scripted

Volume 8, Issue 2, August 2011

ARGUMENTS IN CONSIDERING THE SIMILARITY OF ALGORITHMS IN PATENTING

Kenneth Oksanen, Perttu Virtanen, Eljas Soisalon-Soininen, Jukka Kemppinen[*]

Abstract

The determination as to whether or not two algorithms in a computer programme are similar enough to be considered 'the same' algorithm can be crucial in patent prosecution and other legal disputes, from theft of trade secrets to patent infringement. Establishment of prior art by the responsible court involves consideration of the contemporary practices of software engineers and computer scientists who develop and implement algorithms. This paper, co-authored by two computer scientists and two legal professionals, reviews those arguments that can be used to assess the similarity of algorithms, in relation to the criteria of novelty for a grant of patent.

DOI: 10.2966/scrip.080211.138

[*]Kenneth Oksanen, Lic.Tech, project manager, Aalto University; Perttu Virtanen, Ph.D, LLM, post-doctoral researcher, HIIT, Aalto University; Eljas Soisalon-Soininen, Ph.D, professor, Aalto University; Jukka Kemppinen, Ph.D, professor emeritus, multiple docent, Aalto and Turku University.

1. Introduction

Whether two algorithms - described abstractly in a document or employed concretely in, for example, a computer programme - are 'the same' or different from one another, may be a crucial determination in certain legal disputes. Such disputes will range from trade secret thefts or copyright violations to the determination of the scope of a patent,[1] which is the legal focus of the current paper.[2] The court responsible for such a decision must find a balance between protection of the intellectual property rights of the claimant on the one hand, and avoidance of a false judgment against the innocent respondent on the other. It must also consider the contemporary sophisticated practices of software engineers and computer scientists who develop and implement algorithms as part of the establishment of *prior art.*[3]

The term "*algorithm*" has several historical uses, including the Arabic system of numeration and the art of calculation. In current speech it means more generally any sequence of simple actions entailed in the performance of some more complex task. A recipe for baking a cake, which involves a sequence of steps (adding ingredients, mixing, and heating), for example, constitutes an algorithm. In computing, an algorithm more specifically defines *a set of precise rules for performing a recursive computation for solving a problem in a finite number of steps*. The rules may contain mathematical or logical operations, repetition, procession to another rule, or temporary performance of another set of rules; they allow access to the internal state of the algorithm, taking *input* and producing *output*.

Input and output to the algorithm must be considered very loosely. In a word processor, for example, input is via the keyboard and mouse, and output is the display. Instructions to a word processor to read a file or save a file result in read and saved data that can be considered as additional inputs and outputs, respectively. The operating system that provides the file access primitives to the word processor views the requests of the word processor in reversed roles. In fact, association of the output of one algorithm with the input of another is the standard method of composing algorithms into larger algorithms and eventually into programmes, regardless of whether the association is accomplished through ordinary function calls, shared memory, a middleware of some sort, or a network connection and explicit encoding to a data exchange protocol.

---

[1] On various forms of IP protection available for software, see eg G Mowery: "Intellectual Property Protection in the U.S. Software Industry" in: W Cohen and S Merill (eds), *Patents in Knowledge-Based Economy* (Washington, D.C.: National Academy Press, 2003) at 7-5.

[2] Algorithms, by themselves, are not usually regarded as patentable "as such" but only when reduced to practical applications in the form of a computer programme. In the United States, a claim consisting solely of simple manipulations of abstract concepts, numbers, or signals do not constitute patentable "processes" as mentioned in *Gottschalk v Benson*, 409 U.S. 63, 71-72, 175 USPQ 673, 676 (1972). This lore is, however, to some extent inaccurate, as discussed in Chapters 5 and 6. See also: M. Lemley *et a:* Life after *Bilski* [2011] 63 *Stanford Law Review* 1315-1347.

[3] Prior art or 'state of the art' in patent law, in the main constitutes all information that has been made available to the public in any form before a given date that may be relevant to a claim of novelty in the first instance, and subsequently to the existence of an inventive step.

Rules in algorithms may refer to variables and the assignment of new values to them. Algorithms may employ various *data structures* in order to manage larger amounts of data than can be expressed in a fixed number of variables. The oldest known data structure is the *array*. Whereas mathematicians store an unbound number N values in a variable indexed through subscripting x_1, x_2, ..., x_N, a software engineer writing C or Java would use an array x[0], x[1], …, x[N-1]. Arrays are all we need to deliver the message in this presentation, but typical programming languages may contain such data structuring mechanisms as records, objects (as in object-oriented programming) and in some cases linked lists or associative arrays indexed by arbitrary values instead of mere integers. Programming languages are frequently accompanied by standard *libraries* which may provide tens of additional data structures and the Internet and scientific literature can be consulted for thousands more.

In the context of the consideration of similarity of computer algorithms, a matter that has perhaps obfuscated the patentability discourse within the legal spheres deserves attention. In the context of patenting, the patentable subject-matter is habitually referred to as the computer programme or, more generically as software.

Although there is a current tendency to move away from generic patent claims[4] that exacerbate the problem of insufficient disclosure acknowledged in the USA,[5] there remains a wealth of granted patents that disclose in broad terms the functions of a piece of software, but divulge only generically the means to this end. They provide little or no guidance on *how* the software in question could be written or should be implemented. Flowcharts and detailed descriptions of the patented programmes, let alone source code or object code, are often absent.[6]

The protectable subject-matter may not therefore be the computer programme or its subroutines, as sometimes implied by statutes[7] and academic literature. Rather, the core of the patentable invention often[8] consists of algorithms, understood broadly, along with their practical implementation couched (to obtain the grant and maximum protection) in terms of 'systems', 'methods', 'devices', 'products' and the like. Occasionally, the instructions in the patent specification might lead to an educated guess as to the actual programme that can be projected from these instructions.[9] It is

---

[4] See cases: *Finisar v DirecTV* (2007-1023, 1024) (Fed Cir 2008); *Aristocrat Technologies Australia v International Gaming Technology* (2007-1419) (Fed Cir 2008); also *LizardTech v Earth Resource Mapping Inc* 424 F.3d 1336 (Fed Cir 2005). As a particular U.S. instance of this, see eg R Merges, "Software and Patent Scope: A report from the Middle Innings" (2007) 85 *Texas Law Review* 1628-31, at 1652.

[5] See note 4 above and eg S Lindholm, "Marking the Software Patent Beast" (2005); also available at http://ssrn.com/abstract=642123 (accessed 19 July 2011), 18; J Cohen and M Lemley, "Patent Scope and Innovation in the Software Industry" [2001] 89 *California Law Review* 1-58, 24.

[6] D. Burk-M. Lemley : *The Patent Crisis and How the Courts Can Solve It*, University of Chicago Press, Chicago 83-85;

[7] This applies to Europe, where Article 52 of the 1973 European Patent Convention *expressis verbis* mentions that computer programmes are only non-patentable to the extent that an attempt is made to patent a computer programme *as such.*

[8] In software patenting, understood broadly, algorithms, data structures, protocols, software architectures and designs have in practice been granted patents.

9 Thus, in principle, when an algorithm is implemented in a computer programme, the actual programme, whether in source or object code, together with supporting documentation, is protected by the copyright, while the functional idea of the algorithm underlying the programme remains in the

possible that a patent application may be drawn narrowly enough that it crystallises into an actual computer programme, and such applications do exist.

Most programmes, however, are not protected directly. The correct subject-matter[10] is the underlying algorithms or sometimes even logical designs[11] that may cover tens of different types of software as eventually implemented in a given computer programme. This *fata morgana* of patentable subject-matter is arguably a major point of confusion[12] in the field of intellectual property at the turn of the Millennium, together with the protection of "databases", particularly in the EU.[13]

Whether the broad scope of patentability and the proliferation of granted patents is a good or bad thing, is a different question. It is perhaps useful first to discern what is actually happening, instead of proceeding on the basis of the perceived patent protection of computer programmes. The main focus of this paper is that the algorithms underlying computer programmes are the focal point of discourse on patentability of computer software. The similarity or distinction of two algorithms therefore assumes a dominant role when a patent infringement dispute or the issue of novelty or obviousness arises.

In subsequent chapters we will address individual arguments, and some non-arguments, that could be presented in support of the sameness or distinctness of two given computer algorithms. We can provide justifications for some of these arguments, but in many cases they merely encode what we - as practitioners and educators in computer science and software engineering - consider to be obvious to a person having ordinary skill in the art. Further, chapters 5 and 6 discuss the criteria for considering algorithms, and their variations and applications, as novel or non-obvious given the current state of the art.

---

realm of patent law. In practice, the demarcation between the two is less clear, see eg R Ballardini, "Scope of Protection for the Functional Elements of Software, in Search of New IP Regimes" (2010) *IPR University Center* 27-62 available at: http://papers.ssrn.com/sol3/papers.cfm?abstract_id=1599607 (accessed 19 July 2011).

[10] The permissive approach to accepting broad claims has been partly tested in the USA recently in *Bilski et al v Kappos, Under Secretary of Commerce for Intellectual Property and Director, Patent and Trade Mark Office*, 561 US Supreme Court No. 08-964 (28 June 2010), hereafter '*Bilski v Kappos*' and its antecedent brethren *re Nuijten*, 500 F.3d 1347 and *re Comiskey*, 554 F.3d 967. See eg J Duffy: **"**The Death of Google's Patents?" (2008) available at http://www.patentlyo.com/patent/ 2008/07/the-death-of-go.html (accessed 19 July 2011); C Harkins, "Throwing Judge Bryson's curveball: a pro patent view of process claims as patent-eligible subject matter" (2008) 7 *J Marshall Rev Intell Prop L* 701; the references to cases are found in the articles. On logical designs, see eg *Sun Java Enterprise System Deployment Planning White Paper* at: http://download.oracle.com/docs/cd/E19263-01/817-5759/log_architect.html

[11] For different models and stages of software engineering and development, see e.g. "Software Process Models", available at http://www.the-software-experts.de/e_dta-sw-process.htm (accessed 19 July 2011).

[12] See eg P Virtanen, "The Emperor's New Clothes: Determining the Subject-matter of Software Patents" [2010] 79 *Nordiskt Immateriellt Rättskydd* 1-13; see also references therein in footnote 40.

[13] See e.g. P Virtanen, *Evolution, Practice and Theory of European Database IP Law*, (Lappeenranta: Acta Universitatis Lappeenrantensis, 2008); P Samuelson; J Reichman and P Samuelson: "Intellectual Property Rights in Data?" [1997] 50 *Vanderbildt Law Review* 51-166.

2. Sameness-Preserving Transformations

We will use as our running example the domain of sorting an array of values into ascending order.[14] This domain has the virtue of being easy to describe, requires modest programming skills, and is rich with analogues in everyday life, yet sufficiently complex to be solved by a large number of different algorithms. Some of these are classics in computer science that are included in all university-level curricula of computer science and software engineering.[15] We start with the perspective of a computer scientist, and then compare this with the extant legal concepts and a short outline of the relevant legal doctrine.

Our first sorting algorithm is Selection sort. The basic idea is to find the minimum value of the array, swap it with the first value in the array, and repeat these steps for the remainder of the array, ie starting at the second position, then third, etc until the whole array has been traversed. That was an informal and arguably incomplete description of Selection sort; its implementation in a programming language provides a more formal description. With suitably declared variables, the actual algorithm is:

```
for (i = 0; i < N - 1; i++) {
    /* Find the index of the lowest x[i], ..., x[N-1]
       into min. */
    min = i;
    for (j = i + 1; j < N; j++)
        if (x[j] < x[min])
            min = j;
    /* Swap x[i] and x[min]. */
    tmp = x[min];
    x[min] = x[i];
    x[i] = tmp;
}
```

The above code is C, but coincidentally it also works, with no or at most trivial changes, in two other widely used programming languages, C++ and Java. A sequence of trivial transformations can translate the above code to numerous other programming languages, such as Ruby:

```
0.upto(x.size-1) do |i|
    min = i
```

---

[14] We readily admit that we do not know whether sorting, or "producing a totally ordered multiset", as a mathematician might call it, would be a mathematical algorithm in the eyes of the US Supreme Court as in *Gottschalk v Benson*, see note 1 above. In its abstract formulation, the sorting operates through a series of arithmetic operations that are mathematical but the sorting algorithm can subsequently be *applied* to many different ordering tasks, starting from alphabetic ordering and proceeding to numerous everyday and professional or industrial sorting operations in which the underlying mathematical principle is more distant from the relevant, concrete task such as sorting letters in mail, scheduling flights, or indexing a book.

[15] T Cormen, C Leiserson, R Rivest and C Stein, *Introduction to Algorithms* (Massachusetts Institute for technology Press, 2009), now in its third edition, is probably the most popular and most commonly cited introductory book on the theme of algorithms. It makes reference to all the sorting algorithms presented in this paper, and provides an overview of what should at minimum be considered 'ordinary skill in the art'.

```
        (i+1).upto(x.size-1) do |j|
            min = j if x[j] < x[min]
        end
        x[i], x[min] = x[min], x[i]
    end
```

Although the two implementations of algorithms above are in different programming languages, they are undoubtedly the same algorithm. If one of these codes violates a patent, therefore, then surely the other would also.

In copyright and trade secrecy violations the debate is, alas, not that clear, because the goal is to establish illicit derivation of one from the other, not independent discovery or derivation. The most apparent similarities (the variables named x, i, j and min) are quite natural in their place and could be a coincidence or could be inherited from some common source, such as a text book or a public domain pseudo-code implementation on the Internet. Hence their similarity bears little if any evidence of one implementation having been derived from the other. Only if hundreds or thousands of variable names coincide with few exceptions would suspicions arise. The programmes shown above are simply not large enough to determine their relation to one another.

Sameness-preserving transformations can be deeper than the superficial variance in indentation and lexical conventions. The description of the idea in Selection sort does not specify how one searches a minimum value in the array; an iteration from the end of the array to the first remaining item is just as viable, giving the inner loop:

```
        min = N - 1;
        for (j = N - 2; j >= i; j--)
            if (x[j] < x[min])
                min = j;
```

Further, should one want to sort the array in descending instead of ascending order, one repeatedly searches for the maximum instead of the minimum value in the remaining array. Alternatively, the same effect could be reached if one were to restructure the outer iteration to begin from the last, then proceed to the penultimate and towards the beginning of the array. Together these two variations cancel out each other.

```
    for (i = N - 1; i >= 1; i--) {
        max = i;
        for (j = i - 1; j >= 0; j--)
            if (x[j] > x[max])
                max = j;
        tmp = x[max]; x[max] = x[i]; x[i] = tmp;
    }
```

A software engineer comparing this to the original code, especially the structure of their iterations, may wish the comfort of more than a few seconds to verify that these codes indeed perform the same thing and do so essentially by employing the same algorithm. Yet, as Selection sort is usually described in the computer science literature, little doubt of that should exist.

In patent law and practice, the similarity of objects has a bearing on two important premises for patentability: novelty and the existence of an inventive step, or in other words, non-obviousness. In short, an invention is not patentable, for lack of novelty, if the claimed subject matter was disclosed before the date of filing. Even if there is no anticipation and the invention is novel, the invention is not patentable if an imaginary person, having ordinary skill in the art, would know how to solve the problem to which the invention is directed, by using the same mechanism. In such a case, the invention, even though not the same as the prior art, lacks an inventive step. The invention is similar enough that a skilled professional would render it obvious.[16]

Sameness-preserving transformations can be compared with the Doctrine of Equivalents in Patent law and IPR management parlance. An infringing device, process or other embodiment of invention that does not fall within the literal scope of a patent claim may nevertheless be considered equivalent to the claimed invention. This extension to the literal infringement of the patent is limited to trivial or insignificant changes, just as the set of sameness-preserving transformations discussed in this section.

It is also possible however that a device, even though it falls within the literal description of a claim of patent, to perform the same or a similar function as the patented device but in a way that is substantially different. The suspect invention is therefore, in principle, so far removed from the invention as disclosed in the patent that it is considered in law to be an entirely different thing. Even if a claim as literally read includes the impugned thing, claims are limited by construction to cover the invention disclosed in the patent and to exclude a thing which is different from the disclosed invention. This result is sometimes known as the "Reverse Doctrine of Equivalents".[17]

3. Dissimilarity in the underlying idea

We have argued above that despite "trivial" and mechanical changes, either to the *implementation* of the algorithm in a programming language or in the *idea* of the algorithm, the identity of the algorithm is retained. As we shall see in this Section, there is, however, a surprisingly low limit to how far such reasoning can go.

Recall that the basic idea of Selection sort (on arrays) is to find the minimum value of the array, swap it with the first value in the array, and repeat these steps for the remainder of the array, etc. In Insertion sort the basic idea is to take the first value in the input array, insert it into the correct position in the output array, and repeat these steps for the remainder of the input array until no more input elements remain. Notice

---

[16] This is of course a very rough sketch of the general idea, since these are some of the core concepts of patentability. For a good introduction to the main concepts and comparison between the US and European systems shedding more light on both see eg C Nard, "History and Architecture of the Patent System" *Bocconi IP Summer Transatlantic Academy Paper* (25th June 2007); on a good account of the obviousness, see J Duffy, "Inventing Invention: A Case Study of Legal Innovation" (2007) 86 *Texas Law Review*, available at: http://papers.ssrn.com/sol3/papers.cfm?abstract_id=1087067## (accessed 19 July 2011).

[17] J Marr, "Foreseeability as a Bar to the Doctrine of Equivalents" (20 May 2003), available at: http://ssrn.com/abstract=410027 (accessed 19 July 2010); W Long, "United States Doctrine of equivalents: where we now stand" (2007) available at: http://www.buildingipvalue.com/07US_Can/p.127-130%20Sutherland.pdf (accessed 19 July 2010).

the subtlety: we changed the search from the selection of the correct value to a search for the correct place to insert. A moderately dexterous person can compare these two algorithms in practice on a suit of playing cards; while the difference in ideas may seem trivial, in computer science these two algorithms are considered two different algorithms.

As with a hand of playing cards, the input and output arrays can be made to occupy the same physical array. One such way to implement Insertion sort is given below:

```
for (i = 0; i < N; i++) {
    v = x[i];
    for (j = i; j >= 0 && x[j - 1] > v; j--)
        x[j] = x[j - 1];
    x[j] = v;
}
```

For sake of completeness, and to facilitate further debate, we include a third elementary sorting method, Bubblesort. This also concludes our exposure to source code.

In Bubblesort the idea is to swap the first value in the array with all smaller values in the remainder of the array, and then repeat the same with the rest of the array, etc. If one ignores "stableness" which will be discussed later, Bubblesort and Insertion sort can be derived from each other by sameness-preserving transformations, but they would be so elaborate and contrived that other arguments against their sameness (presented in this paper) would prevail.

```
for (i = 0; i < N - 1; i++)
    for (j = i + 1; j < N; j++)
        if (x[i] > x[j]) {
            tmp = x[i];
            x[i] = x[j];
            x[j] = tmp;
        }
```

According to computing folklore, Selection and Insertion sorts are the fastest sorting methods for array sizes up to around ten or twenty. The number of comparisons and array accesses however grows quadratically with N for all the elementary sorting algorithms discussed so far, and more sophisticated algorithms become faster.

Divide and conquer is a frequently used paradigm for deriving algorithms. The large problem is recursively broken up into two or more subproblems until they become easily solvable and then the solved subproblems are combined into a solution to the original problem.

When applied to sorting, this paradigm could be applied for example as follows: Choose a random value, the pivot, in the array to be sorted. Move all values smaller than the pivot to the beginning of the array and all values larger than the pivot to the end of the array. Repeat the above two steps recursively to the subarrays until they contain at most one element (or as an optimisation, until Selection or Insertion sort becomes an efficient sorting algorithm for the subarrays). This is roughly the

underlying idea of Quicksort, invented in 1960.[18] It turns out that the two first steps, also called partitioning of the array, can be performed in time linearly proportional to the size of the (sub)array. Therefore a lucky choice of pivot values will lead to a computational requirement of N + 2*(N/2) + 4*(N/4) + ... + N*(N/N) = N $\log_2$ N comparisons or swaps. For sufficiently large N this is less than the quadratic computing time required by the elementary sorting methods.

In Quicksort the linear-time array partitioning work is performed when breaking up the problem into smaller subproblems and no work needs to be done when combining the solved subproblems into a sorted original array. An alternative application of Divide and Conquer would be to perform no work in breaking up the array but instead perform a linear-time subarray merging when combining the sorted subarrays into the whole sorted array. This subtle change from partitioning when breaking up to merging when combining leads changes the resulting algorithm from Quicksort to Merge sort. It exhibits similar performance characteristics as Quicksort, but the computer science community again considers it a new sorting algorithm in its own right.

It has been shown mathematically that no sorting algorithm based on comparing entire values can be faster than Quicksort or Merge sort by more than a constant factor. By breaking the compared values into parts, however, one can sort even faster. The idea is to treat the keys as a sequence of units, for example characters. We then reserve one bucket for each distinct character and with one pass through the values to be sorted we can place all values which begin with character 'A' into the bucket for characters 'A', values which begin with 'B' to the bucket for 'B's, etc. Next, for all buckets containing a nontrivial number of elements we recursively perform the same sorting procedure again but now place the values into subbuckets according the second letter, then the third etc. The final sorted array is obtained by concatenating all the buckets. Analysing the number of operations would show that the number of operations depends linearly on the product of the number of values and on the number of characters that form the (longest) value.

This sorting algorithm has been used by postal services for ages. A variation of it was patented for the punched card sorting machines of Herman Hollerith in 1889. Hollerith founded the Tabulating Machine Company in 1911, which merged with some other companies in 1924 to form IBM. Later it has been reinvented several times, most notably by Harold Seward for general purpose computers in 1954, and eventually it was dubbed Radix sort.

4. Sameness in functionality

Computer scientists and software engineers may colloquially call two algorithms equivalent if they produce the same output for the same input, possibly with attention also to the required memory and computing times. This unfortunate overloading of the term must, however, not be confused with the consideration of equivalence of claims and description deriving from the algorithm in the software patenting context. Yet arguments for non-equivalence (in the computer science sense) can in some cases be useful in arguing dissimilarity of the algorithms. The converse may not be true as

[18] C Hoare, "'Partition: Algorithm 63,' 'Quicksort: Algorithm 64,' and 'Find: Algorithm 65'" *Communications of the ACM* (1961) 4(7), 321-322.

it is often implausible that the given algorithm can be the only one producing the same output.

Of course, two algorithms can be considered merely trivial variations of each other even if their output would be different. For example, to change the sorting algorithms output from ascending to descending order requires such a trivial variation in the code that no one would consider the modified code to implement a different algorithm. A subtler change in output, for example the stability of sorting – whether the sorting algorithm preserves the relative order of equal values – can, however, be a distinguishing factor. Insertion sort, as we presented it, is not stable whereas Bubblesort is. Quicksort, by default, is not stable, whereas Merge sort can be.

As the output of the algorithm one might also regard the memory consumption and computing time.[19] By this argument one could show, for example, that none of Selection sort, Quicksort and Radix sort can be similar to each other.[20] Similarly, Quicksort is evidently different from Mergesort because an unfortunate choice of pivot elements causes Quicksort to assume a quadratically growing computing time whereas Mergesort can be guaranteed to consume never more than time proportional to N log N. Furthermore, one can argue that Selection sort and Insertion sort must be different algorithms since Insertion sort can process a readily sorted input array in time linearly proportional to N whereas Selection sort always requires time proportional to N^2.

5. Sameness through abstraction of application

Writing software can be a very laborious task. Competent programmers therefore try to avoid duplicate work by writing easily reusable code. This can be achieved by writing the code generic (abstract, parametric) so that aspects of the code can be changed without any deeper understanding of the algorithm itself. Providing powerful (yet understandable and safe) methods of abstraction and generality is in fact one of the major driving research questions in programming language design.

In sorting algorithms two aspects would immediately be generalised by any competent programmer: the type of *values* to be sorted and the code that performs the comparison. By generalising the type the very same code (and consequently the same algorithm) is able to sort integers, strings, names of people, phone numbers, dates, addresses, etc. Varying the comparison function not only corresponds to the change in type, but can also counter for whether the resulting order is ascending or descending, whether certain characters (such as 'u' and 'ü') should be treated as equal, or whether parts of the value should be ignored. Consequently, there is no reason why the very same sorting algorithm could not sort the appendix of references in a book, a row of integers in lotto, or the chronological schedule of departing flights on the airport.

While bringing generality to an algorithm may involve varying levels of innovation, the reverse process of specialisation (instantiation) of a given generalised algorithm is usually considered trivial. A computer scientist or software engineer can instantiate a

---

[19] This has also been suggested by A Chin "Computational Complexity and the Scope of Software Patents" (1998) 39 *Jurimetrics Journal* 17-28.

[20] This argument would also be used to argue that there are a few variants of Radix sort mentioned in Section 6 that could be considered different algorithms.

readily generalised sorting algorithm to sort, for example, names of people in a few seconds of typing. Assuming the physical devices were readily available, s/he will regard the sorting of a deck of cards or a herd of cattle according to weight as equally obvious, void of any inventive step.

The discovery of applicability of the algorithm and various preparatory and subsequent steps could easily however result in a non-obvious idea. A classic example, and an illustration as to how inventions build on each other, is the following amalgam of inventive applications of previously known algorithms from computer science: the use of Discrete Fourier Transform[21] for efficient multiplication of large integers,[22] in modular exponentiation, and to construct a public-key cryptography protocol.[23]

In the patenting law context, the motivation to obtain the broadest possible monopoly and the widest available protection for the invention is an incentive to couch applications in terms that are as generic as possible. An example of a broad patent can be found in the well-known U.S. grant to the above mentioned RSA public key encryption algorithm, claimed in altogether 40 claims. The first such claim involved essentially a cryptographic communications system comprising any communications channel with means of encoding and decoding for word signals but the *de facto* limitations on scope arose from the specific - and famous - encoding and decoding method.

This approach may also backfire, however: the classic U.S. Supreme Court case of *O'Reilly v Morse*[24] dating back to 1850s provides a good example of this. In the original eighth patent claim one Samuel Morse claimed *any use* of electromagnetism for printing intelligible signs, characters, or letters at a distance. The Supreme Court found the claim to be too broad, drawing a parallel with the case of a patent of a mere principle, and thus inadmissible.

The outer limits of the scope of patentability, relevant in the software and algorithm context, come from general exclusions to patentability, provided by Article 52(2) and (3) in the European Patent Convention (EPC). These bar from patentability *as such* discoveries, scientific theories and mathematical methods, aesthetic creations,

---

[21] In mathematics, the Fourier Transform converts a cyclic function to its frequency and phase components. The Discrete Fourier Transform (DFT) performs the same on a finite number N sampled points of the function. The Fast Fourier Transform (FFT) is an algorithm for computing the DFT in time proportional to N log N. Such an algorithm was apparently first discovered by the famous mathematician Carl Friedrich Gauss in 1805, but usually it is attributed to J Cooley and J Tukey, "An algorithm for the machine calculation of complex Fourier series" (1965) 19 *Mathematics of Computation* 297–301.

[22] If the N points given to the DFT are coefficients of polynomials or digits of large integers, then the pairwise product of the transformed points corresponds to the multiplication of the polynomials or integers. Since N multiplications require time proportional to N and DFT and inverse DFT require time proportional to N log N, this results in an efficient algorithm for multiplying large polynomials and integers. See A Schönhage and V Strassen, "Schnelle Multiplikation großer Zahlen", (1971) 7 *Computing* 281–292.

[23] U.S. Patent no. 4,405,829 "Cryptographic communications system and method" (20 September 1983), filed December 1977.

[24] *O'Reilly v Morse* 56 U.S.62 (1853), at 112-116.

schemes, rules and methods for performing mental acts, playing games or doing business, and programmes for computers and presentations of information.

By comparison, in the American context, exceptions to patentability evolved as a result of case law and are still found therein. In *Diamond v Diehr*[25], for example, a case concerning a patented rubber curing method involving an Arrhenius equation[26] in algorithmic form, the US Supreme court excluded from patent protection *laws of nature, natural phenomena, and abstract ideas*. Much effort has gone into the inquiry as to what belongs to these categories and what does not.[27]

Another set of limitations upon the scope of the claims is found in the requirement of sufficient disclosure already mentioned above,[28] which in practice often has the effect of reducing the scope of the patent application. The claim in *Morse*[29] was also rendered void on this basis. In a sense, patent law gives the applicants a certain freedom of discretion to define the scope of their claimed invention, provided the mentioned exclusions do not apply, and the application is sufficiently disclosed.

6. Sameness Through Abstraction of Internal Details

In some cases, forms of abstraction may require deeper modification than occurs through changes to a few lexical items in the code that implement the algorithm. For example, inspecting the codes of Selection sort, Insertion sort, Bubblesort, and should we have presented them also Quicksort and Mergesort, one would notice that the algorithms never arrive at a value in the middle of the array without first traversing through adjacent places from either edge of the array. This moving to adjacent values and also the swapping of values can be performed equally well on linked lists as on arrays: incrementing the indexes i and j to the array x is replaced by following the next-fields in the linked lists nodes; swapping values in two indexes is replaced by a handful of linked list manipulation statements. In the case of Quicksort, a doubly linked list will be needed, but consequently Quicksort can also be made stable. This was such a significant variation that its description was published in a peer-reviewed journal in 1981.[30] Yet the computer science community still regards it as variation of Quicksort rather than a new sorting algorithm. Analogously, Radix sort is frequently modified, for example, by changing the order of traversing the characters in the keys

---

25 *Diamond v Diehr,* 450 U.S. 175 (1981), at III; the case cites numerous other cases and the discourse has not abated ever since, the current USSC *Bilski v Kappos* case evincing this.

26 The Arrhenius equation is a simple and markedly accurate formula for the temperature dependence of the rate constant, and consequently to the rate of a chemical reaction. The equation is coined after Swedish chemist Svante Arrhenius, who provided a physical justification and interpretation for the formula in 1889.

27 Of latest developments in the USA, see *Bilski v Kappos* at note 10 above and, *Prometheus Laboratories , Inc v Mayo Collaborative Services and Mayo clinic Rochester* CAFC 17 December 2010, 2008-1403. The Supreme Court had earlier remanded the case to the Federal Circuit to reconsider the case "in light of Bilski"and the Federal Circuit redecided the case; The Supreme Court granted Petition for a writ of certiorari again on Jun 20 2011, No. 10-1150 and the case is *sub judice*.

28 U.S. Patent Code § 112; additionally, in the US the "best mode requirement" also limits the spectre of the claims in this respect.

29 See note 24 above, at 117-121.

30 D Motzkin, "A Stable Quicksort" (1981) 11 *Software – Practice and Experience* 607-611.

and the method of implementing "buckets", but these modified Radix sorts are still usually recognised as notable variations rather than entirely new sorting algorithms.

A similar process of abstraction of internal details can be used to create amalgamates of sorting algorithms. For example, if the recursive application of Quicksort is abstracted to a subroutine which reverts to Insertion sort if the array to be sorted is shorter than, say, twenty elements, we have derived an eighth sorting algorithm by reusing two of the seven sorting algorithms already presented in this paper. While such designs can have significant practical value, such amalgamate algorithms are rarely considered non-obvious.

Exempli gratia, a patent search directed at over seventy authors referred to as inventors of new sorting algorithms,[31] found that no one had filed a patent on the ideas they had published, either through academic channels or on the Internet.[32] This suggests the limit of non-obviousness in patenting and the (academic and computer science related) treatment of two algorithms as the same are not identical issues. For the sake of argument, however, it may be proposed that the finding of a new algorithm as described in the example above parallels the finding of non-obviousness in the patenting field. Consequently, the next question that arises is whether the criterion of non-obviousness as understood among software engineers and computer scientists is different from that applied by patent engineers and managers who determine the ideas that enter the patenting process.

A positive answer to the above question would raise a contradiction. The non-obviousness or inventive step in patenting is directly linked both to the existing knowledge made public in the field, and to the expertise of an average professional skilled in the relevant trade.[33] By article 56 EPC, "An invention shall be considered as involving an inventive step if, having regard to the state of the art, it is not obvious to a person skilled in the art." By the same token, pursuant to § 103 (a) of the U.S. Patent code:

> A patent may not be obtained though the invention is not identically disclosed or described as set forth in section 102 of this title, if the differences between the subject matter sought to be patented and the prior art are such that the subject matter as a whole would have been obvious at the time the invention was made to a person having ordinary skill in the art to which said subject matter pertains…

These two factors should ideally return the appraisal of the merits back to the computer scientists and their skill in the first instance. Should the difference

---

31 http://en.wikipedia.org/wiki/Sorting_algorithm and pages referred from there. We do not claim Wikipedia to be the most authoritative source, but it is either more extensive or more up-to-date than the printed alternatives.

32 As a near exception, professor Michael S Paterson *et al* have filed the U.S. patent 6,185,220 on a method for laying out sorting networks on VLSI, but it is unclear whether the actual sorting networks are covered by the patent.

33 For more detailed treatment of the inventive step and non-obviousness, see e.g. L Bently and B Shermann. *Intellectual Property Law*, (Oxford: OUP, 2009), at 488-506; an interesting study from Australian standpoint is H Moir, "How high is the inventive step?: Some empirical evidence" (2009) *EPIP conference paper*, available at: http://www.epip.eu/conferences/epip04/files/MOIR_Hazel.pdf (accessed 19 July 2011).

nevertheless exist, one of the reasons for the discrepancy lies in that algorithms are[34] patented when they are embedded in a particular application in a certain field. Another one may well be that the searches for prior art are far less than optimal and result in the granting of overlapping patents, which is widely recognised as a problem.[35]

Alternatively, one might ask whether computer scientists granted patents on early elementary sorting algorithms, such as Selection and Insertion sort, are distinct algorithms of a lower status than sorting algorithms developed later. This might be true simply because as the field became more thoroughly researched (and taught), the general level of competence rose. People also have a natural tendency to limit the number of items to learn and remember, and hence tend to cluster or abstract more diverse items into one group as the number of items increases. One may further ask whether the same should apply with equal strength to the process of accepting patent applications.

The practice in patenting is slightly different. While at the infancy and pioneer stages of an industry one sees patents with broad scope, patenting does not taper off later. Instead patents become more numerous and increasingly narrow in scope, provided that the demand for products and services produced induces the research and development efforts to this end. This applies particularly in the software industry that has often been characterised, as mentioned above, as cumulative, incremental and competitive or co-petitive instead of relying on discrete inventions and stand-alone products building upon them.[36]

7. Discussion

This paper has presented a number of arguments that a computer scientist might use to argue that two given algorithms[37] are two different algorithms or such trivial variations of the same algorithm that the variation falls below expected competence of a contemporary software engineer or computer scientist. Unfortunately, and as one would expect, no waterproof method exists for testing this.

Ideally one would like to devise a psychometric test in which a handful of reputedly competent software engineers and computer scientists and the inventor are given the same problem to be solved. If these experts can within, say, an hour of discussion present solutions close to those of the inventor, then a sufficient level of non-obviousness has clearly not been reached.

Unfortunately problems are seldom sufficiently well-defined and consequently there is rarely ever a chance for a group of experts to converge upon a solution near to that

---

34 Or rather should be provided the doctrine on tying the invention to the concrete application prevails also in practice.

35 See e.g. M Lemley, "Rational Ignorance at the Patent office" (2001) 95 *Northwestern University Law Review* 1-32, at 1-2, available at: http://papers.ssrn.com/sol3/ papers.cfm?abstract_id=261400, which provides, in part, a slightly divergent view concerning patents overall, and an ample list of references at footnote 1.

36 R Merges and R Nelson, "On the Complex Economies of Patent Scope" (1990) 90 *Columbia Law Review* 839, at II and III.

37 This is presented in the setting of general purpose computers.

of the inventor. In such a case, the test could be devised so that a part of the group searches the literature and the Internet for similar existing algorithms, and presents them to the remaining group along with the algorithm of the inventor. They then judge which algorithms are most similar to one another. If the algorithm of the inventor is judged between themselves to be more similar to one of the prior algorithms than the existing algorithms, then the algorithm of the inventor would not be judged as sufficiently non-obvious.

The use of psychometric tests in the appraisal of patentability of an algorithm and/or the implementing computer programme, or the validity of an existing software patent, must be evaluated against the backdrop of whatever is in current use. During the patent prosecution phase, a patent examiner first compares and evaluates, on the basis of his or her experience, the patent application and its specification[38] with the description, claims and references together with the information available from the supporting databases and arguments brought forward by the inventor and patent agent or attorney. These parameters are brought to bear with the application of the patent-granting or rejecting "algorithm", the legal code with inherent elasticity that establishes the rules governing the appraisal.

In the second alternative, the judiciary makes the final decision. By the same token, an evaluation is based on patent documents, witnesses, expert witnesses, other admissible evidence, presentations of the parties involved *etc* as input, together with relevant legal rules for determination of an infringement or related validity contest. After the execution of the legal "algorithm" a judgment is passed.[39] These descriptions are, of course, incomplete and provided here as a rough outline of the relevant procedures.

As mentioned above, short recourse to U.S. practice implies that there has been a trend to employ a relatively high standard of non-obviousness, while the patents that are granted are entitled to broad protection.[40] While reliable statistics are not easy to come by, the higher standard has already facilitated more than 100,000 U.S. software patents in 2002.[41] The breadth of scope has its own interesting tale to tell. As the importance of the courts and particularly that of CAFC[42] is widely recognised in the

---

38 At the level of language, the EPC speaks only of patent application including then, *inter alia*, the description and claims together with possible drawings, eg UK and U.S. law uses word specification to cover the same aspects. See L Bently and B Shermann. *Intellectual Property Law*, (Oxford: OUP, 2009), at 333; R Schechter and J Thomas, *Schechter and Thomas' Intellectual Property: The law of copyright, patents and trademarks* (St. Paul, Minnesota: Thomson-West, 2003), at 393-394. This legal typology has to compared with software development where one step in the process is also specification, the task of precisely describing the software to be written, preceded by domain analysis and then succeeded by creation of software architecture referring to abstract representation of the system, implementation in the form of coding and writing the requisite documentation, and testing and debugging the programme.

39 For an illustrative example of how a court performs the decision making, see *Microsoft Corp v I4I Limited Partnership et al*, U.S. Supreme Court, June 9, 2011, No 10-290, and earlier district and appeals court decisions.

40 D Burk and M Lemley. "Designing Optimal Software Patents", in: R Hahn (ed), *Intellectual Property Rights in Frontier Industries; Software and Biotechnology* (AEI Press, 2005), Chapter 4.

41 F Lévêque and Y Ménière, *The Economics of Patents and Copyright*, (Berkeley Electronic Press, 2004), available at: http://www.bepress.com/leveque/ (accessed 19 July 2011), at 47.

42 See for example, R Thomas, "Debugging Software Patents: Increasing Innovation and Reducing

determination of actual patent scope, the language of the claim has been allowed to prevail at a high level of abstraction when the software invention has been disclosed.

In the European EPO/EPC settings, in spite of the "as such" exclusion from patentability regarding software, the number of EPO issued software patents in 2002 was some 30,000.[43] While some software patents concern only particular instances of a computer programme or its subroutines embracing the practical implementation of the underlying idea protected by a patent, the system has allowed claims to cover a broader area of application.[44] The sheer statistics show that the similarity appraisal and the methodology used has remarkable practical significance.

8. Conclusion

In this paper, we have reviewed arguments that may be used in determining the similarity of algorithms, and compared the limits of novelty and particularly of non-obviousness required in granting patents. The broad scope for patenting software currently implies that not only computer programmes but also the underlying algorithms may be patented, and this makes the discourse noteworthy. Computer science, software engineering and law have slightly different perspectives, inherent to each respective paradigm, on the patentability of software and of underlying algorithms. These views meet in the practice of patenting and related litigation. We have attempted to present the two standpoints, and illustrate the possible differences with a view to increasing the understanding of the interplay between the two.

Uncertainty in the Judicial Design of Optimal Software Patent Law" (2008), to be published in Santa Clara Computer and High Technology Law Journal. Also available at SSRN: http://papers.ssrn.com/sol3/papers.cfm?abstract_id=1126450 (accessed 19 July 2011), at 4.

43 See note 41 above.

44 See eg K Beresford, *Patenting European Software under the European Patent Convention* (London: Sweet & Maxwell, 2000), at 54 and the examples therein.