Tweaking for Better Algorithmic Implementation

Zirou Qiu, Ziping Liu, Xuesong Zhang
Computer Science Department
Southeast Missouri State University
Cape Girardeau, MO, U. S. A.
zqiu1s@semo.edu, zliu@semo.edu, xzhang@semo.edu

Abstract

Small changes in the implementations of a classic algorithm sometimes can improve its performance. In this paper, tweaking for better algorithmic implementation is discussed over heap sort algorithm and Hierholzer's algorithm. In heap sort algorithm, a new way to heapify is introduced, in which constant elements' swapping is avoided when elements are moved to their correct positions in the binary tree. In the new proposed implementation for Hierholzer's algorithm, searching the Euler circuit is started with the vertex of the highest indegrees (or outdegree) instead of a random pick. The paper proves that the performance of both algorithms is improved with the proposed tweaked implementations.

Keywords: Max Heapify, Heap, Euler circuit, Hierholzer's algorithm, busiest node

1 Introduction

Algorithms play an important role in computing. How to enhance the performance of a classic algorithm is always actively discussed. In this paper, we will discuss our tweaked implementations for heap sort algorithm and Hierholzer's algorithm.

1.1 Heap Sort

A sorting algorithm rearranges elements in a specific order. Different sorting algorithms have different running time, which could be divided into three categories: O(n), $O(n*\log_2(n))$ and $O(n^2)$ [5] (Journal). Sorting algorithms also differentiate each other by locations where a sorting occurs at or by stability. Merge Sort is an example of external, stable sorting algorithm, whereas heap sort is an internal, unstable sorting algorithm [7] (Journal). Due to its advantage on efficiency, heap sort is a widely used sorting algorithm.

In heap sort, binary heap is used, which is essentially a binary tree data structure in an array form [2] (book). There are two types of binary heap and they are distinguished by the heap properties (relationship between parents and children). In a max-heap, parent nodes are greater than or equal to their child nodes, whereas in a min-heap, parent nodes are less than or equal to their child nodes [2] (book). Figure 1 illustrates a max-heap. Heap sort is an algorithm which sorts elements in $O(n*\log_2(n))$ using a heap as internal structure. Heap sort has a better worst-case running time than quick sort, however not as efficient as quick sort on average [6] (web).

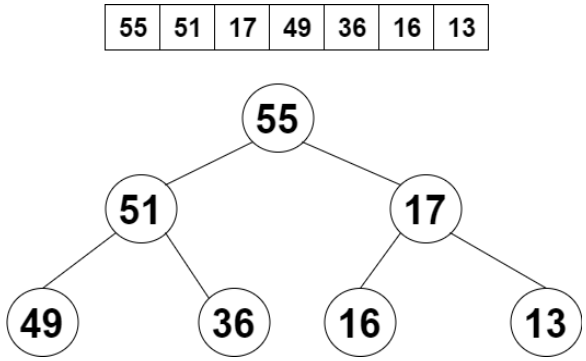

Figure 1: Max-Heap

Several enhancements have been discussed by researchers aiming to improve the execution time of heap sort. Ingo Wegener introduced a bottom-up approach of heap sort in 1993 [9] (Journal). Weak-heap sort is another variation of heap sort which used a new heap structure [3] (Journal). Later, D. Levendeas and C. Zaroliagis applied heap sort using multiply heaps where all heaps are stored in a single array [6] (web).

Heapify is the most important procedure in maintaining max/min heap property [8] (Journal). In this paper, we present a way of performing heapify such that no constant swapping is needed for moving disordered elements to the correct positions (partial algorithmic hibernation). In our discussion of heapifying, only the example of max-heapify procedure is studied in this paper, since min-heapify should be very similar.

1.2 Hierholzer's Algorithm

A graph consists of vertices and edges where two vertices are connected with a directed or undirected edge. A weight function could be applied to each edge. A graph is defined as G = (V, E), where V is the set of vertices in this graph and

978-1-943436-08-8 / copyright ISCA, CAINE 2017
October 2-4, 2017, San Diego, California, USA

E is the set of edges in this graph [2] (book). Every path in a graph is made up with a sequence of edges with two vertices at each end. If edges are undirected, paths are bidirectional. A cycle is a path which two vertices at each end are the same.

Given a directed graph G= (V, E), the graph contains a Euler circuit if G is a cycle and travels each edge exact once (vertices could be visited multiple times) [2] (book). Figure 2 illustrates a graph with Euler circuits. When we know for sure that G contains a Euler circuit, the current algorithm on finding that Euler circuit is Hierholzer's Algorithm [4] (book). It starts with a random vertex v, follows unvisited edges until comes back to v. Then the algorithm looks for next vertex v2 that has unvisited edges leaving it and performs the same procedure until all edges are visited. With the selection of random vertex as the starting node, the algorithm guarantees the search of Euler circuit. However, if a different approach is applied on the selection of a starting node, can the search of Euler circuit be sped up? In the following sections, we will discuss our proposed method and exploration on this topic.

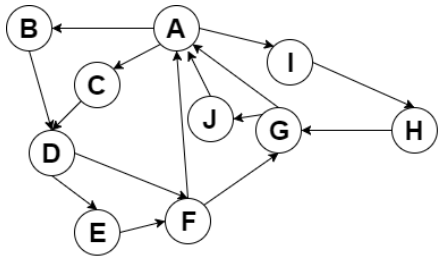

Figure 2: Euler Circuit

The remaining contents of this paper are organized as following. In section 2, the pseudocodes for both proposed implementations are discussed. Section 3 presents the analysis of the proposed implementations. Section 4 shows the execution time comparison for heap sort algorithm. Section 5 gives conclusion.

2 Tweaked Implementations

2.1 Enhanced Max-Heapify

In conventional heap sort, the disordered element is swapped with a child at each recursion call of max-heapify. To speed up the execution of the algorithm, we apply partial algorithmic hibernation as following: a copy of the disordered element is kept and it won't be assigned back to the heap until its correct location is found. Figure 3 illustrates the idea behind this implementation.

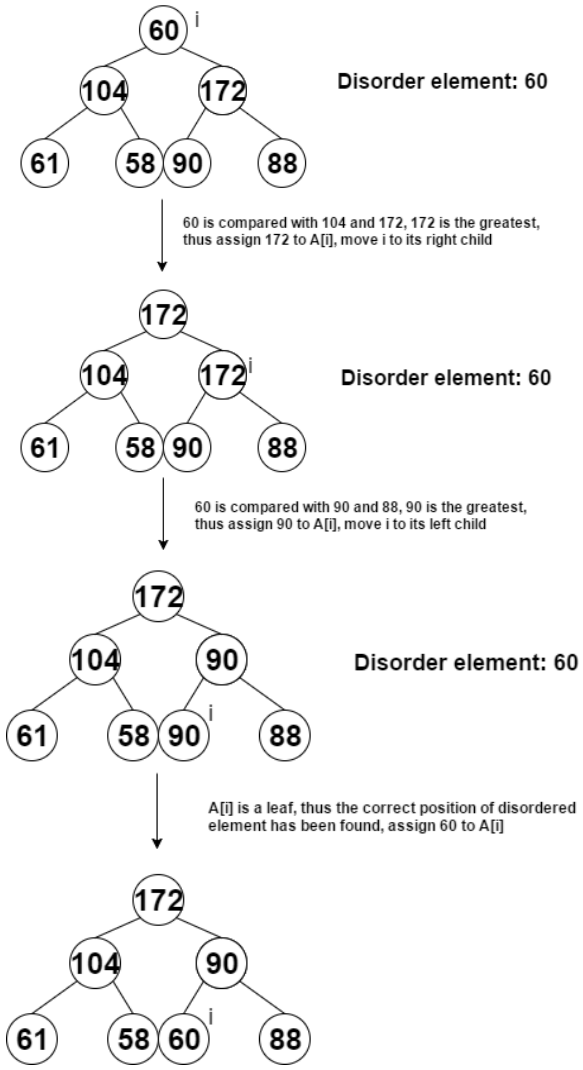

Figure 3: Proposed Implementation of Max-Heapify

Figure 3 demonstrates one case on which we can terminate the max-heapify procedure. Another case is when disordered element is greater than all the children of *A[i]*.

To demonstrate the difference between the conventional implementation and our proposed implementation, in the following two subsections, the pseudocodes for both the conventional implementation and our proposed implementation are presented. We start with pseudocode of max heapify procedure of the conventional heap sort from Thomas H. Cormen, and his colleges [2] (book). Then we present the pseudocode of our proposed implementation.

2.1.1 Max-Heapify of Conventional Heap Sort

COMMENT

A is a zero-based input array

END COMMENT

Max-Heapify (A, i)

```
01 left_index = 2 * i + 1
02 right_index = left_index + 1
03 IF left_index <= A.heap-size - 1 and A[left_index] > A[i]
DO:
04        largest_index = left_index
05 END IF
06 ELSE DO
07        largest_index = i
08 END ELSE
09 IF right_index <= A.heap-size - 1 and A[right_index] >
A[largest_index]  DO:
10        largest_index = right_index
11 END IF
12 IF largest_index != i DO:
13        swap A[i] with A[largest_index]
14        Max-Heapify(A, largest)
15 END IF
```

2.1.2 Max-Heapify of Proposed Algorithm

Max-Heapify(A, i)

```
01 target_value = A[i]
02 DO:
03        left_index = 2 * i + 1
04        right_index = left_index + 1
05        largest_index = i
06        IF left_index <= heap_size - 1 && A[left_index]
> target_value DO:
07                largest_index = left_index
08        END IF
09        IF right_index <= heap_size – 1 && A[right] >
A[left] && A[right] > target_value DO:
10                largest_index = right_index
11         END IF
12        IF largest_index != i DO:
13                A[i] = A[largest_index]
14                 i = largest_index
15        END IF
16        ELSE DO:
17                A[i] = target_value
18        END ELSE
18 WHILE A[i] != target_value;
```

Line 01 keeps a copy of disordered element. Lines 03 to 04 compute the index of right child (*right_index*) and left child (*left_index*) of *A[i]*. Lines 05 assigns *largest_index* to *i*. Lines 06 to 11 find the largest value among *target_value*, *A[right_index]* and *A[left_index]* (if they are in the heap), then assign *largest_index* to the index of the largest value. Lines 12 to 15 checks if *largest_index* still equals i, if not, that means a *A[i]*'s child is greater than *target_value*. We assign the value of that child to *A[i]* and set *i* equal to *largest_index*. As line 16 to 18 suggest, if largest_index still equals to i, that means the correct position has been found. Simply assign *target_value* to *A[i]*, which terminates the while loop.

2.2 Enhanced Hierholzer's Algorithm Implementation

In the conventional Hierholzer's algorithm, Euler circuit search starts with a random vertex v, follows unvisited edges until comes back to v. Then the algorithm looks for next vertex v2 that has unvisited edges leaving it and perform the same procedure until all edges are visited. With the selection of random vertex as the starting node, the algorithm guarantees the search of Euler circuit. However, the algorithm may take longer to find an Euler path with the brute-force selection of the starting vertex. Hence, we propose the following alternation in the implementation: during the creation of nodes, each vertex is assigned with an attribute *d* which represents the in-degree of the node. Then the algorithm chooses the node with the highest *d* value as the starting vertex rather than picks a starting vertex randomly. When an edge is visited, the attribute *d* of nodes on this edge is decrement by one and this edge is removed from the graph. All nodes are stored in a priority queue implemented by Fibonacci heap, keyed by *d* value. Furthermore, instead of finding the next vertex that has unvisited edge leaving it, we could simply pop the next element with the highest *d* value in the priority queue. The graph is implemented by adjacent list.

To demonstrate the difference between the conventional implementation and our proposed implementation, the pseudocodes for the conventional implementation is presented in subsection 2.2.1, and our proposed implementation is presented in subsection 2.2.2. And in subsection 2.2.3 we present our proposed implementation with an example as shown in Figure 4.

2.2.1 Pseudocode of Conventional Hierholzer's Algorithm

COMMENT

The precondition is that the graph is guaranteed to have at least one Euler Circuit

END COMMENT

Hierholzer (G)

```
01 Let euler_circuit be an empty array
02 FOR each u in G.V DO:
03        Let current_circuit be an empty array
```

```
04      IF u.adj_list[] is not empty DO:
05              Append u into current_circuit
06      END IF
07      WHILE u.adj_list[] is not empty DO:
08              Let v be the next vertex in u.adj_list[]
09              Append v into current_circuit
10              Remove v from u.adj_list
11              u = v
12      END WHILE
13      Substitute current_circuit into euler_circuit
14 END FOR
15 Return euler_circuit
```

2.2.2 Pseudocode of Enhanced Hierholzer's Algorithm

COMMENT

d attribute of each vertex is initialized during the creation of the graph

END COMMENT

Enhanced-Hierholzer (G)

```
01 Let Q be a priority queue keyed on d attribute of nodes
02 Let euler_circuit be an empty array
03 FOR each vertex v in G.V DO:
04      Q.push(v)
05 END FOR
06 WHILE Q.top().d != 0 DO:
07      u = Q.top()
08      let current_circuit be an empty array
09      WHILE u.d != 0 DO:
10              append u to current_circuit
11              Let v1 be the next vertex in u.adj_list[]
12              Q.Decrease-Key(u, u.d-1)
13              remove v1 from u.adj[]
14              u = v1
15      END WHILE
16      substitute current_circuit into euler_circuit
17 END WHILE
18 return euler_circuit
```

Line 01 creates the priority queue *Q*. Line 02 creates the array *euler_circuit* which holds the Euler Circuit. Lines 03 – 05 push each vertex into *Q*. Lines 06 – 17 is the main while loop which terminates when the d value of the first node in *Q* is 0. This case also implies that all edges have been visited and Euler path has been found. Inside this while loop, line 07 assign the node with the greatest d value to *u*. Line 08 creates an empty array *current_circuit*. *current_circuit* holds vertices constitute the new circuit. Lines 09 – 15 presents the inner while loop, which terminates when all edges leaving *u* have been visited. As lines 10 – 14 suggest, *u* will be copied to *current_circuit*, then *d* value of *u* will be decrement by 1. The next node v1 in the adjacency list of *u* is removed from the list and u is assigned to v1. Line 16 substitutes *current_circuit* into *euler_circuit* for each iteration of the first while loop. At last, line 18 returns *euler_circuit*.

2.2.3 Pictorial Representation of the Proposed Implementation

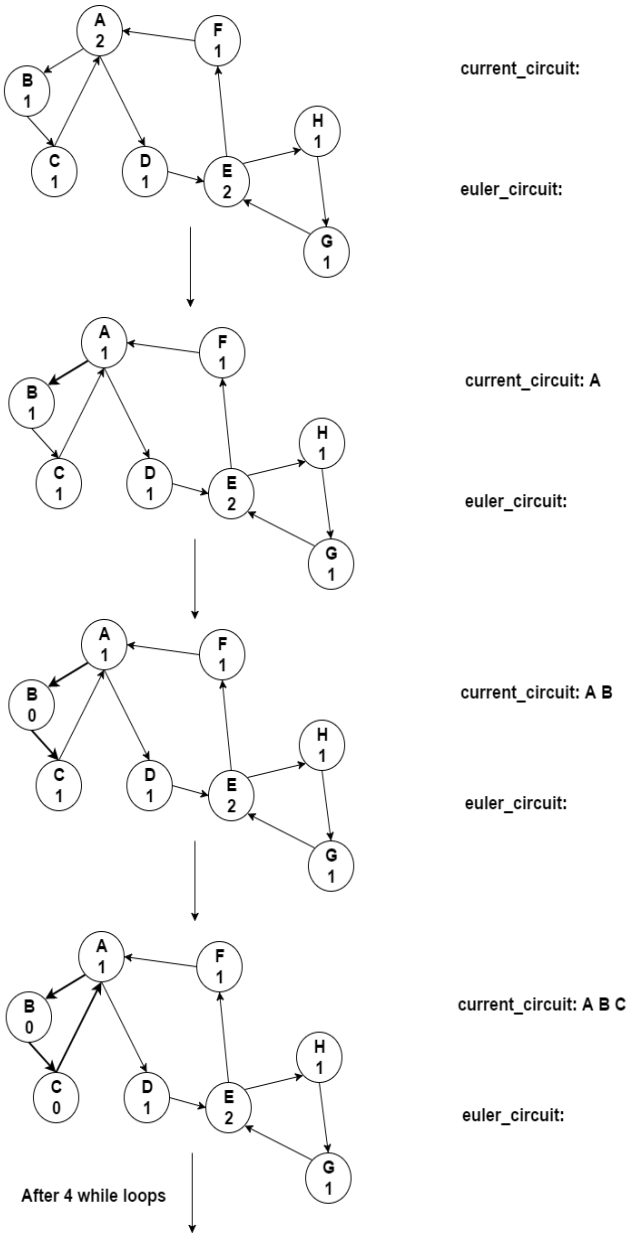

Figure 4: Example of Proposed Implementation for Hierholzer's Algorithm

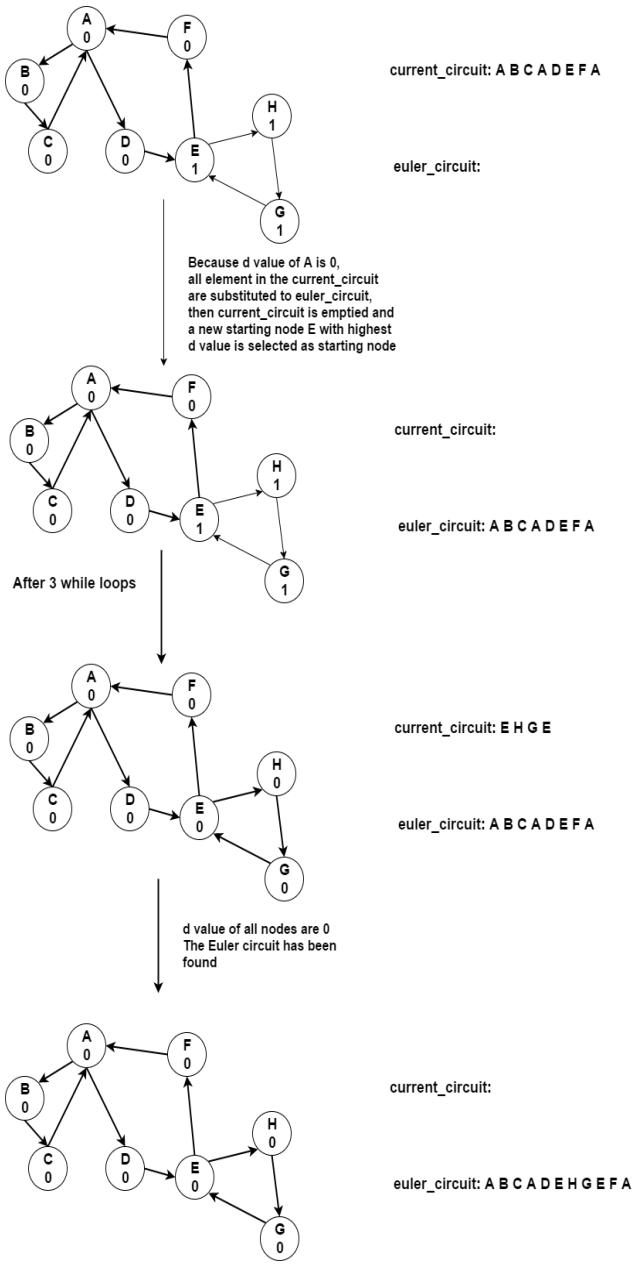

Figure 4: Example of Proposed Implementation for Hierholzer's Algorithm Continue

3 Analysis of the Proposed Implementations

3.1 Max-Heapify

For the sake of simplicity, let the size of input array n equals 2^{h+1}-1 which makes the tree complete. In this case, h is the height of the binary tree where h = $O(\log_2(n))$. Only assignment operations of elements in the heap is counted to for the comparison between proposed algorithm and conventional algorithm.

3.1.1 Conventional Max-Heapify

In the worst case, there are three assignment operations (one swap) being called at each level of the heap. The total number of assignment operation of moving one disordered element to the correct position in the worst case is 3*h.

There are constant number of assignment operations and comparisons called for each invocation of max-heapify, thus the running time of conventional max-heapify is O(h) = $O(\log_2(n))$.

3.1.2 Enhanced Max-Heapify

In the worst case, there is one assignment operations being called at each level of the heap, thus, the total number of assignment operation of moving one disordered element to the correct position in the worst case is h, which is significantly reduced when compared with the conventional implementation's 3*h of assignments.

Like the conventional max-heapify, there are constant number of assignment operations and comparisons called for going down one level in the heap, thus, the running time of the proposed algorithm is O(h) = $O(\log_2(n))$.

3.2 Enhanced Implementation for Hierholzer's Algorithm

Based on the pseudocode given in section 2.2.2, pushing all vertices into the priority queue takes O(V) times. The inner while loop is called |E| times. Because Fibonacci heap is used to implement the priority queue, in a dense graph, Decrease-Key operation takes O(1) amortized time. Operations like appending nodes to the array, remove nodes from adjacency list and assignment operation takes constant time. Thus, calling the while loop takes O(E) times. The overall running time of enhanced hierholzer's algorithm implementation is O(V + E).

4 Execution Time Comparison for Heap Sort

In this section, we present our execution time comparison results on the experimental data for both the conventional implementation as well as the tweaked implementation for heap sort algorithm. Source code was written in C++11 and tested on windows 8.1, with Intel Core i5 processor and 8GB of RAM. Each input array consists numbers randomly generated using C++ random number engine with uniform integer distribution. Numbers in each input array range from 0 to the size of the array.

Table. 1 shows the execution time in seconds of the conventional implementation on Heap Sort and the

proposed implementation on Heap Sort. From the result of the experiment data, the execution-time superiority of the proposed implementation over the conventional implementation is more obvious with the increment of the input size. The proposed implementation is 20.85% faster than the conventional implementation at the input size of 40,000,000.

Table 1: Execution time in seconds of the Conventional and the Proposed Implementations for Heap Sort

Input Size	Heap Sort	Proposed Algorithm
10,000	0.010027	0.008023
100,000	0.064046	0.051037
1,000,000	0.828587	0.659468
5,000,000	5.36782	4.28906
10,000,000	11.9155	9.60184
20,000,000	26.686	21.121
25,000,000	34.7668	27.5416
30,000,000	42.1981	33.4308
40,000,000	58.5807	46.369

5 Conclusion

This paper discusses the tweaking for better algorithmic implementation over heap sort algorithm and Hierholzer's algorithm. For heap sort, both the algorithmic analysis and the experimental data result demonstrate that the enhanced max-heapfy implementation successfully sorts elements in ascending order with lower execution time than the conventional implementation. Constant swapping elements is avoided in the proposed max-heapify procedure. The proposed implementation has running time of $O(n*\log_2(n))$, however it performs less assignment operations than conventional implementation. For the Hierholzer's Algorithm, the new proposed implementation uses a priority queue to store all vertices which make the process of searching next starting element easier. Starting at the vertex with the higher in-degree avoids the case where the program needs to look for next starting constantly.

Reference

[1] Nidhi Chhajed, Imran Uddin and Simarjeet Singh Bhatia. A Comparison Based Analysis of Four Different Types of Sorting Algorithms in Data Structures with Their Performances. *International Journal of Advanced Research in Computer Science and Software Engineering*, 3(2): 373 – 380, 2013.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. *Introduction to Algorithms.* The MIT Press, third edition, 2009.

[3] Stefan Edelkamp and Ingo Wegener. On the Peformance of WEAK-HEAPSORT. *Annual Symposium on Theoretical Aspects of Computer Science*, 254-266.

[4] D. Jungnickel. Graphs, *Networks and Algorithms*, *Algorithms and Computation in Mathematics*. Springer-Verlag Berlin Heidelberg 2013.

[5] Gaurav Kocher and Nikita Agrawal. Analysis and Review of Sorting Algorithms. *International Journal of Scientific Engineering and Research*, 2(3): 81- 84, 2014.

[6] D. Levendeas and C. Zaroliagis. Heapsort using Multiple Heaps. http://students.ceid.upatras.gr/~lebenteas/Heapsort-using-Multiple-Heaps-final.pdf

[7] Vandana Sharma, Satwinder Singh and K. S. Kahlon. Performance Study of Improved Heap Sort Algorithm and Other Sorting Algorithms on Different Platforms. *International Journal of Computer Science and Network Security*, 8(4): 101- 103, 2008.

[8] Reyha Verma and Jasbir Singh. A Comparative Analysis of Deterministic Sorting Algorithms based on Runtime and Count of Various Operations. *International Journal of Advanced Computer Research*, 5(21): 380 – 384, 2015.

[9] Ingo Wegener. BOTTOM-UP-HEAPSORT, a new variant of HEAPSORT beating, on an average, QUICKSORT (if n is not very small). *Theoretical Computer Science*, 118: 81-98, 1993.