Proof-of-Concept: Creating "Fuzzy" Sorting Algorithms

Stephany Coffman-Wolph
West Virginia University Institute of Technology
405 Fayette Pike, Montgomery, West Virginia 25136
sscoffmanwolph@mail.wvu.edu

Abstract

Sorting algorithms are common tools for manipulating data and used in both standalone circumstances and within larger more complex algorithms. Thus, it is highly desirable for sorting algorithms to be efficient in terms of storage and computation. By applying the concept of fuzzy logic (an abstract version of Boolean logic) to any well-known algorithm, it generates an abstract version (i.e., fuzzy algorithm) that often results in computational improvements. Although the algorithm may produce a less precise result, this is counteracted by gaining computational efficiency with minimal acceptable trade-offs (e.g., small increase in space requirements, loss of precision). Using an established three-step framework for fuzzification of an algorithm, the resulting new fuzzy algorithm goes beyond a simple conversion of data from raw to fuzzy data by converting the operators and concepts within the algorithm to their abstract equivalents. The purpose of this paper is to demonstrate, as a proof-of-concept, that sorting algorithms can be converted into their corresponding fuzzy sorting algorithms. This paper discusses the preliminary results of: (1) how to apply the general framework by developing the corresponding fuzzy algorithms for a variety of sort algorithms, (2) the success of applying the framework through the development of several fuzzy sort algorithms including fuzzy shell sort, fuzzy strand sort, and fuzzy bucket sort, and (3) the possible applications and benefits of these fuzzy sort algorithms.

Motivation

Sorting algorithms are used prolifically and have been thoroughly researched. This resulted in a wealth of algorithm options for sorting. Every sort algorithm produces the same result, but many have unique properties or techniques (e.g., used for an almost sorted list, reversing a sorted list, or easily adding an additional value). Given the frequent use of sorting algorithms, being able to sort efficiently (in regards to both space and time) is highly desirable. If precise sorting is not necessary, a fuzzy sort algorithm could be advantageous as fuzzy algorithms take advantage of integer calculations and generally execute fewer steps than the equivalent traditional non-fuzzy algorithm [Coffman-Wolph and Kountanis, 2013a]. (In the event that high precision is required, the results from the fuzzy algorithm could be defuzzified and run through a non-fuzzy algorithm to complete the sorting).

Introduction and Background

The next five sections provide the necessary background materials for this paper. The first section discusses the three-step fuzzy algorithm framework used to convert a traditional non-fuzzy algorithm into the corresponding equivalent fuzzy algorithm. The second section briefly discusses sorting algorithms and their use within the paper. In the third section the author discusses types of data and the fuzzy sorting algorithms. The fourth section covers the notation used within the paper to denote when an element is fuzzy. In the final section, the author provides a description of what it means to be fuzzily sorted.

The Fuzzy Algorithm Framework

The general broad-spectrum framework for the fuzzification of any algorithm is a simple three-step conversion process that converts any algorithm from a traditional non-fuzzy version into a corresponding fuzzy version [Coffman-Wolph and Kountanis, 2013b]. The three steps of the framework are as follows:

1. Decide what can/should be fuzzified and determine the category of each piece to be fuzzified
2. Fuzzify each piece (identified in step 1) based on the category
    a. Scale/normalize the data, then fuzzify the data
    b. Fuzzify the operators
    c. Fuzzify the concepts
3. Defuzzification (if needed/applicable)

As can be seen in the framework, there are three main categories of components within an algorithm that can be fuzzified or made more abstract: (1) data, (2) operators, and (3)

concepts. The fuzzification of data is the process of taking "raw"/non-fuzzy data and converting it into fuzzy data. The fuzzification of operators is the process of converting a mathematical, logical, or comparative operator to its fuzzy counterpart, which operates on fuzzy sets instead of pure numbers. The fuzzification of concepts, the most difficult of the three, is the conversion of an idea into a similar fuzzy idea. Together these three core techniques create a fuzzy algorithm.

The first and second steps of the framework are needed to apply the fuzzy algorithm to a variety of problems. The first step of the framework is to make an in-depth examination of the traditional (non-fuzzy) algorithm and decide which elements to fuzzify. It is often helpful to determine what the solution will look like in the fuzzy form. This will lay the foundation for the entire fuzzified algorithm. Therefore, it can be helpful to work backwards through the algorithm to determine what elements (data, operators, or concepts) need to be changed for the solution to be produced. Defuzzification, the opposite of fuzzification, is used to convert the result back to a non-fuzzy result and can be applied to data or concepts. In some cases, this step might be unnecessary and, hence, why the framework denotes this step as optional. Deciding to do defuzzification is both problem and usage dependent. For example, if the fuzzy algorithm is being used to narrow the solution space and the information will be fed into a non-fuzzy algorithm, then it is important to develop a method of defuzzification.

This process has already been successfully applied to the following algorithms: (1) a concept-oriented fuzzification for finding fuzzy patterns [Coffman-Wolph and Kountanis, 2013c], (2) a fuzzification of both data and the operators for a fuzzy process particle swarm optimization (FP2SO) [Coffman-Wolph and Kountanis, 2013a], (3) a fuzzification of multiple algorithm components to find strategies for adversarial situations from game theory [Coffman-Wolph, 2013], (4) a fuzzification of the simple simplex method for the transportation problem [Coffman-Wolph and Coffman, Jr, 2014], (5) the fuzzification of various linear and nonlinear search techniques used in optimization algorithms [Coffman-Wolph and Coffman, Jr, 2016], and (6) the fuzzification of the Golden Ratio Search [Coffman-Wolph, 2016].

The Sorting Algorithm

Generally, sort algorithms put data "in order". This paper will explore a variety of sort algorithms ranging from the simplistic comparison sorts to slightly more complex distribution sorts. Given the commonality of these sort algorithms, this paper will focus only on the specific algorithmic details needed for fuzzification conversion and skip in-depth discussions of the algorithms themselves. To simplify the discussion, this paper will primarily consider integer data. These concepts are easily extended to other data types as discussed in the next section.

Data

Sorting algorithms are generally able to handle a wide variety of data types (e.g., numbers, strings, or combinations). The fuzzy sort algorithms will need to have the data "converted" to the fuzzy equivalent. Fuzzy algorithms generally take advantage of integer calculations and, usually, have the data in integer format. In the circumstances that require a conversion of non-integer numeric data, it is often accomplished using scaling. A fuzzy number is not a single value, but is represented by a set of values. The size/range of values is problem/data dependent and can also be affected by the level of precision the user desires. For strings, there are several possibilities for fuzzifying the data including: converting to a number or considering only the ASCII value equivalent of the first character (or the first few characters). If there is a combination of data (i.e., records from a database), then a combination of fuzzy data would be required. The greatest performance advantage is obtained using integer values, so the closer the data is to an integer format the better.

Notation

Throughout this paper, a subscript f will be used to denote when a value, variable, operator, or concept is fuzzy. Given that not everything is required to be fuzzy within a fuzzy algorithm, the subscript f will help the reader distinguish between portions of the algorithm that remain traditional/non-fuzzy and those that have been converted to a corresponding fuzzy version. As mentioned in the previous section, it is important to keep in mind that a single fuzzy value is represented by a set of non-fuzzy traditional values. For example, 5_f is a "fuzzy-five" and contains the value 5 and possibly the values 4 and 6 as well (depending on the definition of 5_f in the context of the problem). In other circumstances, 5_f could represent all the real number values in the range of 4.5 to 6.5.

Definition of Fuzzily Sorted

A traditional sort algorithm stops when the algorithm has guaranteed that all data are sorted in the required order (e.g., numerical order, alphabetical order, reverse order, or a combination). To proceed with the fuzzification of the sort algorithm, the key question to answer is: when is the $data_f$ from a fuzzy sort algorithm considered sorted sufficiently? In general, adding fuzzy logic to an algorithm creates a more abstract version of this algorithm. Therefore, with a fuzzy sort, it will be a more abstract version of the sort algorithm and, thus, the output is abstractly sorted. The definition of "abstractly sorted" will depend on the circumstances of the application and may be problem specific and/or sorting algorithm specific (i.e., dependent on the fuzzification process). These definitions have one thing in common, that the data will not always be perfectly sorted. One could describe

the data as "slightly out of order". For a fuzzy sort, one measure could be an associated percentage which represents the required/desired level of sufficient "sorted-ness". The level or percentage can be adjusted as needed for desired precision. Another measure, for a fuzzy sort, could be that the results are sorted within a specific number of significant digits or letters/symbols.

The Fuzzy Sort Algorithm

The following sections discuss the fuzzification of several well-known sort algorithms including: selection, insertion, bubble, shell, strand, and bucket sort. Each subsection below will focus on how the algorithm is converted from the traditional algorithm to a fuzzy algorithm using the three-step framework for conversion. A discussion of the likely advantages and disadvantages will be included for each of the sort algorithms.

Fuzzy Version of Selection, Insertion, and Bubble Sort

Selection, Insertion, and Bubble sorts are three of the most common comparison-based $O(n^2)$ sort algorithms [Sedgewick, 1998; Weiss, 1999]. Each algorithm can be easily converted from the traditional algorithm into a fuzzy equivalent version. The comparison operators, < and >, would be converted to $<_f$ and $>_f$ to handle the fuzzified data, but the algorithms would otherwise remain unchanged and operate in the same manner. The fuzzy selection, fuzzy insertion, and fuzzy bubble sorts do not experience any computational advantages except the possible use of integer comparisons over more complex data types. Additionally, the output of these fuzzy sorts would be dependent on the precision level the user wants to achieve when they fuzzify the data and decide the exact functionality of the $<_f$ and $>_f$. Therefore, the author concludes that while these algorithms can be converted to their fuzzy equivalents, there is no advantage to do this unless they are being applied as part of a more complex sort algorithm (e.g., the bubble sort as a last step in a more sophisticated sort algorithm).

Fuzzy Version of the Shell Sort

The traditional shell sort is an "in-place" sort and considered an abstract version of an insertion sort [Sedgewick, 1998; Weiss, 1999]. (The comb sort is a very similar "in-place" sort but uses the bubble sort as an underlying basis and basically differs only at the implementation level). When the fuzzified version of the shell sort is implemented, the algorithm will be even further abstracted. Both the fuzzy and traditional versions of the sort use a single list with portions of the list designated as sorted and un-sorted. (Traditional implementations have the sorted portion at the front of the list and the un-sorted portion towards the back of the list). During the process of converting the algorithm to a fuzzy algorithm, the concept of the x^{th} element will be fuzzified using the concept of a fuzzy set [Zadeh, 1965]. For the traditional shell sort, the algorithm takes every x^{th} element within the un-sorted portion of the list and sorts them into the correct sequence before merging them into the sorted portion of the list. In the corresponding fuzzified version of this algorithm, the algorithm will take every x_f^{th} set of elements from the un-sorted portion of the list, $sort_f$ these elements, and merge these elements into the $sorted_f$ portion of the list. The fuzzy shell sorting algorithms would likely have an advantage in situations where data naturally "clusters".

Consider the following data points: 88, 92, 98, 100, 7, 5, 3, 13, 17, 15, 24, 36, 42, 57, 55, 73, and 77 to illustration the fuzzy shell sort (and the traditional shell sort). When using the fuzzy shell sort, the first step is to determine the definition of the x_f^{th} element(s). For this small of a list, the fuzzy algorithm will use a two-element range at each x_f^{th} element. Thus, the algorithm will consider the x^{th} element and the $(x+1)^{th}$ element to be the x_f^{th} element. The fuzzy algorithm begins by selecting the 5_f^{th} elements: (88, 92), (5, 3), (24, 36), and (73, 77). These elements are sorted, removed from the un-sorted portion of the list, and placed into the sorted part of the list: 3, 5, 24, 36, 73, 77, 88, and 92 and the un-sorted portion is: 98, 100, 7, 13, 17, 15, 42, 57, and 55. The next step is to take the 3_f^{th} elements which are (98, 100), (17, 15), and (57, 55). These elements are sorted into the existing list and removed from the unsorted list: 3, 5, 15, 17, 24, 36, 55, 57, 73, 77, 88, 92, 98, and 100 and the remaining portion of the unsorted list is: 7, 13, and 42. These last three elements can be sorted into the sorted portion of the list: 3, 5, 7, 13, 15, 17, 24, 36, 42, 55, 57, 73, 77, 88, 92, 98, and 100. For each of the x_f^{th} elements, the only additional processing required is a simple swap (as there are only two elements) before sorting and merging. The traditional version of the shell sort algorithm run on the same data set would require 5 total passes to get the data in numeric order, which is an additional two passes over this fuzzy algorithm version. The author projects from preliminary data that this technique would cut the number of passes in almost half as each pass pulls twice the amount of data.

Fuzzy Version of the Strand Sort

The strand sort is a fairly straightforward comparison-based sorting algorithm. The algorithm looks for strands of in-order data and slowly builds the sorted list [Black, 2008]. The following data points: 55, 57, 89, 92, 98, 95, 100, 3, 5, 7, 13, 15, 17, 24, 36, 42, 73, and 77 will be used to demonstrate the fuzzy strand sort (and the traditional strand sort). The traditional algorithm begins by creating a sub-list obtained by comparing each value to the next value and stopping

when the next is less than the previous. The traditional algorithm would find the following strand: 55, 57, 89, 92, 98. The remaining elements are: 95, 100, 3, 5, 7, 13, 15, 17, 24, 36, 42, 73, and 77. However, the fuzzy version of the strand sort performs fuzzy "less than" comparisons (e.g., $<_f$ means $x <_f y$ if $x < y + 5$) and, thus, could create the following sub-list: 55, 57, 89, 92, 98, 95, 100 with the remaining elements being: 3, 5, 7, 13, 15, 17, 24, 36, 42, 73, and 77. The traditional algorithm would need to take an additional step to get to the same place (merging the 95, 100 into the sub-list). From that point onward both the traditional and fuzzy versions of strand sort would produce the same strands. The final sorted list for the traditional strand sort is always precisely in-order. However, the fuzzy strand sort could have a few data points slightly out of order (as can be seen with the 98 and 95 above). The fuzzy strand algorithm required fewer overall steps and could be the better sort algorithm when the slightly out of order data are within an acceptable level of precision. Given the nature and organization of the data that works well for the strand sort, the fuzzy strand algorithm could take the same or fewer steps than the traditional version of the algorithm. However, the fuzzy strand algorithm would never take more steps then the traditional version. The fuzzy strand algorithm output might be slightly out of order, but the max amount is predictable and controlled by the specific implementation of the fuzzy set size.

Fuzzy Version of the Bucket Sort

A fuzzy sort most closely resembles the traditional distribution based sorts or "bucket" sorts, but differs because of the underlying definition of a fuzzy value. In a traditional bucket sort, an item sorts into one category/bucket [Sedgewick, 1998; Weiss, 1999]. In a fuzzy sort since each (fuzzy) value is represented by a set, an item can sort into multiple categories/buckets. Additionally, each category/bucket is a (fuzzy) value and it too is represented by a set, thus, the categories/buckets may overlap and do not have to be distributed evenly.

Given the following set of numbers: 24, 36, 50, 42, 57, 0, 88, 73, 92, 46, 7, 81, 15, 21, 100. Both the traditional and fuzzy bucket sort algorithms begin by sorting each number into a bucket/bin. For this example, consider that there are four bucket/bins for the traditional bucket sort: (1) 0-25, (2) 26-50, (3) 51-75, (4) 75-100. The numbers will be sorted as follows using the traditional algorithm:

1. 24, 0, 7, 15, 21
2. 36, 50, 42, 46
3. 57
4. 88, 73, 92, 81, 100

For the fuzzy algorithm, the four fuzzy buckets/bins are defined as: (1) 25 and below, (2) 20-50, (3) 45-82, and (4) 70 and above. The numbers could be sorted as follows using the fuzzy algorithm (and fuzzy buckets):

1. 0, 7, 15, 21
2. 24, 36, 50
3. 42, 57, 46, 73
4. 88, 81, 92, 100

Using the same four fuzzy buckets/bins, the fuzzy algorithm could also have produced the following:

1. 0, 7, 15, 21
2. 24, 36, 42, 46
3. 50, 57, 73, 81
4. 88, 92, 100

Observe that the output for the traditional algorithm would require further processing on each bin to complete the algorithm. Generally, in these kinds of algorithms, either each bin/bucket is subdivided into more bin/buckets or the contents of each bucket are sorted using a basic sort algorithm. In the first output of the fuzzy algorithm, the data in each bin/bucket except number four is almost in-order (and bin number four needs the 88 and 81 flipped). (Merging these bins/buckets into a final sorted list would be an easy process and eliminate the issue that the 46 is in bin number three while the 50 is located in bin number two). The second output of the fuzzy algorithm, the fuzzily sorted data is completely in order and does not require further processing. Both outputs of the fuzzy algorithm are better in-order than the output produced by the traditional algorithm

In this example, it is important to note that that there is both overlap between the different buckets and different sizes for each bucket/bin. Changing the overlap amounts and size of bins could change the outcome of the algorithm and it would be important to choose these dependent on the data being sorted. In other words, the bin size and overlap are both problem and data specific. If the user wants/needs a high level of precision on the fuzzy sort, then the overlap and bin size would need to be chosen carefully and the algorithm would probably need to continue by further sorting each bin. However, if each bucket/bin has a small quantity of data, in many cases it will not be necessary to continue sorting.

The total number of steps taken by the fuzzy bucket/bin sort is controlled in the same way as the traditional algorithm – basically divide into to bucket/bins until each bin has a certain amount (and then use another sorting technique to complete each bin). With the fuzzy bucket sort, it would be possible to eliminate the final step of using another sorting technique to completely sort each bin especially when the slightly out of order numbers are within an acceptable lack of precision. The output of a fuzzy bucket/bin algorithm might be slightly out of order, but can be partial controlled by the specific implementation of the fuzzy bucket/bin sizes and the overlap allowed among bins. The

fuzzy bucket/bin algorithm is also more adaptable and, from the preliminary data, more accurate when sorting data with fewer bins than needed by the traditional algorithm.

Possible Applications for Fuzzy Sort Algorithms

As mentioned earlier, sorting is most commonly used to manipulate data especially for display to a user or within a search algorithm, but sorting algorithms are found within a diverse range of interesting and complex algorithms. There are several operation research examples that implement a sort algorithm as an essential element of a larger algorithm including shortest processing time rule, load-balancing problem, and event-driven simulation [Hillier and Lieberman, 1990; Lasden, 1970; Luenberger, 1973]. In discrete event simulation, a sorting algorithm is used to order events but a fuzzy sort algorithm would be an unworkable solution because the events are required to be in exactly time order (and, as seen earlier, a fuzzy sort could have a few items out of order). However, there are circumstances where that functionality of the fuzzy sort algorithm would not have detrimental or negative effects on operation research algorithms. For example, in queuing problems, sorting is used to determine the ordering of operations and the operations list is constantly being re-ordered. Given that a precise ordering is not necessary and the nearly continuously resorting of the data, a faster fuzzy sort algorithm would be helpful to the overall runtime of the algorithm. Another example is bin packing, if objects are sorted from largest to smallest, then greedy algorithms can find optimal results. Using a fuzzy sort algorithm to produce a "good enough" or "close enough" list from largest to smallest will produce close to optimal results but spend less precious processing time. For queues, sequence rules, and other similar algorithms, a fuzzy sort can provide an approximate order which would be sufficient for most heuristic algorithms.

Discussion and Concluding Remarks

After preliminary analysis, further study is needed to investigate the promising implications seen in the aforementioned sample fuzzy sort algorithms. This paper has demonstrated that it is possible to convert a sort algorithm into a more abstract fuzzy version using the established three-step framework. However, it is important to note that often the success of the conversion process is dependent on the techniques and underlying design of the algorithm itself. Likewise, the measure of usefulness of the fuzzy sort algorithm is dependent on the precision required in the resulting sorted output. The "in-place" sort algorithms demonstrated some reduction in number of iterations needed by moving to a fuzzy sort algorithms. However, the bucket-based sort algorithm behavior corresponded the best to the natural habits of a fuzzy sort algorithm and, thus, showed the best promise of the algorithms presented in this paper.

Future Work

Using the knowledge gained in this proof-of-concept paper, the author plans to further investigate the fuzzification of sort algorithms by focusing attention on the bucket-like sorts and the "in-place" sorts. One next step would be to complete an implementation of these algorithms (both fuzzy and non-fuzzy) and run benchmark data on these algorithms to determine if the promise shown above holds for various cases. Additionally, the author needs to: (1) verify that fuzzifying the sort algorithm does not have adverse effects to sorting the data and (2) determine the best and worst cases for the fuzzy sorts. For adverse effects, the author will thoroughly investigate the negative or dangerous effects fuzzification could have on an algorithm and determine if there exist any countermeasures. The author also plans to use the knowledge gleamed from this experience to further refine details of the three-step framework. This research encompasses the need to determine which algorithm characteristics imply that there is a smooth transition from non-fuzzy to fuzzy using the framework and which algorithm characteristics imply that a transition is infeasible or impractical.

References

[Black, 2008] Black, P. E. 2008. Strand Sort. In *Dictionary of Algorithms and Data Structures*. https://www.nist.gov/dads/HTML/strandSort.html [online].

[Coffman-Wolph, 2016] Coffman-Wolph, S. 2016. Fuzzy Algorithms: Applying Fuzzy Logic to the Golden Ratio Search to Find Solutions Faster. In the Proceedings of the 27th Modern Artificial Intelligence and Cognitive Science Conference (MAICS 2016).

[Coffman-Wolph and Coffman Jr, 2016] Coffman-Wolph, S and Coffman, Jr, P. 2016. Fuzzification Of Search Techniques for Linear and Nonlinear Optimization. Conference Presentation. In the Proceedings of the INFORMS Annual Meeting, Nashville, Tennessee.

[Coffman-Wolph, 2015] Coffman-Wolph, S. 2015. The Hunch Factor: Exploration into Using Fuzzy Logic to Model Intuition in Particle Swarm Optimization. In the Proceedings of the 26th Modern AI and Cognitive Science Conference (MAICS 2015).

[Coffman-Wolph and Coffman Jr, 2014] Coffman-Wolph, S and Coffman, Jr, P. 2014. Fuzzification of the Special Simplex Method for the Transportation Problem. Conference Presentation. In the Proceedings of the INFORMS Annual Meeting, San Francisco, California.

[Coffman-Wolph, 2013] Coffman-Wolph, S. 2013. Fuzzy Search Strategy Generation for Adversarial Systems using Fuzzy Process Particle Swarm Optimization, Fuzzy Patterns, and a Hunch Factor. Ph.D. diss., Department of Computer Science, Western Michigan University, Kalamazoo, MI.

[Coffman-Wolph and Kountanis, 2013a] Coffman-Wolph, S. and Kountanis, D. 2013a. Fuzzy Process Particle Swarm Optimization.

In the Proceedings of the 43rd Southeastern Conference on Combinatorics, Graph Theory, & Computing. Winnipeg: Utilitas Mathematica Pub. Inc.

[Coffman-Wolph and Kountanis, 2013b] Coffman-Wolph, S. and Kountanis, D. 2013b. A General Framework for the Fuzzification of Algorithms. In the Proceedings of the 4th Biennial Michigan Celebration of Women in Computing (MICWIC 2013).

[Coffman-Wolph and Kountanis, 2013c] Coffman-Wolph, S. and Kountanis, D. 2013c. Finding Strategies in Adversarial Situations. In the Proceedings of the 24th Modern AI and Cognitive Science Conference (MAICS 2013).

[Hillier and Lieberman, 1990] Hillier, F. S., and Lieberman, G. J. 1990. *Introduction to Operations Research*. McGraw-Hill.

[Lasden, 1970] Lasden, L. (1970) *Optimization Theory for Large Systems*. New York, NY: Macmillan Publishing Co, Inc.

[Luenberger, 1973] Luenberger, D. 1973. *Introduction to Linear and Nonlinear Programming*. Reading, MA: Addison-Wesley Publishing Company.

[Sedgewick, 1998] Sedgewick, R. 1998. *Algorithms in C++ Third Edition, Parts 1-4*. Boston: Addison-Wesley Publishing Company, Inc.

[Weiss, 1999] Weiss, M. A. 1999. *Data Structures & Algorithm-mAnalysis in C++*. Reading, MA: Addison Wesley Longman, Inc.

[Wismer and Chattergy, 1978] Wismer, D. and Chattergy, R. 1978. *Introduction to Nonlinear Optimization*. New York, NY: Elsevier North-Holland, Inc.

[Zadeh, 1965] Zadeh, L.A. 1965. Fuzzy Sets. *Information and Control* 8: 338-353.