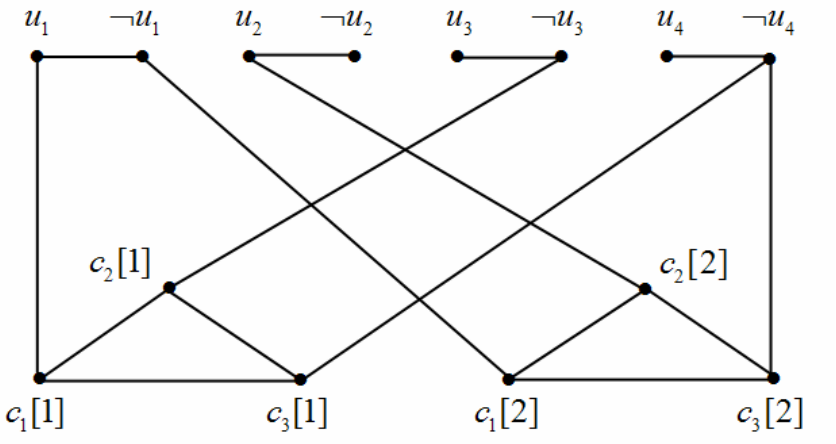

Figure 2. VERTEX COVER instance resulting from the 3SAT instance $(u_1 \vee \neg u_3 \vee \neg u_4) \wedge (\neg u_1 \vee u_2 \vee \neg u_4)$.

For each variable that appears in the 3SAT instance, the VERTEX COVER instance has two vertices, representing the variable and its negation, connected by an edge. Each clause in the 3SAT instance is represented by three vertices $c_1[i], c_2[i], c_3[i]$, connected by three edges to form a triangle. Finally, each of the three vertices representing each clause is connected to the vertex that represents the corresponding variable (or its negation) as it appears in the 3SAT instance.[75]

While this polynomial-time reduction from 3SAT to VERTEX COVER is simple and even elegant, it requires some computational time and some expansion in the instance size. A person in possession of an efficient algorithm for VERTEX COVER might well wonder if there were a faster way of solving 3SAT directly, instead of first having to convert each instance of 3SAT to an instance of VERTEX COVER to be solved. This concern about the overhead of polynomial-time reductions becomes even

[74] The required size of the vertex cover (n=8) is determined by adding the number of variables (four) to twice the number of clauses (two) in the given 3SAT instance. *See* GAREY & JOHNSON, *supra* note 33, at 55.

[75] *See id.* at 54-56.

more warranted when more distant problems on the tree of NP-complete problems are considered.

4. Bessen and Meurer's "Equivalence"

According to Bessen and Meurer, a software developer trying to solve the map-coloring problem might inadvertently infringe a patent claim directed to a traveling-salesman algorithm (or vice versa) because of the "equivalence" between the two problems. This possibility, the authors contend, is illustrative of an inherent and unique deficiency in the notice function of software patent claims — at least those that are "worded broadly enough."[76] Given the context provided above, however, it is difficult to imagine that such a problematic ambiguity in the scope of a software patent claim would ever arise.

In understanding the effect that the equivalence among NP-complete problems might have on software patent scope, it is important to distinguish between problems and algorithms. A chain of polynomial-time reductions that demonstrates the equivalence between two NP-complete problems does not thereby show that all algorithms for solving those problems are equivalent. It shows only that given a hypothetical algorithm for solving one problem, it is possible to derive a particular algorithm for solving the other. Moreover, the derived algorithm provides only an indirect solution that may be inefficient and even impractical.

As shown in Figure 3, the chains of polynomial-time reductions from MAP COLORING to TRAVELING SALESMAN and vice versa both involve several links.

[76] *See supra* text accompanying notes 28-30.

MAP COLORING $\propto$ SATISFIABILITY $\propto$ 3SAT $\propto$
VERTEX COVER $\propto$ HAMILTONIAN CIRCUIT $\propto$
TRAVELING SALESMAN

TRAVELING SALESMAN $\propto$ SATISFIABILITY $\propto$ 3SAT
$\propto$ MAP COLORING

Figure 3. Chains of polynomial-time reductions proven between MAP COLORING AND TRAVELING SALESMAN.

For Bessen and Meurer's scenario to take place, it would require more than the fact that a claim directed to a polynomial-time traveling-salesman algorithm was "worded broadly enough." It would require that an independently discovered algorithm for the map-coloring problem correctly implemented each of the detailed and intricate polynomial-time reductions in the chain, as well as each of the steps recited in the claim to the traveling-salesman algorithm. A more broadly worded claim to a traveling-salesman algorithm might cover the use of the recited computational steps across a wider range of fields, but it cannot widen the range of conditions under which a polynomial-time reduction is logically correct. (From the description above of one such reduction, from VERTEX COVER to 3SAT,[77] it should be clear that these conditions are mathematically well-defined and precise.) It seems most unlikely that an independent scientist, seeking a direct and efficient solution to the map-coloring problem, would in passing replicate the details (and assume the overhead) of the entire chain of reductions to the traveling-salesman problem.

It is also worth noting here that the equivalence among NP-complete problems due to polynomial-time reductions does not imply equivalence between specific algorithms for solving those problems under patent law's doctrine of equivalents. Under the doctrine of equivalents, an accused device that does not fall literally within the scope of a claim may nevertheless be found to infringe "if it performs substantially the same function in substantially the same way to obtain the same result";[78] this "triple identity" determination is to be applied to a claim "as an objective inquiry on an element-by-element basis."[79] A chain of polynomial-time reductions, however, does not translate an algorithm for solving one problem into an algorithm for solving another on a step-by-step or element-

[77] See supra text accompanying note 75.

[78] Graver Tank & Mfg. Co. v. Linde Air Products Co., 339 U.S. 605, 608 (1950) (quoting Sanitary Refrigerator Co. v. Winters, 280 U.S. 30, 42 (1929)).

[79] Warner-Jenkinson Co., Inc. v. Hilton Davis Chemical Co., 520 U.S. 17, 40 (1997).

by-element basis; rather, it converts an instance of one problem into an instance of the other. By the time the steps of the original algorithm are to be performed on the converted instance, all of the polynomial-time reductions have already been completed, and can play no part in a step-by-step analysis of equivalence to the original algorithm. Thus, in Bessen and Meurer's scenario, a correct implementation of the entire chain of polynomial-time reductions by the accused algorithm would be a prerequisite not only for a finding of literal infringement, but for a finding of infringement by equivalents as well. As discussed above, it is highly unlikely that an independently designed algorithm would happen to follow this approach.

Finally, it should be remembered that the notion of "equivalence" via polynomial-time reductions between a newly discovered map-coloring algorithm and a previously claimed traveling-salesman algorithm (or vice versa) presupposes a state of the world in which polynomial-time algorithms for NP-complete problems are known to exist; i.e., that P=NP. As we have seen, few computer scientists believe this to be the case.[80] Moreover, it is almost unimaginable that anyone who discovered a polynomial-time traveling-salesman algorithm, thereby proving that P=NP, would simply patent the algorithm and fail to announce the broader result. In sum, software developers have very little to fear from inadequately noticed patents on polynomial-time algorithms for NP-complete problems.

B. Linear Programming and Karmarkar's Algorithm

1. Karmarkar's Contributions

Bessen and Meurer's second illustration of the problematic "abstractness of software technology" concerns Narendra Karmarkar's celebrated (and patented) algorithm for *linear programming*, which solves a form of constrained optimization problem commonly used in operations research and public policy analysis. The linear programming problem is to maximize (or, alternatively, to minimize) the value of a given linear function in real variables (the *objective function*), where the variables are subject to a system of linear inequalities (the *constraints*).[81] The more general problem in which the objective function and constraints may be nonlinear is referred to as *mathematical programming*; a mathematical programming problem that is not a linear programming problem is known

[80] *See supra* text accompanying note 54

[81] *See* ANTHONY V. FIACCO & GARTH P. MCCORMICK, NONLINEAR PROGRAMMING: SEQUENTIAL UNCONSTRAINED MINIMIZATION TECHNIQUES 1 (1968).

as a *nonlinear programming* problem.[82]

When Cook and Karp published their first results on the theory of NP-completeness in 1971-72,[83] linear programming had already been long recognized as an important computational problem,[84] but no one knew then whether or not it could be solved in polynomial time. It was not until 1979 that Leonid Khachiyan showed linear programming to be tractable by presenting an algorithm that required at most $O(n^6L^2)$ time to solve a problem with n variables and L input bits.[85]

Karmarkar announced his algorithm in May 1984 at the Association for Computing Machinery's annual symposium on theoretical computer science[86] and submitted a revised and extended exposition of the algorithm to the mathematics journal *Combinatorica* in August 1984 for publication later that year.[87] While his results came too late to be credited with resolving the question of linear programming's tractability, they were groundbreaking in other ways. Previous linear programming algorithms, including Khachiyan's, searched for possible solutions (known as "iterates") by moving from corner to corner around the boundary of the n-dimensional region (known as a "polytope") defined by the constraints of the problem. Karmarkar's insight was that interior points provide richer information than boundary points on which direction will lead to the greatest improvement in the objective function. A prior art "exterior-point" method and Karmarkar's "interior-point" method are contrasted in Figure 4.

[82] *See id.*

[83] *See supra* text accompanying notes 64-70.

[84] See generally VERA RILEY & SAUL I. GASS, LINEAR PROGRAMMING AND ASSOCIATED TECHNIQUES; A COMPREHENSIVE BIBLIOGRAPHY ON LINEAR, NONLINEAR, AND DYNAMIC PROGRAMMING (1958) (reviewing research as of 1958).

[85] *See* Leonid G. Khachiyan, *A Polynomial Algorithm in Linear Programming*, 244 DOKLADY AKADEMIIA NAUK. SSSR 1093 (1979), *translated in* 20 SOVIET MATH. DOKLADY 191 (1979). The parameter L accounts for the complexity of real-number calculations that may require an arbitrary degree of precision.

[86] *See* Narendra Karmarkar, *A New Polynomial-Time Algorithm for Linear Programming*, PROCEEDINGS 16TH ACM SYMP. ON THEORY OF COMPUTING 302 (1984).

[87] *See* Narendra Karmarkar, *A New Polynomial-Time Algorithm for Linear Programming*, 4 COMBINATORICA 373 (1984).

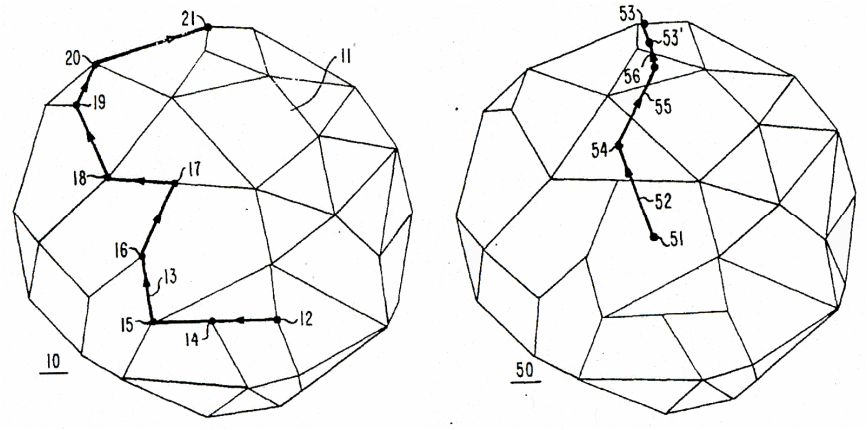

Figure 4. Comparison between George Dantzig's simplex exterior-point algorithm (left) and Karmarkar's interior-point algorithm (right) for linear programming. In the simplex method, the search proceeds entirely on the boundary of the polytope (from the initial iterate (point 12) to the solution (point 21). In Karmarkar's method, the search proceeds within the interior of the polytope from the initial iterate (point 51) until a solution is reached that satisfies the condition for termination (point 53).[88]

At each iteration, Karmarkar's algorithm performs a projective transformation on the polytope so that the previous iterate, a boundary point, is mapped into the interior of the transformed polytope.[89] From that interior point, the algorithm finds the next iterate by moving along a line, in the direction that maximizes the objective function, until it reaches the boundary.[90] By following this more efficient approach, Karmarkar's algorithm achieves a worst-case running time of $O(n^{3.5}L^2)$, a vast improvement over Khachiyan's algorithm for practical purposes.[91] Karmarkar's algorithm also has the virtue that it is relatively easy to implement.[92]

Karmarkar filed a U.S. patent application on Apr. 19, 1985 titled "Methods and Apparatus for Efficient Resource Allocation."[93] The patent

[88] *See* U.S. Patent No. 4,744,028, at cols. 2-4.

[89] *See* Michael J. Todd, *The Many Facets of Linear Programming*, 91 MATH. PROGRAMMING SERIES B 417, 427 (2002).

[90] *See id.*

[91] *See* Karmarkar, *supra* note 86, at 302.

[92] *See, e.g.*, E.R. Swart, *How I Implemented the Karmarkar Algorithm in One Evening*, 15 APL QUOTE QUAD 13 (1985) (providing source code of a 92-line program implementing the Karmarkar algorithm in Array Processing Language).

[93] *See* U.S. Patent No. 4,744,028.

issued on May 10, 1988 and was assigned to his employer, AT&T Bell Laboratories.[94]

2. Doubts as to Karmarkar's Contributions

According to Bessen and Meurer, the validity of Karmarkar's patent is called into doubt by both prior and subsequent developments. They correctly note that the use of interior-point methods to solve linear programming problems was not new in 1984, but (as Philip Gill et al. documented in 1986) had a long and distinguished history dating back to the 1940s and 1950s, including efforts by John von Neumann, Alan Hoffman, Charles Tompkins, and Ragnar Frisch.[95] In practice, these earlier interior-point methods were not competitive with George Dantzig's simplex algorithm, an exterior-point method that was known to have worst-case exponential running time[96] but, because of its conceptual simplicity, was considered acceptable for reasonably small problems.[97] (They also did not succeed in developing a polynomial-time algorithm for linear programming; that achievement would be left to Khachiyan in 1979.[98]) Accordingly, researchers found it more fruitful to investigate the application of interior-point methods to nonlinear programming. By 1968, when operations researchers Anthony Fiacco and Garth McCormick published their treatise on nonlinear programming, their presentation of interior-point methods and related results constituted one full chapter and parts of four others.[99]

In the years following the publication of Karmarkar's algorithm, some researchers began to identify connections between the earlier work focused on nonlinear programming and Karmarkar's more recent work on linear programming.[100] (It is worth noting that Karmarkar himself did not

[94] *See id.*

[95] *See* Philip E. Gill *et al.*, *On Projected Newton Barrier Methods for Linear Programming and an Equivalence to Karmarkar's Projective Method*, 36 MATH. PROGRAMMING 183, 184 (1986) (citations omitted).

[96] *See* Victor Klee & George J. Minty, Jr., *How Good is the Simplex Method?*, in INEQUALITIES III, at 159 (O. Sisha ed. 1972).

[97] The simplex algorithm is still the only computational linear-programming method presented in introductory operations research textbooks, *see, e.g.*, David R. Anderson et al., AN INTRODUCTION TO MANAGEMENT SCIENCE: A QUANTITATIVE APPROACH TO DECISION MAKING, chs. 17-18 (2007), and remains "the method of choice" for many applications. *See* Roy Marsten et al., *Interior Point Methods for Linear Programming: Just Call Newton, Lagrange, and Fiacco and McCormick!*, 20 INTERFACES 105, 115 (1990).

[98] *See supra* note 85 and accompanying text.

[99] *See* FIACCO & MCCORMICK, *supra* note 81, at chs. 3, 5-8.

[100] *See* Marsten, *supra* note 97, at 105-06 (1990) (noting that shortly after 1984, "[m]any others worked on bringing Karmarkar's method, which at first appeared to be

acknowledge any such connections in his patent application or either of his 1984 publications.[101]) In their 1986 paper,[102] Gill et al. note that Frisch's interior-point methods[103] allow for a choice of the direction the search algorithm is to take from one iterate to the next. One possible way of determining this direction is to minimize a quadratic approximation to a "barrier function" $F(x)$, defined by

$$F(x) = c^T x - \mu \sum_{j=1}^{n} \ln x_j ,$$

that incorporates both the problem's objective function and its constraints.[104] Gill et al. refer to this direction as the "Newton search direction" in honor of Sir Isaac Newton, who is credited with discovering this numerical approach to approximating the minima of differentiable functions.[105] Their main result is that for a particular value of the parameter μ, the Newton search direction is the same as the direction prescribed by Karmarkar's algorithm.[106] Gill et al. are careful to characterize their finding as "an existence result, showing that a special case of the [Newton] barrier method would follow the same path as the [Karmarkar] projective method."[107] In the article's introduction, however, they describe this result more broadly as "a formal equivalence between the Newton search direction and the direction associated with Karmarkar's algorithm."[108] The title of their article is broader still, suggesting equivalence not merely between the search directions employed by the respective methods, but between the methods themselves: "On Projected Newton Barrier Methods for Linear Programming and an Equivalence to Karmarkar's Projective Method."[109]

A 1990 article by Roy Marsten et al. also describes Gill et al.'s existence result in broad terms as "an equivalence between Karmarkar's method and projected Newton barrier methods."[110] In an elegant exposition, Marsten et al. outline the respective contributions of Fiacco and

coming completely out of left field, into our classical framework of optimization").

[101] None of Karmarkar's lists of references cites any of the literature on nonlinear programming. *See* U.S. Patent No. 4,744,028; Karmarkar, *supra* note 86 at 311; Karmarkar, *supra* note 87, at 395.

[102] *See* Gill, *supra* note 95.

[103] *See* K. Ragnar Frisch, *The Logarithmic Potential Method of Convex Programming* (1955), unpublished manuscript, University Institute of Economics, Oslo, Norway, *cited in* Gill, *supra* note 95.

[104] *See* Gill, *supra* note 95, at 185-86.

[105] *See id.* at 186.

[106] *See id.* at 190-91.

[107] *Id.* at 191.

[108] *Id.* at 184.

[109] *Id.* at 183.

[110] Marsten, *supra* note 97, at 106.

McCormick, Newton, and the eighteenth-century Italian mathematician Joseph-Louis Lagrange to the "special case of the [Newton] barrier method" identified by Gill et al.[111] They do this not only to present Gill et al.'s results to "a wider audience" in the operations research and management science community,[112] but to respond to what they saw as hubris on the part of Karmarkar and AT&T:

> In 1984, Narendra Karmarkar began the "new era of mathematical programming" with the publication of his landmark paper. Shortly thereafter his employer, AT&T, invited the professional mathematical programming community to roll over and die. Speaking as representatives of this community, we took this as rather a challenge.[113]

Accordingly, Marsten et al.'s title and abstract suggest an account of the "new era" in which Karmarkar's contributions may be rightly omitted as redundant:

> *Interior Point Methods for Linear Programming:*
> *Just Call Newton, Lagrange, and Fiacco and McCormick!*
>
> Interior point methods are the right way to solve large linear programs. They are also much easier to derive, motivate, and understand than they at first appeared. Lagrange told us how to convert a minimization with equality constraints into an unconstrained minimization. Fiacco and McCormick told us how to convert a minimization with inequality constraints into a sequence of unconstrained minimizations. Newton told us how to solve unconstrained minimizations. Linear programs are minimizations with equations and inequalities. Voila![114]

Marsten et al.[115] and other researchers (including Karmarkar himself[116]) also sought to improve the performance of Karmarkar's algorithm in cases where its calculations involved *sparse matrices*; i.e., matrices that have very few nonzero elements. By using fast sparse-matrix algorithms for "Cholesky factorization," an important subroutine used in the numerical solution of systems of linear equations, Marsten et al. were able to accelerate a procedure that accounts for about 90 percent of the running

[111] See id. at 106-08.
[112] *See id.* at 105.
[113] Id. (quotation unattributed in original).
[114] Id.
[115] *See id.* at 110-15.
[116] *See* Ilan Adler et al., *An Implementation of Karmarkar's Algorithm for Linear Programming*, 44 MATHEMATICAL PROGRAMMING 297 (1989) (naming Karmarkar as co-author); Ilan Adler et al., *Data Structures and Programming Techniques for the Implementation of Karmarkar's Algorithm*, 1 ORSA J. COMPUT. 84 (1989) (same).

time of Karmarkar's algorithm in practice,[117] thereby addressing the algorithm's "main weakness."[118]

In 1991, one of Marsten's coauthors, Matthew Saltzman, addressed his concerns about Karmarkar's algorithm and patent to an even wider community by posting a long message to the USENET discussion group sci.math.num-analysis summarizing the points made in the Marsten et al. article.[119] Saltzman also goes on question the novelty of, and sufficiency of disclosure in, Karmarkar's patent, and opines: "IMHO, this patent has not benefitted society. If faster LP [linear programming] algorithms are a benefit to society, then the benefit has occurred despite, not because of the patent."[120]

Given Gill et al.'s self-styled "equivalence" result, Marsten et al.'s apparent desire to write Karmarkar out of the mathematical programming history books, and subsequent advances in sparse-matrix calculations, it is easy to see how a casual reader of the technical literature might be left in doubt as to Karmarkar's contributions, and even be persuaded by a research scientist's uninformed legal opinion on the validity of Karmarkar's patent. For purposes of legal inquiry into the validity and scope of Karmarkar's patent, however, Bessen and Meurer need not have relied on these scientists' conclusory and somewhat misleading descriptions of "an equivalence between Karmarkar's method and projected Newton barrier methods" when a precise statement of Gill et al.'s actual existence result was already available.[121]

3. Karmarkar's Contributions Relative to the Prior Art

Contrary to Bessen and Meurer's assertion, Gill et al. did not "demonstrate[], in fact, that Karmarkar's algorithm is equivalent to a class

[117] *See* Marsten, *supra* note 97, at 111.

[118] *See id.* at 112.

[119] *See* Matthew Saltzman, *Re: The Karmarkar Algorithm (long)*, USENET (March 24, 1991), available via the Internet Archive <http://www.archive.org> at <http://www.cs.uvic.ca/~wendym/courses/445/06/interiorpoint.txt> (visited July 15, 2008) (cited by BESSEN & MEURER, *supra* note 6, at 312).

[120] *See* BESSEN & MEURER, *supra* note 6, at 202-03.

[121] *See supra* text accompanying note 31. Bessen and Meurer also appear to have been influenced by a posting by one of Marsten's co-authors to the USENET newsgroup sci.math.num-analysis, which questions the validity of Karmarkar's patent on novelty and disclosure grounds and cites the Gill and Marsten articles. *See* Matthew Saltzman, *Re: The Karmarkar Algorithm (long)*, USENET (March 24, 1991), available at <http://web.archive.org/web/20070223080354/http://www.cs.uvic.ca/~wendym/courses/445/06/interiorpoint.txt> (visited July 15, 2008) (cited by BESSEN & MEURER, *supra* note 6, at 312).

of techniques that was known and applied to linear problems during the 1960s."[122] Gill et al.'s existence result shows only that some of Frisch's and Fiacco and McCormick's methods can be tailored so that the resulting algorithm proceeds to search the same iterates as Karmarkar's algorithm. The necessary tailoring choices for this result, however, were not "known and applied" during the 1960s, and the available evidence (discussed below) strongly indicates that they were neither known nor obvious until Karmarkar's algorithm appeared. Thus, it would be blatant hindsight reconstruction to cite these choices, first publicly embodied in Gill et al.'s 1986 results, as "prior art" against a 1984 invention as Bessen and Meurer[123] and Saltzman[124] suggest.

In a 1994 treatise on interior point methods,[125] Dick den Hertog describes the range of design choices available to users of the Frisch/Fiacco-McCormick methods. Specifically, he identifies the following three "important elements in the design of such a method: (1) the iterative method used to (approximately) minimize the logarithmic barrier function; (2) the criterion for terminating the iterative minimizations; and (3) the updating scheme for the barrier parameter μ."[126]

Karmarkar's algorithm provided significant new advances with respect to all three of these design elements. First, as Michael Todd explains in a 2000 article,[127] Karmarkar's use of a projective transformation to "normalize" or "center" each iterate[128] represented a "very intriguing" new idea at the time for minimizing the logarithmic barrier function.[129] Second, Todd writes, another new idea was the use of "a nonlinear potential function, invariant under such transformations" to measure progress toward

[122] BESSEN & MEURER, *supra* note 6, at 202.

[123] See id. at 203 ("Would anyone have seen Karmarkar's algorithm as novel in light of the techniques used in the 1960s? Certainly not after 1986, when their equivalence was proved.").

[124] *See* Saltzman, *supra* note 121 ("A case can be made for prior art, though. . . Gill, et al. (1986) showed that in fact, Karmarkar's method was equivalent to a projected Newton barrier algorithm.").

[125] D. DEN HERTOG, INTERIOR POINT APPROACH TO LINEAR, QUADRATIC AND CONVEX PROGRAMMING (1994).

[126] Id. at 12.

[127] Todd, *supra* note 89.

[128] *See* U.S. Patent No. 4,744,028, at col. 10 ("This projective transformation can be thought of as an orthogonal transformation into the unit simplex, thereby achieving the normalizing or centering property.").

[129] *See* Todd, *supra* note 89, at 426-27. Todd adds that while "projective transformations are not used much in interior-point methods nowadays[, t]he key concept of making a transformation or changing the metric so that the current iterate is in some sense far from the boundary remains highly valuable." Id. at 427.

the termination condition.[130]

Finally, as Dave Bayer and Jeffrey Lagarias note, Karmarkar's updating scheme for μ differs from any method in which successive values of μ are determined as a function of the current iterate *y*,[131] such as that employed by Fiacco and McCormick.[132] There is nothing in any of Bessen and Meurer's sources to suggest that Karmarkar's scheme for μ was obvious prior to his invention. Even Gill et al. do not suggest any motivation for their choice of a particular value for μ beyond emulating the iterative behavior of Karmarkar's algorithm after the fact. In fact, in describing their main theorem as "an existence result," they note that "[t]his does not mean that the [Frisch/Fiacco-McCormick] barrier method *should* be specialized" by setting μ to this value.[133]

In the patent law context, the algorithm that results from Karmarkar's combination of design choices is most accurately characterized as a "range or value of a particular variable" that is included within a wider range disclosed in the prior art: namely, the entire class of Frisch/Fiacco-McCormick barrier methods.[134] An invention of this type is presumed obvious,[135] but this presumption may be rebutted by a showing that the range "produces new and unexpected results."[136]

As a general matter, there is ample evidence available to rebut the presumption of obviousness raised by the Frisch/Fiacco-McCormick prior art. Karmarkar's results — an easily implemented linear algorithm with a $O(n^{2.5})$-factor speedup over the previous world record, and the first interior-point algorithm to be shown to run in polynomial time — were, at the time, as new and unexpected as any developments in all of applied

[130] *See id.* at 427.

[131] *See* D.A. Bayer & J.C. Lagarias, *Karmarkar's Algorithm and Newton's Method*, 50 MATH. PROGRAMMING 291, 293 ("[I]f $\mu(y)$ is considered to be a function of *y* then the projected Newton method direction of [the barrier function] is usually not the projective scaling direction.").

[132] In the Fiacco-McCormick method, successive values of the barrier parameter (denoted therein by the variable *r* rather than μ) are chosen by a "computing rule" that chooses values for *r* that minimize the norm of the gradient of the barrier function $V(y)$ at the point *y*. *See* FIACCO & MCCORMICK, *supra* note 81, at 116.

[133] *See* Gill, *supra* note 95, at 191.

[134] *See, e.g.*, Haynes Int'l, Inc. v. Jessop Steel Co., 8 F.3d 1573, 1577 n. 3 (Fed.Cir.1993) ("[W]hen the difference between the claimed invention and the prior art is the range or value of a particular variable, then a prima facie rejection is properly established when the difference in range or value is minor.") (emphasis omitted).

[135] *See id.*

[136] Ormco Corp. v. Align Technology, Inc., 463 F.3d 1299, 1311 (Fed. Cir. 2006); *see also* Iron Grip Barbell Co. v. USA Sports, Inc., 392 F.3d 1317, 1322 (Fed.Cir.2004); In re Geisler, 116 F.3d 1465, 1469 (Fed. Cir. 1997).

mathematics.[137] Bessen and Meurer's observation that without Marsten et al.'s later-developed sparse matrix techniques, "Karmarkar's algorithm by itself was not particularly efficient compared to the linear-programming techniques of the 1940s" takes nothing away from the new and unexpected nature of these achievements, particularly in the context of patent doctrine's minimalist approach to the general utility requirement.[138]

The idea of borrowing interior-point methods from nonlinear programming to compete with advanced exterior-point methods for linear programming was also unexpected, as Margaret Wright writes:

> Prior to 1984, there was, to first order, *no connection* between linear and nonlinear programming. For historical reasons that seem puzzling in retrospect, these topics, one a strict subset of the other, evolved along two essentially disjoint paths. Even more remarkably, this separation was a fully accepted part of the culture of optimization — indeed, it was viewed by some as inherent and unavoidable.[139]

Wright concludes that Karmarkar's algorithm catalyzed an "interior point revolution," uniting the two branches of mathematical programming in an unexpected way.[140]

Of course, Bessen and Meurer's validity concerns must be directed to Karmarkar's individual patent claims, each of which is subject to separate novelty and nonobviousness determinations according to its scope. For this reason, we turn now to address Bessen and Meurer's concerns regarding the scope of Karmarkar's claims.

[137] *See, e.g.*, Chin, *supra* note * (describing presentation of Karmarkar's algorithm to a "packed audience of MAA [Mathematical Association of America] members" at the 1985 Joint Mathematics Meetings).

[138] To be eligible for a patent, a claimed invention need not supersede or work better than the prior art. *See* Lowell v. Lewis, 15 F. Cas. 1018, 1019 (C.C.D. Mass. 1817). (rejecting argument that a claimed pump lacks general utility unless it is "for the public, a better pump than the common pump").

[139] Margaret H. Wright, *The Interior-Point Revolution in Optimization: History, Recent Developments, and Lasting Consequences*, 42 BULL. AM. MATH. SOC'Y 39, 40 (2004) (emphasis in original). As Saltzman has noted, *supra* note 119, Fiacco and McCormick's book does briefly discuss the application of interior-point methods to linear programming. *See* FIACCO & MCCORMICK, *supra* note 81, at 111-12 & 180-83. The book's emphasis, however, is on examining special cases of the more general techniques presented (in which linearity and/or convexity serve as simplifying assumptions), rather than on presenting methods that are efficient in comparison with other linear programming algorithms.

[140] *See* Wright, *supra* note 139, at 39-40; *see also* Mark A. Paley, *The Karmarkar Patent: Why Congress Should "Open the Door" to Algorithms as Patentable Subject Matter*, 22 COMPUTER L. REPORTER 7 (1995) (describing Karmarkar's algorithm as "a revolutionary problem solving method").

4. The Scope of Karmarkar's Patent Claims

Bessen and Meurer express concern about "the difficulty of determining the boundaries of [Karmarkar's] patent," specifically the possibility that Karmarkar's patent claims might read on "the techniques used in the 1960s."[141] Any such claim would be of questionable novelty in light of the prior art, and might unjustly enrich AT&T by enabling it "to assert its patent successfully against people who used linear-programming techniques based on those used in the 1960s."[142] Bessen and Meurer do not identify any particular claim language as giving rise to these concerns, but instead appeal to what they view as software's inherent and distinctive resistance to linguistic line-drawing:

> The abstractness of the patented algorithm means that these determinations cannot be made with certainty. Patent law assumes that two technologies can be unambiguously determined to be equivalent or distinct; this sets the patent boundaries. Yet for software, this assumption simply does not hold. Although this assumption works for most other technologies, it distinctly does not — or does so insufficiently well — for software algorithms. And if computer scientists cannot make these determinations with any certainty, how can we expect judges and juries to do so?[143]

Setting aside the fact that disputes over ambiguous claim scope arise in every technological field, this is a circular argument. Ultimately, the full extent of the Karmarkar example's support for Bessen and Meurer's argument that "software patents are different" turns on this one-paragraph blanket assertion that software "distinctly does not" satisfy the linguistic assumptions that work "for most other technologies."

A complete construction of all of Karmarkar's patent claims is far beyond the scope of this Article. It is relatively straightforward, however, to address Bessen and Meurer's concerns about overbreadth here.

As shown in Figure 5, Karmarkar's patent has 36 claims, of which 22 are independent and 14 are dependent. Nine of the claims (19, 24, 25, 28-31, 33, and 34), including three independent claims, expressly recite mathematical terms that refer specifically to Karmarkar's particular design choices within the class of Frisch/Fiacco-McCormick methods as described

[141] *See* BESSEN & MEURER, *supra* note 6, at 203.
[142] *See id.* This is not a real-world concern, since Karmarkar's patent expired in 2005.
[143] *See* BESSEN & MEURER, *supra* note 6, at 203.

in the patent specification.[144]

Each of the remaining independent claims recites the word "means" or "step" in connection with at least one functional aspect of Karmarkar's projective transformation (indicated by the terms quoted in Figure 5) without any "structure, material, or acts" to implement that function. Accordingly, § 112, ¶ 6 provides that these means-plus-function and step-plus function claims be limited in scope to algorithms that implement a projective transformation as described in the specification.[145]

[144] *See* U.S. Patent 4,744,028, at cols. 7-8 (describing the mathematical steps needed to perform the projective transformation prior to the minimization step during each iteration).

[145] *See* Aristocrat Technologies Australia Pty. Ltd. v. International Game Technology, 521 F.3d 1328, 1333-38 (Fed. Cir. 2008); Harris Corp. v. Ericsson Inc., 417 F.3d 1241, 1253 (Fed. Cir. 2005); WMS Gaming Inc. v. International Game Technology, 184 F.3d 1339, 1348-49 (Fed. Cir. 1999). The statute provides:

> An element in a claim for a combination may be expressed as a means or step for performing a specified function without the recital of structure, material, or acts in support thereof, and such claim shall be construed to cover the corresponding structure, material, or acts described in the specification and equivalents thereof.

35 U.S.C. § 112, ¶ 6.

claim #	dependent claim(s) #	means-plus-function element(s)	step-plus function element(s)	function implemented by projective transformation	function implemented by potential function	express limitation to projective transformation
1			●	"normalizing"		
2		●		"selecting"		
3	4,5,6,7		●	"centralizing"		
8	9,10,11,12	●		"selecting"		
13		●		"selecting"		
14		●		"normalizing"		
15			●	"normalizing"		
16			●	"normalizing"		
17		●		"stepping"		
18			●	"normalizing"		
19						"vector c_p"
20			●	"transforming"	"substantially coincident"	
21			●	"transforming"	"substantially corresponds"	
22		●		"identifying"		
23			●	"transforming"		
24						"projective transformation x_i'"
25						"pointer vector c_p"
26			●	"centralizing"		
27	28,29,30		●	"transforming"	"satisfactory minimization"	
28						"matrix B"
29						"orthogonal projection"
30	31					"new transformed initial vector"
31						"radius"
32	33		●	"transforming"	"satisfactory minimization"	
33	34					"matrix B"
34						"orthogonal projection"
35			●	"rescaled"		
36			●	"rescaling"		

Figure 5. Each one of the 36 claims in Karmarkar's patent appears to have at least one express limitation or § 112, ¶ 6 functional element that narrows its scope sufficiently to address Bessen and Meurer's concerns.

It therefore appears that all 36 claims are limited in scope to the disclosed implementation of Karmarkar's projective transformation, and at least eight of the claims are further limited in scope to the disclosed

implementation of Karmarkar's potential function. Far from exploiting the ambiguity of language to attain overbroad claim scope, Karmarkar's software patent claims are cabined by express recitals and by § 112, ¶ 6 into the very design choices that accurately represent his contributions relative to the prior art.

C. Discussion

Apart from the failure of Bessen and Meurer's illustrations to support their claims about the unique linguistic unwieldiness of software-related inventions, the claims themselves seem deeply counterintuitive. Perhaps more than any other technological fields, the disciplines of computer science and software engineering must rely on mathematically precise specifications of the designs and behaviors of their creations. For this reason, the pervasiveness of abstraction in software technology per se does not doom the field to ambiguous line-drawing. Computer scientists are well aware that their work involves abstraction; the best computer scientists are able to express that abstraction with precision and rigor.[146] The real question for software patent doctrine is not how to drive abstraction out of the patent system, but how the law can affirm and harness cognitive abstraction skills to promote innovation, rather than allow their abuse to evade otherwise generally applicable requirements for patentability.

II. KLEMENS

A. Klemens's Proposal

Software-related inventions have historically created difficulties for the courts in attempting to draw the line between patentable and unpatentable subject matter. The long march from *Benson*[147] and *Diehr*[148] to *Alappat*,[149]

[146] *See generally* Jeff Kramer, *Is Abstraction the Key to Computing?*, 50 COMMUNICATIONS OF THE ACM 37 (2007) (discussing the importance of abstraction skills in the computer science profession).

[147] Gottschalk v. Benson, 409 U.S. 63 (1972) (holding unpatentable claims to a method for converting binary coded decimal number representations into binary number representations).

[148] Diamond v. Diehr, 450 U.S. 175 (1981) (holding patentable a claimed method of operating a rubber-molding press reciting steps of a mathematical algorithm for calculating the cure time based on the Arrhenius equation).

[149] In re Alappat, 33 F.3d 1526 (Fed. Cir. 1994) (holding a general-purpose machine programmed to perform a series of computational steps patentable as a "new machine").

State Street Bank,[150] and *Bilski*[151] has been long and sinuous, and may not be finished.[152]

Klemens argues that the line drawn by the Court of Customs and Patent Appeals in the *Freeman-Walter-Abele* line of cases[153] and repudiated by the Federal Circuit in *State Street Bank*[154] should be restored. Klemens favors the test because it effectively distinguishes between "bona fide physical inventions" and "information processing algorithms with a trivial physical step" such as operation of a standard I/O device[155] and takes seriously the Supreme Court's dictum in *Diehr* that "insignificant postsolution activity will not transform an unpatentable principle into a patentable process."[156] Specifically, Klemens's proposal is to exclude from § 101 patentable subject matter all combination claims of the following form:

Patent N

> 1. A useful computing machine, comprising
>
> (a) a mathematical algorithm, which may be creatively and painstakingly derived, but which is clearly unpatentable by the mathematical algorithm exception, and
>
> (b) an obvious physical step such as loading the algorithm onto a stock computer, which meets the requirements for patentable subject matter but is unpatentable because it is not novel.[157]

[150] State Street Bank & Trust Company v. Signature Financial Group, Inc., 149 F.3d 1368 (Fed. Cir. 1998) (holding the transformation of financial data through a series of mathematical calculations patentable as producing "a useful, concrete and tangible result").

[151] In re Bilski, 545 F.3d 943 (Fed. Cir. 2008) (en banc) (holding unpatentable a claimed process for managing financial risks as neither tied to a particular machine nor resulting in a physical transformation).

[152] *See id*. at 994-95 (Fed. Cir. 2008) (en banc) (Newman, J., dissenting) (noting that the majority decision leaves open the questions of whether "*Alappat*'s guidance that software converts a general purpose computer into a special purpose machine remains applicable" and whether the inventions in *State Street Bank* and *AT&T v. Excel* are patentable subject matter).

[153] In re Freeman, 573 F.2d 1237, 1245 (C.C.P.A. 1978); In re Walter, 618 F.2d 758, 767 (C.C.P.A. 1980); In re Abele, 684 F.2d 902 (C.C.P.A. 1982).

[154] See State Street Bank, 149 F.3d at 1374 ("[T]he Freeman-Walter-Abele test has little, if any, applicability to determining the presence of statutory subject matter").

[155] *See* Klemens, *supra* note 4, at 2-3 (describing test); *id.* at 35 (restating the paper's recommendation as a "regression" to the practice of "respecting the caveats about postsolution activity in the Freeman-Walter-Abele test").

[156] 450 U.S. at 191-92; *see* Klemens, *supra* note 4, at 36 (explaining importance of "respecting the declaration" in *Diehr*).

[157] This appears to be a refinement of Klemens's earlier proposal that for a programmed general-purpose computer to be patentable, "a machine would have to be built that may rely on mathematics but does something innovative beyond it. . . . If the entire

Klemens contends that "the great majority of software patent applications are clearly of the form of Patent N: an algorithm loaded onto a stock computing device."[158]

The "machine-or-transformation" test articulated in the Federal Circuit's recent en banc decision in *In re Bilski*[159] calls for critical inquiries that nominally address Klemens's concerns; i.e., whether the claimed process "is tied to a particular machine or apparatus"[160] or "transforms a particular article into a different state or thing,"[161] (as opposed to the entire universe of digital computers[162] or insignificant post-solution or extra-solution activity[164]). The decision is unlikely to satisfy Klemens, however, as it applies only to process claims,[165] rejects the *Freeman-Walter-Abele* approach,[166] and (as Klemens himself notes[167]) leaves open the question of whether the act of loading an algorithm onto a stock computer produces a "particular machine."[168] The *Bilski* court also took pains to state as settled doctrine that the patentable subject matter inquiry is to be directed to the claim as a whole[169] and is to be completely independent of any novelty or

Deleted:)

Deleted: or "transforms a particular article into a different state or thing"[163] (as opposed to

design [of the machine] consists of an equation, then there is nothing to be patented; if the design consists of an equation and a trivial machine, then there is still nothing to be patented; if the design is for a new and novel machine informed by mathematics, then there is every reason to grant a patent on the machine's design." *See* KLEMENS, *supra* note 2, at 64. Even as such, Klemens's conflation of "obvious" with "not novel" in paragraph (b) suggests that further refinement is necessary. *See* 35 U.S.C. § 103 (stating that a claimed invention may be novel yet obvious).

In his book, Klemens also proposes that "an inventive physical implementation of a state machine (such as an FPGA [field-programmable gate array], a JVM [Java Virtual Machine] on a chip, or a rubber-curing device) should be patentable, whereas the programs loaded onto them (firmware, a data structure) should not." *See id.* at 64-65. Klemens's reading of the Church-Turing thesis does not impinge on the merits of this proposal, and this Article will not opine on them.

[158] *See* Klemens, *supra* note 4 at 36.

[159] 545 F.3d 943 (Fed. Cir. 2008) (en banc).

[160] Id. at 954.

[161] Id. at 954.

[162] See id. at 953-54 (contrasting *Benson* with *Diehr*).

[164] See id. at 957 & n.14.

[165] Id. at 951.

[166] See id. at 958-59.

[167] *See* Ben Klemens, *In regards to* In re Bilski, available at http://ben.klemens.org/blog/arch/00000009.htm (visited Nov. 20, 2008) (stating Klemens's view, in a blog entry one day after the decision, that "the ruling does make progress" but "won't answer the key, central question").

[168] 545 F.3d at 995 (Newman, J., dissenting) ("We aren't told when, or if, software instructions implemented on a general purpose computer are deemed 'tied' to a 'particular machine.').

[169] See id. at 958 (citations omitted) ("[T]he Court has made clear that it is inappropriate to determine the patent-eligibility of a claim as a whole based on whether

nonobviousness considerations,[170] thereby making it clear that Klemens's approach to the validity of machine claims has no place in current § 101 jurisprudence.

Like Bessen and Meurer, Klemens supports his proposal for legal change in large part with empirical research on the economic costs of the status quo to both the patent system[171] and the software industry.[172] In the context of a policy argument directed to Congress, this research might prove to be highly useful and persuasive. The other part of Klemens's case, however, is based on an imprecise and superficial reading of the theoretical computer science literature. Klemens repeatedly argues that a widely adopted working hypothesis in computer science, known as the *Church-Turing thesis*, compels a doctrinal change in the application of the § 101 patentable subject matter requirement to software generally and Patent N specifically. It does not, and any courts to whom Klemens addresses this argument[173] should be informed accordingly.

B. The Church-Turing Thesis

The Church-Turing thesis is the outgrowth of contemporaneous efforts by computer science pioneers Alonzo Church and Alan Turing to define the class of mathematical problems that were amenable to solution by computer.[174] Turing's theory developed around the Turing machine model,[175] while Church's work focused on a notation for expressing algorithms as functions known as the lambda calculus.[176] The Turing machine is described in detail elsewhere in this Article;[177] what now follows

selected limitations constitute patent-eligible subject matter. . . . Thus, it is irrelevant that any individual step or limitation of such processes by itself would be unpatentable under § 101.").

[170] See id. (citations omitted) ("[T]he Court has held that whether a claimed process is novel or non-obvious is irrelevant to the § 101 analysis. Rather, such considerations are governed by 35 U.S.C. § 102 (novelty) and § 103 (non-obviousness).").

[171] *See* KLEMENS, *supra* note 2, at 84, 90-91 & 107; Klemens, supra note 4, at 27-32.

[172] *See* KLEMENS, *supra* note 2, at 92-107; Klemens, *supra* note 4, at 21-27.

[173] *See* End Software Patents Project, *End Software Patents: Resources for Lawyers* <http://endsoftpatents.org/resources-for-lawyers> (visited July 15, 2008) (describing efforts by Klemens's End Software Patents Project to engage the legal community).

[174] *See* MARTIN DAVIS, THE UNIVERSAL COMPUTER: THE ROAD FROM LEIBNIZ TO TURING 163-67 (2000) (providing a historical account of Turing's and Church's independent work on David Hilbert's famous *Entscheidungsproblem*).

[175] *See* Alan M. Turing, *On Computable Numbers with an Application to the Entscheidungsproblem*, 2 PROC. LONDON MATH. SOC. 230 (1936).

[176] *See* ALONZO CHURCH, THE CALCULI OF LAMBDA-CONVERSION (1941).

[177] A caveat: The Turing machine model described earlier, *see supra* text

is a very brief introduction to a few of the concepts behind Church's lambda calculus.[178]

One reason for using the lambda calculus is the latent ambiguity that may exist even in a simple mathematical expression like $x - y$.[179] Is this a function of x or of y (or both, or neither)? We could clarify the situation by writing $f(x) = x - y$, but this forces another symbol, f, into the discussion. This might seem a small complication, but it might be difficult to keep track of such details over the course of a long computation.

Church's solution is to use the special symbol λ to distinguish between two kinds of variables that may appear in a mathematical expression. In Church's lambda calculus, the notation $(\lambda x.x - y)$ indicates that the expression $x - y$ is a function of x.[180] A variable such as x that is preceded by λ is known as a "bound variable"; a variable such as y that is not preceded by λ is known as a "free variable."[181]

The notation $(\lambda x.x - y)$ is treated as a function that can be evaluated for specified values of the bound variable x by substitution; e.g., $(\lambda x.x - y)(1) = 1 - y$.[182]

It is sometimes useful to make the act of substitution more explicit. The lambda calculus provides a "bracket-slash" notation to do this. Thus, the foregoing evaluation can also be written $(\lambda x.x - y)(1) = [1/x](x - y) = 1 - y$. The notation $[1/x]$ indicates that in the immediately following expression (i.e., $x - y$), each occurrence of x is to be replaced by 1.[183]

accompanying notes 34-37, is limited to evaluating Boolean-valued ("yes" or "no") functions. It is straightforward (but uninteresting for present purposes) to extend the model to evaluate more general functions, *see* JOHN E. HOPCROFT & JEFFREY D. ULLMAN, INTRODUCTION TO AUTOMATA, LANGUAGES, AND COMPUTATION 151 (1979); *see also infra* Appendix (presenting an example of a Turing machine that outputs a string of plus-signs), and it is this unrestricted model that is the subject of the discussion in the sequel.

[178] There are actually several varieties of "lambda calculi," including "typed lambda calculi" in which terms may be given one of a number of "type" designations, each of which is subject to certain specified syntactic restrictions. See J. ROGER HINDLEY & JONATHAN P. SELDIN, LAMBDA-CALCULUS AND COMBINATORS: AN INTRODUCTION 1 (2008) (discussing varieties of lambda calculus); id. at 107-219 (surveying various typed varieties). As the discussion in this Article and in Klemens's writings concerns only the untyped lambda calculus, this Article hereinafter adopts Klemens's practice of referring to the untyped lambda calculus as simply "the lambda calculus."

[179] *See* HINDLEY & SELDIN, *supra* note 178, at 1.

[180] See id. at 1-2.

[181] See id. at 6-7.

[182] See id. at 2.

[183] See id. at 7.

The validity of replacing $(\lambda x.x-y)(1)$ with $[1/x](x-y)$ in the lambda calculus is due to the fact that the lambda calculus includes a number of defined rules for *converting* expressions. This particular conversion rule is known as a β-reduction.[184] β-reductions can be used iteratively to dramatic effect, as the following example illustrates:

$$(\lambda x.(\lambda y.yx)z)v = [v/x]((\lambda y.yx)z) = (\lambda y.yv)z = [z/y](yv) = zv\,.$$[185]

Space precludes a complete presentation of Church's system here, but it should already be apparent that the evaluation and conversion of expressions in the lambda calculus generates a powerful set of computational techniques. In fact, Church's system is known to be as powerful as the Turing machine model, because Turing proved in 1937 that any function that could be computed on a Turing machine could also be evaluated in the lambda calculus, and vice versa.[186]

Over time, Church and Turing's work gave rise to a growing belief among computer scientists that the class of Turing-computable (or lambda-evaluable) functions includes every function that can be computed on any plausible computing device. The assumption that this will continue to be the case, i.e., that the class of Turing-computable functions is the same as the class of all machine-computable functions, has become known as the "Church-Turing thesis" (though sometimes referred to as "Church's hypothesis").[187]

Since no one can claim to have envisioned every computing device that will ever be invented, the notion of a "computable function" has never been formalized. Meanwhile, however, computer scientists have been proving equivalence (or "Turing-completeness") results involving a wide range of programming languages[188] and abstract computational models,[189] giving credence to the Church-Turing thesis and further research that relies upon it as a working hypothesis.[190] As a famous theoretical computer science textbook describes this ongoing research program, "While we cannot hope

[184] See id. at 11-12.

[185] See id. at 12.

[186] *See* Alan M. Turing, *Computability and* λ*-Definability*, 2 J. SYMBOLIC LOGIC 153 (1937).

[187] *See* JOHN E. HOPCROFT & JEFFREY D. ULLMAN, INTRODUCTION TO AUTOMATA, LANGUAGES, AND COMPUTATION 166 (1979).

[188] *See, e.g.*, Robert S. Boyer & J. Strother Moore, *A Mechanical Proof of the Turing Completeness of Pure Lisp*, in AUTOMATED THEOREM PROVING: AFTER 25 YEARS, at 133 (W.W. Bledsoe & D.W. Loveland eds. 1984)

[189] *See, e.g.,* HOPCROFT & ULLMAN, *supra* note 187, at 167-74 (presenting equivalence results for various abstract computational models).

[190] *See, e.g.*, Arthur Charlesworth, Infinite Loops in Computer Programs, 52 Math. Mag. 284, 287-88 (1979) (providing a new proof of one of Turing's theorems, subject to the assumption that the Church-Turing thesis is true).

to 'prove' Church's hypothesis as long as the informal notion of 'computable' remains an informal notion, we can give evidence for its reasonableness."[191]

C. Klemens's Reading(s) of the Church-Turing Thesis

Apart from referring to an unproven hypothesis as a "theorem," Klemens's description in *Math You Can't Use* of the Church-Turing thesis as "[t]he theorem central to this book"[192] is more than apt. To Klemens, the Church-Turing thesis is a panacea for the courts' ill-conceived doctrines on the patentability of software. In both his book and his article, he cites it in support of a dizzying variety of propositions:

1. Anything a computer could possibly do can be done by a Turing machine. Klemens introduces the Church-Turing thesis in the following passage:

> **Theorem 1:** The Church-Turing Thesis
> All computable operations can be evaluated by a Turing machine.
>
> The exact meaning of computable is a technical matter that I will not delve into here; roughly, it means "anything a computer could possibly do." The Church-Turing thesis states that any computer program, written in any language, can be rewritten as a Turing machine.[193]

2. The Church-Turing thesis "indicates that . . . there is a mechanical means of translating any mathematical expression into a computable program, and a means of translating any computable program into a mathematical expression."[194]

3. Software is indistinguishable from pure mathematics. In his book, Klemens reasons that "[s]ince any program in any Turing complete programming language is identical to a system of equations in the lambda calculus, the courts will be unable to draw" the line between pure mathematics and software.[195] In his article, Klemens simply states that the Church-Turing thesis directly implies that "all software is mathematics."[196]

[191] *See* HOPCROFT & ULLMAN, *supra* note 187, at 166.

[192] KLEMENS, *supra* note 1, at 47. Klemens introduces the Church-Turing thesis in his subsequent article no less inaccurately as "a basic result of computer science." Klemens, *supra* note 4, at 9.

[193] KLEMENS, *supra* note 1, at 35.

[194] Klemens, *supra* note 4, at 9-10.

[195] KLEMENS, *supra* note 1, at 35-36.

[196] Klemens, *supra* note 4, at 10.

4. Every application of an algorithm is indistinguishable from pure mathematics; therefore, claim 1 of Patent N should be held invalid. David Gale and Lloyd Shapley conclude their 1962 American Mathematical Monthly article[197] announcing their algorithm for solving the "stable marriage problem" with some reflections from the perspective of economists working on a problem of more general interest to mathematicians. They write: "In making the special assumptions needed in order to analyze our problem mathematically, we necessarily moved further away from the original college admission question, and eventually in discussing the marriage problem, we abandoned reality altogether and entered the world of mathematical make-believe."[198]

Klemens first quotes and later paraphrases this comment as follows: "As Gale and Shapley explained, there is no difference between an application of an algorithm and the algorithm itself."[199] He then reminds the reader that "as the Church-Turing thesis states, the algorithm and pure math are entirely equivalent."[200] Klemens makes these points to imply that examiners erroneously granted several patents that were directed to "a general-purpose computer with a program loaded."[201]

5. Owning a software patent is the same as "own[ing] a piece of mathematics." Klemens provides no explanation for this conclusion, but it appears to follow from propositions 3 and 4.

6. If software had been patentable in the 1930s, the Church-Turing thesis might not have been developed. Noting the contemporaneous development of the lambda calculus by Church and the Turing machine by Turing, Klemens reasons that "any such hyphenated theorem [sic] would be a lawsuit in the making."[202]

7. *"It is impossible to write a section of the Manual of Patent Examination Procedure (MPEP) that allows the patenting of software but excludes from patentability the evaluation of purely mathematical algorithms."*[203] Klemens states that "the proof" of this proposition is to be found in "the formal Church-Turing thesis" and Donald Knuth's comment that "All data are numbers, and all numbers are data."[204]

[197] David Gale & Lloyd S. Shapley, *College Admissions and the Stability of Marriage*, 69 AM. MATH. MONTHLY 9 (1962).

[198] *See id.* at 14 (quoted in KLEMENS, *supra* note 1, at 48-49).

[199] *See* KLEMENS, *supra* note 1, at 63.

[200] *See id.*

[201] *See id.*

[202] Id. at 47.

[203] Klemens, *supra* note 4, at 10.

[204] Id. at 9-10 (citing Letter from Donald Knuth, Professor Emeritus, to Commissioner of Patents and Trademarks, Patent and Trademark Office, available via the Internet Archive <http://www.archive.org> at <http://lpf.ai.mit.edu/Patents/knuth-to-pto.txt> (visited August

D. Discussion

Read in context, Klemens's repeated mischaracterizations of the Church-Turing thesis as a proven theorem are not really that problematic. Like computer scientists, the law can draw conclusions from unrebutted presumptions, and it would be highly prudent to do so on the massive body of evidence that now exists. An alternative interpretation, also in Klemens's favor, is that in citing the Church-Turing thesis he might actually be referring instead to the body of evidence that supports the thesis; i.e., proven Turing-completeness results for numerous languages and machine models. This, however, is the least serious of Klemens's errors.

More serious is Klemens's overstatement of the Church-Turing thesis. As explained above, the Church-Turing thesis arises out of Turing's proof of an equivalence between Church's lambda calculus and the Turing machine. The precise nature of this equivalence is crucial. Specifically, Turing showed that any function that could be computed on a Turing machine could also be evaluated in the lambda calculus, and vice versa. The Church-Turing thesis claims that this particular equivalence — between the classes of functions that can be computed using the respective models — can be extended even to the most powerful plausible models of computation.[205]

In an article titled "The Church-Turing Thesis: Breaking the Myth,"[206] computer scientists Dina Goldin and Peter Wegner address precisely the same commonly held[207] misunderstanding that informs much of Klemens's commentary. Goldin and Wegner state the Church-Turing thesis as follows: "Whenever there is an effective method (algorithm) for obtaining the values of a mathematical function, the function can be computed by a TM [Turing machine]."[208] They go on, however, to report that the thesis "has since been

15, 2008).

[205] See supra text accompanying note 187.

[206] Dina Goldin & Peter Wegner, *The Church-Turing Thesis: Breaking the Myth*, in NEW COMPUTATIONAL PARADIGMS 152 (Springer-Verlag Lecture Notes in Computer Science, vol. 3526, 2005).

[207] *See id.* at 154 (opining that the myth "is dogmatically accepted by most computer scientists). Goldin and Wegner state that at least one popular undergraduate textbook contains the erroneous reinterpretation. *See id.* (citing MICHAEL SIPSER, INTRODUCTION TO THE THEORY OF COMPUTATION (1997)). The allegedly offending textbook does not actually offer a formal statement of the Church-Turing thesis, however, but says that the term refers to the "connection between the informal notion of algorithm and the precise definition" supplied by the lambda calculus and Turing machine models. SIPSER, *supra*, at 143.

[208] Goldin & Wegner, supra note at 153.

reinterpreted to imply that Turing Machines model *all* computation, rather than just functions," to the effect that "[a] TM can do (compute) anything that a computer can do."[209] They respond that "[i]t is a myth that the original Church-Turing thesis is equivalent to this interpretation of it; Turing himself would have denied it."[210]

Goldin and Wegner's insights rebut the first four of Klemens's propositions. With respect to the first, the Church-Turing thesis does not imply that a Turing machine can emulate "anything a computer could possibly do." As Goldin and Wegner point out and every reasonably sophisticated computer user should be able to recognize, modern computers do much more than evaluate functions; they also interact with their users and with their environments.[211]

Regarding Klemens's second proposition, a proof that a particular computational model or programming language is Turing-complete requires only a showing that it can compute all Turing-computable functions; it does not necessarily entail the construction of a "mechanical means of translating" algorithms from one model to the other. Thus, the Church-Turing thesis itself, and the Turing-completeness results that make up the body of evidence supporting it, have nothing to say about the skill and effort needed to write software in a given language for a given machine or the computational resources (time, space, bandwidth, etc.) needed to run the software.

The blindness of Turing-completeness proofs to computational resource constraints highlights a key feature of the Turing machine and lambda calculus models of calculation: they are endowed with infinite computational resources, unlike every real-world computer. Software developed for the real world must contend with scarce resources, and a solution to a computational problem that conserves these resources (e.g., Karmarkar's algorithm) can exhibit nonobvious differences over prior art solutions to the same problem,[212] as well as substantial differences in function, way and result that might support a reverse doctrine of equivalents defense.[213] These legally cognizable differences between abstract

[209] Id. at 153-54.

[210] Id. at 154.

[211] *See id.* at 156 (giving example of a robotic car); Peter Wegner & Dina Goldin, *Computation Beyond Turing Machines*, 46 COMMUNICATIONS OF THE ACM 100, 101 (2003) ("The field of computing has greatly expanded since the 1960s, and it has been increasingly recognized that artificial intelligence, graphics, and the Internet could not be expressed by Turing machines. In each case, interaction between the program and the world (environment) that takes place during computation plays a key role that cannot be replaced by any set of inputs determined prior to the computation.").

[212] *See supra* text accompanying notes 122-140.

[213] *See* Andrew Chin, *Computational Complexity and the Scope of Software Patents*,

computational models and real-world computers present a further challenge to Klemens's essentially rhetorical efforts to extend Turing's narrowly defined, formal notions of equivalence into the realm of patent doctrine.

Klemens's third and fourth propositions appeal specifically to the mathematical form of the functions that can be expressed in Church's lambda calculus. As explained above, however, the proofs of equivalence between the lambda calculus and other Turing-complete models of calculation stop well short of constructing algorithms that are "identical" or "entirely equivalent." Klemens's fourth proposition also relies on a dubious interpretation of Gale and Shapley's remarks.

Klemens's fifth and sixth propositions are gross misstatement of patent law. The Patent Act confers rights to exclude, not ownership rights to mathematics or anything else,[214] and precludes Church, Turing or anyone else from obtaining (and, a fortiori, asserting in a "lawsuit in the making") any patent rights that could cover a scientific hypothesis such as the Church-Turing thesis[215] — particularly one so admittedly indefinite with respect to the notion of "computable functions."[216]

Finally, the original articles formulating the Church-Turing thesis are all open to public examination, and one will search them in vain for a proof of Klemens's seventh proposition — Donald Knuth's quip notwithstanding.

III. CONCLUSIONS

As surveys of the empirical patent law literature, Bessen and Meurer's and Klemens's books both identify a host of symptoms — overwhelmed examiners, high litigation costs, and structural distortions of software-related industries — that strongly indicate an economic misalignment between the patent system and the pursuit of software innovation. Their diagnoses of the problem, however, suffer from factual errors and misinterpretations of computer science concepts. Particularly problematic are their various treatments of abstraction and equivalence in computer science, which do not map directly or intuitively to notions of abstraction

39 JURIMETRICS 17 (1999). In the context of field-programmable gate arrays, Klemens himself proposes an approach to infringement that would allow an imitator to take a "broad algorithm" from a patented array provided that its implementation details were different from those that "the designers worked to optimize" with respect to the array's physical resource constraints. *See* KLEMENS, *supra* note 2, at 67.

[214] *See* 35 U.S.C. § 154.

[215] *See* Tol-O-Matic, Inc. v. Proma Product-und Mktg. G.M.b.H., 945 F.2d 1546, 1552 (Fed. Cir. 1991) ("By § 101 there is excluded from the patent system such things as *scientific theories*, pure mathematics, and laws of nature.") (emphasis added).

[216] See 35 U.S.C. § 112, ¶ 1.

and equivalence in legal reasoning and patent doctrine. At least as currently presented, their arguments that software is different, and that this difference compels technology-specific changes in patent doctrine, appear to be without empirical support.

The factual corrections provided in this Article serve as a timely reminder that an empirical approach to patent law reform calls for attention not only to economic methods, but also to the scientific principles and stakeholder perspectives that pervade patent law and practice. Scholars interested in diagnosing the disconnect between the patent system and software innovation should know what computer scientists have said on the subject.

For example, European computer scientists Martin Campbell-Kelly and Patrick Valduriez recently conducted a detailed technical review of the fifty most-cited software patents issued since 1990.[217] They found little evidence that obvious or overbroad patents had been granted.[218] Their main cause for concern was that forty-four of the patents "had medium or low disclosure that would make reproducing the invention either time-consuming or problematic."[219] The scientists' findings support a more modest approach to software patent reform, which would aim to elaborate the enablement and written description requirements in accordance with the standard practices of software engineers for documenting and validating their inventions.[220] They have also conducted a subsequent study in the area of anti-spam software patents.[221]

While both of these studies are of considerable interest to the scientific community,[222] Campbell-Kelly and Valduriez have taken the exceptional and commendable step of publishing their results in American student-edited law reviews, rather than in peer-reviewed scientific journals.

[217] Martin Campbell-Kelly & Patrick Valduriez, *A Technical Critique of Fifty Softwaer Patents*, 9 MARQ. INTELL. PROP. L. REV. 249 (2005).

[218] See id. at 281.

[219] Id.

[220] *See supra* text accompanying note 146; *see also* Jay P. Kesan, *Carrots and Sticks to Create a Better Patent System*, 17 BERKELEY TECH. L.J. 145, 167-69 (2002) (arguing that the Patent Office should require the use of standard modeling and representational languages in software patent disclosures); *but see* Ajeet P. Pai, Note, *The Low Written Description Bar for Software Inventions*, 94 VA. L. REV. 457, 490-93 (2008) (arguing that patent law should continue to maintain a low written description requirement for software inventions).

[221] Martin Campbell-Kelly & Patrick Valduriez, *An Empirical Study of the Patent Prospect Theory: An Evaluation of Anti-Spam Patents*, 11 VA. J. L & TECH. 10 (2006)

[222] *Cf.* Wolfgang Emmerich et al., *The Impact of Research on the Development of Middleware Technology*, ACM TRANSACTIONS ON SOFTWARE ENGINEERING AND METHODOLOGY, vol. 17, no. 4, art. 19 (Aug. 2008) (reporting, *inter alia*, findings regarding the historical importance of patented inventions in the field of middleware technology).

Interestingly, Bessen and Meurer's book discusses at some length an earlier historical article on software patents by Campbell-Kelly,[223] but does not mention any of his empirical studies.[224] Bessen and Meurer may be right to criticize Campbell-Kelly's historical account of the software patent controversy as too narrow, but their equally narrow view of empirical patent law scholarship forecloses an important opportunity to acknowledge the methods and perspectives that computer scientists can contribute to the study of software patenting. Given the significant problems Bessen, Meurer and Klemens have identified, the cause of software patent reform would be better served by a deeper engagement between recognized scholars in the fields of patent law, economics and computer science than has appeared to date.

APPENDIX. A SIMPLE TURING MACHINE

This example of a Turing machine is designed to double the initial number of + symbols on its tape. The Turing machine consists of an infinite strip of tape partitioned into an infinite number of spaces, and a head that can move in either direction along the tape and can print a symbol taken from a finite alphabet into the space where it resides, replacing whatever was in the space before. At any given time, the machine is in one of a finite number of states. The head performs work on the tape through a sequence of moves. During each move, the head may (a) perform a read, write or erase operation, (b) change to any state (or remain in the current state), and (c) move one space either to the left or to the right. The specific move to be taken by the head at any given time is determined by a next move function that depends on (i) the current state of the machine and (ii) the current contents of the space where the head is located.

The table in Figure 6 describes the next move function for this Turing machine. It has five states and uses the alphabet $\{+, \langle \text{blank} \rangle\}$.

Machine State	If head reads a $\langle \text{blank} \rangle$	If head reads a +
State 1	STOP	Write <blank>; change to state 2; move left
State 2	Write +; change to state 3; move left	Remain in state 2; move left
State 3	Write +; change to state	Remain in state 3; move left

[223] Martin Campbell-Kelly, *Not All Bad: An Historical Perspective on Software Patents*, 11 MICH. TELECOMM. & TECH. L. REV. 191 (2005).

[224] *See* BESSEN & MEURER, *supra* note 6, at 188-91.

	4; move right	
State 4	Change to state 5; move right	Remain in state 4; move right
State 5	STOP	Write <blank>; change to state 2; move left

Figure 6. Next move function for a Turing machine that doubles the initial number of + symbols on the tape.

As indicated in Figure 7, the initial content of the tape, or *input*, consists of a single contiguous string of + symbols on an otherwise blank tape. Initially (at time t=0), the head is initially in state 1 and is located at the leftmost + symbol. Given this initial condition and the next move function defined in Figure 6, it is possible to determine the sequence of all subsequent moves. Figure 7 shows how this Turing machine continues for 14 steps and then stops in state 5.

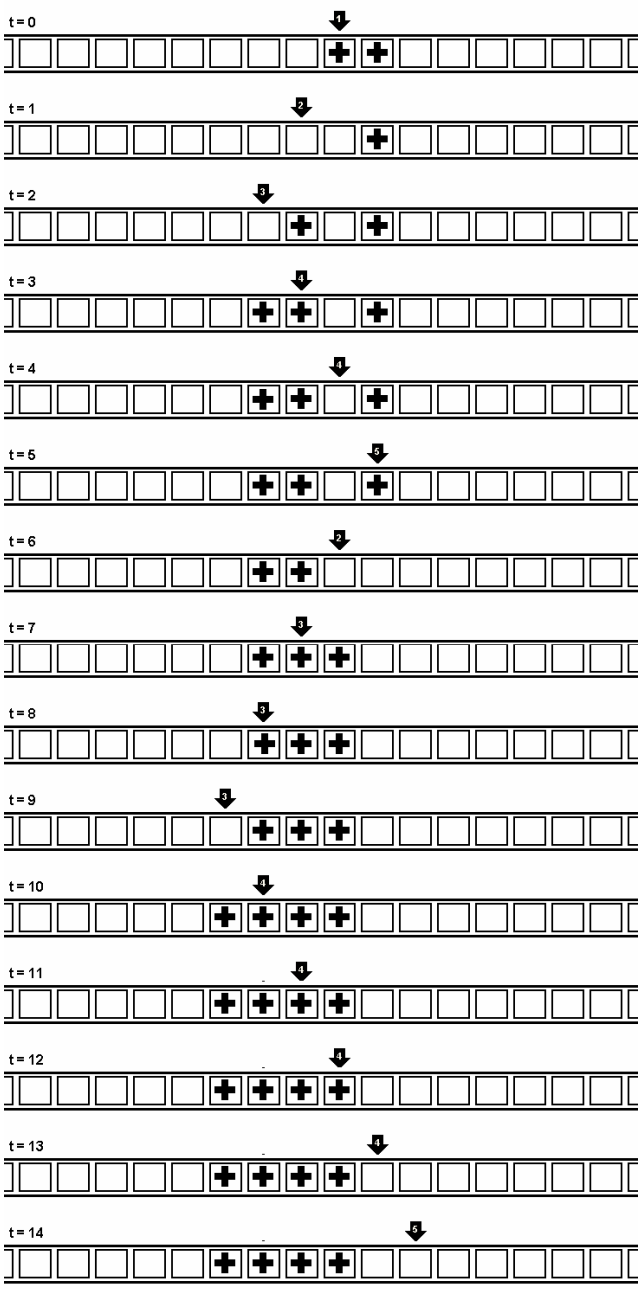

Figure 7. First 14 steps of a computation on a Turing machine with the next move function defined in Figure 6.