Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 154 (2019) 130–136

Procedia Computer Science

www.elsevier.com/locate/procedia

8th International Congress of Information and Communication Technology, ICICT 2019

MinFinder: A New Approach in Sorting Algorithm

Md. Shohel Rana[a,d] , Md Altab Hossin[b], S M Hasan Mahmud[c,d,*], Hosney Jahan[e], A. K. M. Zaidi Satter[f], Touhid Bhuiyan[d]

[a]*School of Computing Sciences and Computer Engineering, The University of Southern Mississippi, Hattiesburg, MS 39406, United States*
[b]*Department of Management Science & Engineering, University of Electronic Science and Technology of China, 61173,China*
[c]*Department of Computer Science and Engineering, University of Electronic Science and Technology of China, 61173,China*
[d]*Department of Software Engineering, Daffodil International University, Dhaka, 1207, Bangladesh*
[e]*School of Computer Science, Sichuan University, 610065, China*
[f]*Department of Computer Science and Engineering, Daffodil International University, Dhaka, 1207, Bangladesh*

Abstract

Sorting a set of unsorted items is a task that happens in computer programming while a computer program has to follow a sequence of precise directions to accomplish that task. In order to find things quickly by making extreme values easy to see, sorting algorithm refers to specifying a technique to arrange the data in a particular order or format where maximum of communal orders is in arithmetic or lexicographical order. A lot of sorting algorithms has already been developed and these algorithms have enhanced the performance in the factors including time and space complexity, stability, correctness, definiteness, finiteness, effectiveness, etc. A new approach has been proposed in this paper in sorting algorithm called *MinFinder* to overcome some of the downsides and performs better compared to some conventional algorithms in terms of stability, computational time, complexity analysis.

Keywords: Sorting; MinFinder; Time and Space Complexity; Correctness; Definiteness; Finiteness; Effectiveness; Inplace; Stability; Loop Invariant;

1. Introduction

A technique that is used for rearranging a set of unordered items into a finite sequence or order, like alphabetical, lowest-to-highest value or longest-to-shortest distance is called sorting algorithm. In sorting algorithms, unordered

* Corresponding author. Tel.: +86-13086692052
E-mail address: hasan.swe@daffodilvarsity.edu.bd

1877-0509
10.1016/j.procs.2019.06.020

items are given as input and deliver ordered arrays or lists as output by performing precise actions on those items. The most used application of sorting algorithms includes forming or displaying items by their price on different e-commerce websites (e.g. amazon, eBay, etc.), defining the order of sites by alphanumeric order on a search engine results page [1]. The importance of sorting is to search data can be improved while data is kept in a sorted way. There are two types of sorting: (i) *internal sorting* where the number of items is small enough to fits into the main memory, and (ii) *external sorting* where the number of items is so large that some of them reside on external storage during the sort. In this paper, we consider several internal sorting algorithms (e.g. Bucket sort, Bubble sort, Insertion sort, Selection sort, Heap sort, Merge sort) [2] in conjunction with some of the optimized algorithms (e.g. Parallel Shell sort, Parallel Quicksort, Parallel and Multithreading Merge sort, etc.) [3]-[7].

In this paper, we propose a new approach for sorting a list of items in simple way (highest-to-lowest value) without using conventional swapping concept that would consume memory. We also try to reduce the computational time that uses only one looping control structure '*for loop'* in conjunction with branching control structure '*goto'* that causes the logic to jump to a specific place in the program to reuse. This proposed sorting algorithm will try to overcome some basic drawbacks of conventional sorting algorithms.

The paper is systematized as follows: section 2 gives an overview of sorting algorithms; section 3 describes related work; section 4 presents our proposed technique, including pseudo-code, flowchart, steps of procedure, programming language implementation; section 5 presents comparison analysis including performance metrics, tools and technology used; and section 6 gives conclusions and future work.

2. Overview of Sorting Algorithms

In order to sort any unsorted items in a particular order, computer researchers make sorting algorithms, no matter what the original order was, and no matter how long the list is. Searching turn into easier when items are sorted, but sorting takes a long time and can be tedious.

2.1. Types of Sorting Algorithms

Following are some of the types of sorting algorithms while developing a new algorithm for sorting task [8].

- **In-place Sorting**: The program does not require any extra space for comparison. (e.g. bubble sort)
- **Not-in-place Sorting**: Needs extra space more than or equal to the elements to sort. (e.g. merge sort)
- **Stable Sorting**: Does not alternate the sequence of similar item in which they appear after sorting the items.
- **Not Stable Sorting**: Alternates the sequence of similar item in which they appear after sorting the items.
- **Adaptive Sorting**: Takes advantage of already 'sorted' items, which means don't try to re-order them into sorted form while the items has already sorted.
- **Non-Adaptive Sorting**: Try to force every single item to be re-ordered by confirming their sortedness.

Some terms are generally devised while discussing sorting techniques:

- **Increasing order**: A sequence of values where every next element is greater than the previous. (e.g. *1, 2, 4, 5, 7, 9*).
- **Decreasing order**: A sequence of values where every next element is smaller or less than the previous. (e.g. *9, 7, 5, 4, 2, 1*).
- **Non-increasing order**: Occurs when the list of items contains duplicate values. A sequence of values where every next element is less than or equal to but not greater than any previous element. (e.g. *9, 7, 5, 2, 2, 1*).
- **Non-decreasing order**: Also occurs when the list of items contains duplicate values. A sequence of values where every next element is greater than or equal to but not less than the previous one. (e.g. *1, 2, 2, 5, 7, 9*).

2.2. Properties of Sorting Algorithms

Sorting is a process that can be implemented through several algorithms where any of these algorithms contains the following criteria [9].

- **Input**: The algorithm must have input values from a definite set.

- **Output**: The algorithm produces output values that are defined as the solution to the problem using input set.
- **Definiteness**: The steps to sort the unordered items is used in sorting algorithm must be defined precisely.
- **Correctness**: The algorithm must yield the correct output values for every finite set of inputs.
- **Finiteness**: The algorithm must yield the desired output after a predetermined numeral step.
- **Effectiveness**: The algorithm should accomplish each step exactly using a finite amount of time.
- **Generality**: It should have the applicability for all sorting related problems, not just for a specific set.

2.3. Standards for Selecting Sorting Algorithm

In order to select a sorting algorithm, consider the standards described in Table 1 that leads to take initial decision while performing sorting task [10].

Table 1. Standards for sorting algorithm

Criteria	Sorting algorithm
when less elements	*Insertion Sort*
When elements are mostly in sorted form	*Insertion Sort*
While considering worst-case scenarios	*Heap Sort*
While considering average-case scenarios	*Quick Sort*
When elements are from a dense universe	*Bucket Sort*
While writing code as simple as possible	*Insertion Sort*

3. Related Study

A study is essential in order to develop or make any sorting algorithm, because not all algorithm works efficiently for the same problem.

Darpan Shah and Kuntesh Jani [11] proposed an improved sorting technique *Dual-Sort Extraction Technique (DSET)* enhances the performance and efficiency of the algorithm that performs two-level sorting where in the first level, the largest number is moved at the end of the dataset and in the second level, the smallest number is moved at the start of the dataset. This procedure continues until the remaining unsorted dataset come into sorted form.

Smita Paira, et al. [12] proposed an iterative approach with two different concepts that lead to consume less space in the stack and achieved better performance for large data compared to the recursive Divide and Conquer sorting algorithms having a worst-case time complexity of $O(n)$.

Khaled Thabit and Afnan Bawazir [13] proposed a sorting algorithm *Min-Max Bidirectional Parallel Selection Sort (MMBPSS)* by using dynamic programming in order to reduce sorting time by increasing the amount of space and to eliminate unnecessary iterations and also advised another new algorithm *Min-Max Bidirectional Parallel Dynamic Selection Sort (MMBPDSS)* that can place two elements: minimum and maximum from two directions using *Dynamic Selection Sort* algorithm in each round in parallel reducing the number of loop required for sorting and saving almost 50% of computational time than classical selection sort.

Abdullah Sheneamer, et al. [14] proposed two techniques of sorting algorithm for natural number by using the array indexing procedure and inserting that number into the proper index of the array without performing any element comparisons and swapping where the first technique improves ArrayIndexed Sorting Algorithm by adding negative numbers and the second technique that refers to Two Arrays-Indexed Sorting Algorithm for Natural Numbers (TAISN). For large array size with same length of digits of input data, these two techniques achieved better performance than the existing sorting algorithms of the $O(n^2)$.

Aayush Agarwal, et al. [15] proposed a new approach by finding the minimum and maximum element from the array and place one the first and last position of the array respectively. Then they obtain new array by incrementing the array index from the first position and decrementing from the last position.

P. Sumathi and V. V. Karthikeyan [16] proposed a new approach that faster than selection sort, *Double Ended Selection Sort Algorithm (DESSA)*. The *DESSA* inserts an array of elements and sort these items in the same array (in-place) by finding maximum and minimum items and exchanges them with the last and first items respectively with decreasing the size of the array by two for next call.

4. *MinFinder* Sorting Algorithm

The proposed *MinFinder* sorting algorithm mainly finds the element whose value is smallest from the list or array and place it to the first position of the list or array by shifting elements one position to the right from the first position to the position of smallest element found. Then find the second smallest element and place to the second position using the same technique. This technique continues until all the unsorted elements place in the proper position of the array (see figure 1). The *MinFinder* sorting algorithm actually follow the in-place sorting mechanism where it sorts the elements within the same array without using extra memory or space and also the *MinFinder* algorithm is a stable sorting because it keeps elements with equal keys in the same relative order in the output as they appeared in the input. Figure 2 describes the flowchart of *MinFinder*.

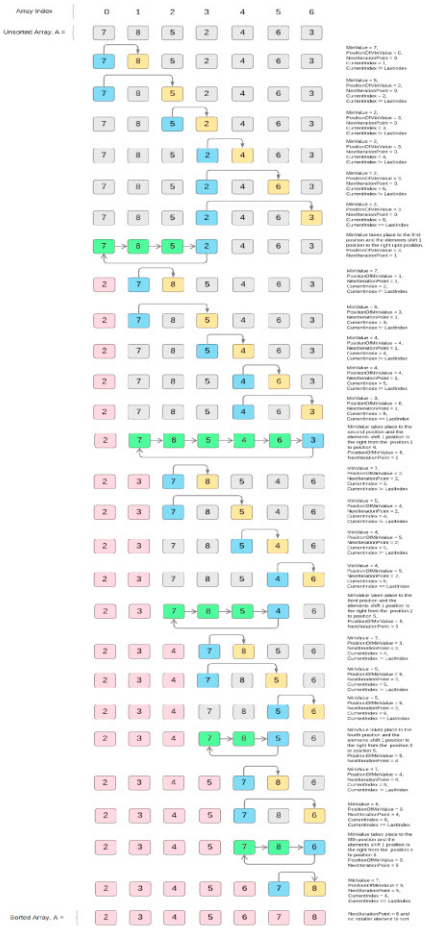

Fig. 1. Working procedure of *MinFinder* Algorithm

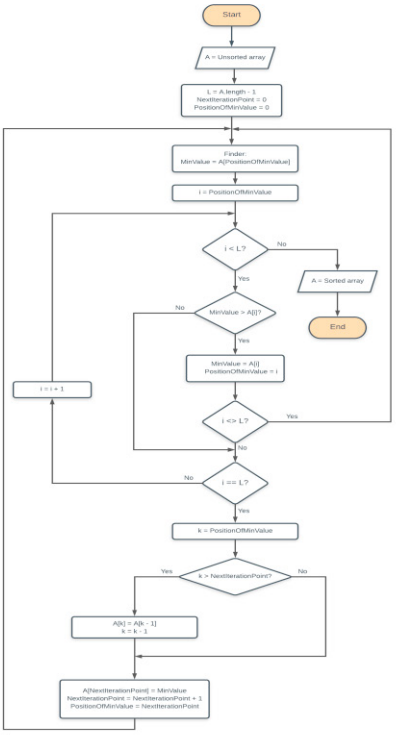

Fig. 2. Flowchart of *MinFinder* Algorithm

4.1. Procedure

The steps of working procedure of *MinFinder* Algorithm can be described as follows:

- **Step 1:** Initialize the variables *A[n], L = A.length() – 1, NextIterPoint = 0, PositionOfMinValue = 0*;
- **Step 2:** Assign a branching control statement *Finder:* for jumping to that specific place from anywhere and select the current element of the array as minimum value that is defined by *MinValue*;
- **Step 3:** Perform iteration until array index is smaller than or equal to the length of array, starting from the position of the current *MinValue;*

- **Step 4:** Check each element of the array using the selected *MinValue* whether the *MinValue* is greater or smaller to the current element. If the MinValue is greater than the current indexed element, then update *MinValue* with the current element and also update the position of *MinValue* to compare the rest of the array elements. *minValue = A[i]; positionOfMinValue = i*; Then check whether the current index is the last index of the array. If the current index is not the last element, jump to Step 2.
- **Step 5:** Check the current index is the last index of the array which make sure that the selected *MinValue* is compared with all the elements of the array. If true then shift array element one position to right from the first element to the position of the smallest element and the selected smallest value assign to the first position of the array. *A[k] = A[k - 1] where k = positionOfMinValue to IterationPoint*.
- **Step 6:** Update the next iteration point and position of min value and jump to Step 2 to repeat those steps until all the items sorted in the array.

4.2. Pseudo-Code for MinFinder

MinFinder(A):

```
1.  L = A.length – 1, NextIterPoint = 0,
    PositionOfMinValue = 0
2.  Finder:
3.  minValue = A[PositionOfMinValue]
4.  For i = PositionOfMinValue + 1 To L
5.      If minValue > A[i]
6.          minValue = A[i]
7.          PositionOfMinValue = i
8.          If i != L
9.              Go to Step 2
10.     If i = L
11.         For j = PositionOfMinValue to
    NextIterPoint
12.             A[j] = A[j-1]
13.         A[NextIterPoint] = minValue
14.         NextIterPoint++
15.         PositionOfMinValue =
    NextIterPoint
16.         Go to Step 2
17. Print(A)
```

Table 2. Complexity Analysis

Complexity			Name	Description
Best	**Avg.**	**Worst**		
$\Omega(1)$	$\Theta(1)$	$O(1)$	*Constant*	***This is the best**. Always takes the same amount of time, regardless of how much data there is.*
$\Omega(\log n)$	$\Theta(\log n)$	$O(\log n)$	*logarithmic*	***Pretty great.** Halve the amount of data with each iteration. If you have 100 items, it takes about 7 steps to find the answer.*
$\Omega(n)$	$\Theta(n)$	$O(n)$	*Linear*	***Good performance.** If the data contains 100 items, this does 100 units of work.*
$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$	*Linearithmic*	***Decent performance.** This is slightly worse than linear but not too bad.*
$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	*Quadratic*	***Kind a slow.** If the data contains 100 items, this does 100^2 = 10,000 units of work.*
$\Omega(n^3)$	$\Theta(n^3)$	$O(n^3)$	*Cubic*	***Poor performance.** If the data contains 100 items, this does 100^3 = 1,000,000 units of work.*
$\Omega(2^n)$	$\Theta(2^n)$	$O(2^n)$	*Exponential*	***Very poor performance.** Adding just one bit to the input doubles the running time.*
$\Omega(n!)$	$\Theta(n!)$	$O(n!)$	*Factorial*	***Intolerably slow.** It literally takes a million years to do anything.*

5. Results and Analysis

In order to sort any unsorted items in a particular order, computer researchers make sorting algorithms, no matter what the original order was, and no matter how long the list is. Searching turn into easier when items are sorted, but sorting takes a long time and can be tedious. Table 2 describes the performance based on the order of complexity [17]. This section shows some comparison study using several metrics includes time and space complexity, loop invariant, computational time, etc.

5.1. Time and Space Complexity

Let the number of elements in the array be *n*. In order to sort the first smallest element, the for loop iterates (*n-1*) times. For second smallest element, the for loop iterates (*n-2*) times, and so on for every case including *Best,*

Average and *Worst*. Hence, the overall time complexity is $\boldsymbol{O(n^2)}$ and the space complexity is ***O(1)*** because it takes constant time for every cases.

$= (n-1) + (n-2) + (n-3) + \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots + 1 = n(n+1)/2 - n = n(n-1)/2 = O(n^2)$

5.2. Loop Invariant

Loop Invariant is the statement about an algorithm that remains true or valid. Three things need to show about loop invariant for correctness of an algorithm [18]. In this section we try to define the loop invariant for showing or proving the correctness of the *MinFinder* algorithm.

- **Invariant:** The algorithm maintains the loop invariant that at the start of each for loop, *A[0, ..., i-1]* contains the elements originally in the *A[0, ..., i-1]* but is in sorted order.
- **Initialization:** Before the first iteration of the for loop *i = PositionOfMinValue*, the invariant trivially holds *A[0]* that is a sorted array.
- **Maintenance:** During the *i*-th loop iteration it finds smallest value and its corresponding position (e.g. *MinValue = A[i] = A[PositionOfMinValue]*) by comparing with rest of the elements. Then the inner for loop is used to shift *A[i-1], A[i-2], ..., A[IterationPoint]* from the position *IterationPoint* to the position *PositionOfMinValue.* Then the *MinValue* is placed in the position *IterationPoint* so that A[*IterationPoint*] ≤ A[*IterationPoint+1*] ≤ … ≤ *A[i-2]*. Thus *A[0, ..., i-1]* sorted + *A[i]* → *A[0, ..., i]* sorted.
- **Termination:** The loop terminates when *IterationPoint* is the last index of the array means no element is left to sort, then the invariant gives a useful information that, *A[0, ..., n-1]* contains of elements originally in *A[0, ..., n-1]*, but in sorted order.

5.3. Performance Results

This section describes the comparison among several sorting algorithms with our proposed *MinFinder* algorithm in terms of Time and Space complexity, stable and inplace sorting, execution time shown in Table 3.

Table 3. Performance based on Time and Space Complexity, execution time, stability and inplace sorting

Sorting Algorithm	Time Complexity			Space Complexity (Worst)	Execution Time based on Input Size (in milliseconds)				Stable?	Inplace?
	Best	Average	Worst		500	1000	5000	10000		
MinFinder	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$	0.204	0.738	16.846	66.472	Yes	Yes
Bubble	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$	0.215	0.507	17.485	67.425	Yes	Yes
Bucket	$\Omega(n)$	$\Theta(n)$	$O(n^2)$	$O(n)$	0.096	0.208	0.583	1.088	Yes	No
Selection	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$	0.468	1.858	45.605	180.123	No	Yes
Heap	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$	$O(1)$	0.036	0.067	0.437	1.02	No	Yes
Merge	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$	$O(n)$	0.108	0.273	1.223	2.354	Yes	No
Quick	$\Omega(n)$	$\Theta(n \log n)$	$O(n^2)$	$O(n)$	0.028	0.078	0.385	0.949	No	Yes
Radix	$\Omega(nk)$	$\Theta(nk)$	$O(nk)$	$O(n+k)$	0.031	0.061	0.3153	0.614	Yes	No
Insertion	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(n)$	0.113	0.466	11.569	45.689	Yes	Yes
Odd-Even	$\Omega(n \log n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$	0.217	0.844	17.489	85.035	Yes	Yes
Shell	$\Omega(n \log n)$	$\Theta(n (\log n)^2)$	$O(n (\log n)^2)$	$O(1)$	0.035	0.076	0.510	1.122	No	Yes
Tree	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n^2)$	$O(n)$	0.102	0.193	0.977	2.168	Yes	Yes

6. Conclusion and Future Works

Sorting technique mainly depends on various environments like low time complexity, less memory and simplicity. Yet, they have certain downsides, some of them lose their efficiency during handling the large data while others may set supplementary upstairs by considering the cost and memory management. In this paper we propose a new technique for sorting algorithm *MinFinder*, that is stable and inplace sorting by reducing the memory consumption in conjunction with solving the sorting problem without using conventional swapping technique. It has $O(n^2)$ time complexity and $O(1)$ space complexity. According to the computational time and complexity analysis, we observe that, the *MinFinder* takes less time than some popular conventional sorting algorithms including Selection sort, Bubble sort, Odd-Even sort. This is proved by analytical and experimental point of view.

In the future, we will try to improve and optimize our proposed algorithm by finding out the simple ways so that it can be applicable in various practical and real-life applications. We will also try to outspread our concepts to devise more algorithms which will be supportive for sorting action as well as software technologies.

References

1. Margaret R. sorting algorithm. last accessed: 2018/10/12, link: https://whatis.techtarget.com/definition/sorting-algorithm.
2. Sorting. last accessed: 2018/10/12 link: https://www.cs.cmu.edu/~adamchik/15-121/lectures/Sorting%20Algorithms/sorting.html.
3. Baddar SA and Mahafzah B. Bitonic sort on a chained-cubic tree interconnection network. Journal of Parallel and Distributed Computing, *2014*, pp. 1744-1761.
4. Mahafzah B. Performance assessment of multithreaded quicksort algorithm on simultaneous multithreaded architecture. Journal of Supercomputing, 66(1), 2013, pp. 339-363.
5. Mokahal F. A Parallel Shell Sort Algorithm for Chained-Cubic Tree Interconnection Network. M.S. Thesis, Department of Computer Science, The University of Jordan, 2017.
6. Tayeh RA. A Parallel Quicksort Algorithm on Optical Chained-Cubic Tree Interconnection Network. M.S. Thesis, Department of Computer Science, The University of Jordan, 2017.
7. Nimer A. A Parallel Merge Sort Algorithm for Chained-Cubic Tree Interconnection Network. M.S. Thesis, Department of Computer Science, The University of Jordan, 2016.
8. Data Structure - Sorting Techniques. last accessed: 2018/10/12, link: https://www.tutorialspoint.com/data-structures-algorithms/sorting-algorithms.htm.
9. Zaveri M. Algorithms I: Searching and Sorting algorithms. last accessed: 2018/10/12, link: https://codeburst.io/algorithms-i-searching-and-sorting-algorithms-56497dbaef20.
10. Pollice G, Selkow S and Heineman GT. Algorithms in a Nutshell. *O'Reilly Media, Inc*, ISBN: 9780596516246, October 2008.
11. Shah D and Jani K. A New Approach Toward Sorting Technique: Dual-Sort Extraction Technique (DSET). In: Mishra D., Nayak M., Joshi A. (eds) *Information and Communication Technology for Sustainable Development*. Lecture Notes in Networks and Systems, vol 10. Springer, Singapore, January 2018, pp. 219-227.
12. Paira S, Agarwal A, Alam S and Chandra S. Doubly Inserted Sort: A Partially Insertion Based Dual Scanned Sorting Algorithm. 2015, DOI: 10.1007/978-81-322-2550-8_2.
13. Thabit K and Bawazir A. A Novel Approach of Selection Sort Algorithm with Parallel Computing and Dynamic Programing Concepts. JKAU: Comp. IT, vol. 2, 2013, pp. 27-44, DOI: 10.4197/ Comp. 2-2.
14. Sheneamer A, Alharthi A and Hazazi H. Two Approaches of Natural Numbers Sorting: TAISN and Improved Array-Indexed Algorithms. International Journal of Computer Applications (0975 - 8887), vol. 121, No. 8, July 2015.
15. Agarwal A, Pardesi V and Agarwal A. A New Approach To Sorting: Min-Max Sorting Algorithm. International Journal of Engineering Research and Technology (IJERT), ISSN: 2278-0181, vol. 2, issue. 5, May 2013, pp. 445-448.
16. Sumathi P and Karthikeyan VV. A New Approach for Selection Sorting. International Journal of Advanced Research in Computer Engineering & Technology (IJARCET), ISSN: 2278-1323, vol. 2, issue. 10, October 2013, pp. 2720-2724.
17. Yerburgh E. A collection of sorting algorithms written in C. last accessed: 2018/10/14, link: https://github.com/eddyerburgh/c-sorting-algorithms.
18. Cormen TH, Leiserson CE, Rivest RL and Stein C. Introduction to Algorithms. 3rd Edition, The MIT Press, Cambridge, Massachusetts, London, England.