IPSJ Journal Vol. 51 No. 9 1832–1846 (Sep. 2010)

Regular Paper

Entropy as Computational Complexity[⋆1]

Tadao Takaoka[†1] and Yuji Nakagawa[†2]

If the given problem instance is partially solved, we want to minimize our effort to solve the problem using that information. In this paper we introduce the measure of entropy, $H(S)$, for uncertainty in partially solved input data $S(X) = (X_1, \ldots, X_k)$, where X is the entire data set, and each X_i is already solved. We propose a generic algorithm that merges X_i's repeatedly, and finishes when k becomes 1. We use the entropy measure to analyze three example problems, sorting, shortest paths and minimum spanning trees. For sorting X_i is an ascending run, and for minimum spanning trees, X_i is interpreted as a partially obtained minimum spanning tree for a subgraph. For shortest paths, X_i is an acyclic part in the given graph. When k is small, the graph can be regarded as nearly acyclic. The entropy measure, $H(S)$, is defined by regarding $p_i = |X_i|/|X|$ as a probability measure, that is, $H(S) = -n(p_1 \log p_1 + \ldots + p_k \log p_k)$, where $n = |X_1| + \ldots + |X_k|$. We show that we can sort the input data $S(X)$ in $O(H(S))$ time, and that we can complete the minimum cost spanning tree in $O(m + H(S))$ time, where m in the number of edges. Then we solve the shortest path problem in $O(m+H(S))$ time. Finally we define dual entropy on the partitioning process, whereby we give the time bounds on a generic quicksort and the shortest path problem for another kind of nearly acyclic graphs.

1. Introduction

The concept of entropy is successfully used in information and communication theory. In algorithm research, the idea is used explicitly or implicitly. In Ref. 1), entropy is explicitly used to navigate the computation of the knapsack problem. On the other hand, entropy is used implicitly to analyze the computing time of various adaptive sorting algorithms [2]. In this paper, we develop a more unified approach to the analysis of algorithms using the concept of entropy. We regard the entropy measure as the uncertainty of the input data of the given problem instance, that is, the computational difficulty of the given problem instance.

†1 Department of Computer Science and Software Engineering University of Canterbury
†2 School of Informatics Kansai University

First let us describe the framework of amortized analysis. Let S_0, S_1, …, S_N be the states of data such that S_0 is the initial state and S_N is the final state. The computation in this paper is to transform S_{i-1} to S_i at the i-th step for $i = 1, \cdots, N$. The potential of state S is denoted by $\Phi(S)$, which describes some positive aspect of data. That is, increasing the potential will ease the computation at later steps. The actual time and amortized time for the i-th step are denoted by t_i and a_i. We use the words "time" and "cost" interchangeably. The amortized time is defined by the accounting equation

$$a_i = t_i - \Delta\Phi(S_i), \quad (1)$$

where $\Delta\Phi(S_i) = \Phi(S_i) - \Phi(S_{i-1})$. That is, the amortized time is the actual time minus the increase of potential at the i-th step. By summing up the equation over i, we have

$$\Sigma a_i = \Sigma t_i + \Phi(S_0) - \Phi(S_N), \text{ or } \Sigma t_i = \Sigma a_i - \Phi(S_0) + \Phi(S_N).$$

The purpose of amortized analysis is to simplify the analysis process. That is, we can focus on one step of accounting equation, proving a_i is bounded by some value. We do amortized analysis in algorithms and data structures separately.

The rest of the paper consists of the following sections. In Section 2, we define the entropy on a list of partially solved data sets, and give a generic algorithm that merges two sets repeatedly. The merging process is regarded as decreasing the entropy of the list. In Section 3, we analyze the minimal mergesort that finishes sorting of partially sorted list with minimum effort, which is given by entropy. Also minimal mergesort is proven to be asymptotically optimal under the entropy measure. In Section 4, an array-based algorithm for merging is used in minimal mergesort, instead of tree-based one in the previous section. Section 5 is devoted to the analysis of the remaining work of Kruskal's algorithm for the minimum cost spanning tree problem. In Section 6, we review the 2-3 heap and define a delete operation, which will be used in the following section of the shortest path problem. Section 7 describes how to use the entropy measure for the single source shortest path problem on a nearly acyclic directed graph. Roughly speaking, a nearly acyclic graph is composed of a small number of

⋆1 Preliminary versions of this paper appeared at CATS 1997 [2] and MFCS 2009 [3]. Part of this research was done while the first author was on leave at Kansai University

acyclic graphs connected to each other by global links. The analysis is done here directly, except for using amortized analysis of data structures, as amortized analysis does not result in simplification. The partial solution cannot be defined before the algorithm starts, that is, only defined dynamically by the algorithm itself. Section 8 addresses this issue, and offers some upper bound on the computing time using the entropy derived from some structural property of the graph, although the bound is not as sharp as the bound in Section 7. Section 9 defines the concept of dual entropy, based on a generic algorithm based on partitioning process, rather than merging process. The result in this section is rather straightforward, but offers some viewpoint of the entropy analysis of this paper. Section 10 concludes the paper.

2. Generic Algorithm and Entropy

In this section we propose a generic algorithm based on merging and its analysis. Those results will be applied to a few examples in the later sections in their original forms or with minor modifications.

Let the state of data be given by a decomposition of a set X, called the list of partially solved sets, as $S(X) = (X_1, \ldots, X_k)$. In the following algorithm, we remove $W_1 = X_i$ and $W_2 = X_j$ for some i and j from the list, merge them and put the merged set back to the list. Then we iterate this process. If the list becomes a list of one set, we finish. The operation "merge" here is the one in the general meaning of putting two sets together under some constraint. We regard each X_i as a solved set, meaning that the elements in it satisfy some constraints given by each problem, and $S(X)$ as a partial solution. In case of sorting, X_i is a set of sorted items. In the shortest path problem, X_i can be a set of vertices such that the subgraph induced from it forms an acyclic graph, etc. In a later section we regard X_i as unsolved and the order in which X_i are placed in $S(X) = (X_1, \ldots, X_k)$ satisfies some constraint, suggesting $S(X)$ is a partial solution in a different definition.

The left arrow at lines 4 and 5 in Algorithm 1 means that some sets from the list M are chosen and removed, and assigned to W_1 and W_2. The left arrow at line 7 means the set is put into list M. The problem of from where we take the sets or to where we put the merged set will depend on the specific nature of the given problem. Variable k maintains the number of the sets in the list. We keep merging while $k > 1$.

Algorithm 1 (Generic algorithm based on merging)
1 Decompose X into $S(X) = \{X_1, \ldots, X_k\}$;
2 $M := S(X)$;
3 **while** $k > 1$ **do begin**
4 $W_1 \leftarrow M$;
5 $W_2 \leftarrow M$;
6 $W := \text{merge}\,(W_1, W_2)$;
7 $M \leftarrow W$;
8 $k := k - 1$;
9 **end**

To analyze this algorithm, we define the following concepts. Let $|X| = n$, $n_i = |X_i|$ and $p_i = n_i/n$. Note that $\sum p_i = 1$. We define the entropy of a decomposition of X, $H(S(X))$, abbreviated as $H(S)$, by

$$H(S) = -n \sum_{i=1}^{k} p_i \log p_i = \sum_{i=1}^{k} |X_i| \log(|X|/|X_i|) \tag{2}$$

The entropy is regarded as a negative aspect of the data, i.e., the less entropy, the closer to the solution. Normally entropy is defined without the factor of n, the size of the data set. We include this to deal with a dynamic situation where the size of the data set changes. Logarithm is taken with base 2 unless otherwise specified. Since p_i $(i = 1, \cdots, k)$ can be regarded as a probability measure, we have $0 \leq H(S) \leq n \log k$ and the maximum is obtained when $|X_i| = n/k$ $(i = 1, \cdots, k)$.

We capture the computational process as a process of decreasing the entropy in the given data set X. We assume $H(S_0) \geq H(S_1) \geq \ldots \geq H(S_N)$. For the potential in equation (1), we set $\Phi(S_i) = -cH(S_i)$ for some constant $c > 0$, which is determined for each application.

The accounting equation becomes $a_i = t_i - c\Delta H(S_i)$, where $\Delta H(S_i) = H(S_{i-1}) - H(S_i)$. That is, the amortized time is the actual time minus a constant factor the decrease of entropy at the i-th step. In Algorithm 1, the merging

process is supposed to decrease the entropy.

Let T and A be the actual total time and the amortized total time. Summing up a_i for $i = 1, \ldots, N$, we have $A = T + c(H(S_N) - H(S_0))$, or $T = A + c(H(S_0) - H(S_N))$. In the following we see three applications, where A can be easily obtained. In many applications, $H(S_N) = 0$, meaning that the total time is $A + O(H(S))$, where $H(S) = H(S_0)$ is the initial entropy.

Theorem 1 Let us assume $n_1 = |W_1|$ and $n_2 = |W_2|$ without loss of generality. If the actual time of the i-th merge at line 6 is bounded by $O(n_1 \log(1+n_2/n_1) + n_2 \log(1+n_1/n_2))$ and the final entropy is 0, the computing time of Algorithm 1 is $O(H(S))$.

Proof. Let $t_i \le c(n_1 \log(1 + n_2/n_1) + n_2 \log(1 + n_1/n_2))$. Since the change of entropy occurs only with n_1 and n_2, we have

$$\begin{aligned}\Delta H(S_i) &= n_1 \log(n/n_1) + n_2 \log(n/n_2) - (n_1 + n_2) \log(n/(n_1 + n_2)) \\ &= n_1 \log(1 + n_2/n_1) + n_2 \log(1 + n_1/n_2) \\ a_i &\le c(n_1 \log(1 + n_2/n_1) + n_2 \log(1 + n_1/n_2)) - c\Delta H(S_i) = 0\end{aligned}$$

Thus $A \le 0$ and we have the result. ■

Let $S'(X) = (X'_1, \ldots, X'_{k'})$ be a refinement of $S(X) = (X_1, \ldots, X_k)$, that is, $S'(X)$ is a decomposition of X and for any X'_i there is X_j such that $X'_i \subseteq X_j$. Then we have $H(S) \le H(S')$. As the entropy is a measure of uncertainty, we can say $S(X)$ is more solved than $S'(X)$.

In later sections, we show a few interpretations of X_i's. In sorting, X_i's are ascending runs which are regarded as solved. In the minimum spanning tree problem, X_i is the set of vertices of a partially solved minimum spanning tree for a subgraph. In shortest paths, X_i is an acyclic subgraph, for which shortest distances are easily computed.

The main point of the paper is to offer a new method for algorithm analysis rather than designing new algorithms.

3. Application to Adaptive Sort – Minimal Mergesort

Adaptive sorting is to sort the list of n numbers into increasing order as efficiently as possible by utilizing the structure of the list which reflects some presortedness. See Estivill-Castro and Wood [4] for a general survey on adaptive sorting. There are many measures of disorder or presortedness. The simplest one is the number of ascending runs in the list. Let the given list $X = (a_1, a_2, \cdots, a_n)$ be divided into k ascending runs X_i $(i = 1, \cdots, k)$, that is, $S(X) = (X_1, X_2, \cdots, X_k)$ where $X_i = (a_1^{(i)}, \cdots, a_{n_i}^{(i)})$ and $a_1^{(i)}$ is the $|X_1| + \cdots + |X_{i-1}| + 1$-th element in X. We denote the length of list X by $|X|$. $S(X)$ is abbreviated as S. Note that $a_1^{(i)} \le \cdots \le a_{n_i}^{(i)}$ for each X_i and $a_{n_i}^{(i)} > a_1^{(i+1)}$ if X_i is not the last list. The sort algorithm called natural merge sort [5] sorts X by merging two adjacent lists for each phase, halving the number of ascending runs after each phase so that sorting is completed in $O(n \log k)$ time. Mannila [6] proved that this method is optimal under the measure of the number of ascending runs.

In this paper we generalize the measure $RUNS(S)$ of the number of ascending runs into that of the entropy of ascending runs in X, denoted by $H(S)$. Then we analyze a sorting algorithm, called minimal merge sort, that sorts X by merging two minimal length runs successively until we have the sorted list. We show that the time for this algorithm is $O(H(S))$ and is asymptotically optimal under the measure of $H(S)$. Hence the measure $H(S)$ is sharper than $RUNS$ measure of $O(n \log k)$.

The idea of merging two shortest runs may be known. The algorithm style based on "meta-sort" in this section is due to Ref. 2). All lists are maintained in linked list structures in this section. Let $S(X) = (X_1, \cdots, X_k)$ be the given input list such that each X_i is sorted in ascending order. Re-arrange X into $S'(X) = (X_{i_1}, \cdots, X_{i_k})$ in such a way that $|X_{i_j}| \le |X_{i_{j+1}}|$ $(j = 1, \cdots, k-1)$, that is, $(X_1, \cdots, X_k)$ is sorted with $|X_i|$ as key. We call this "meta-sort." Since each $|X_{i_j}|$ is an integer up to n, we can obtain $S'(X)$ in $O(k)$ time by radix sort. Now we sort $S'(X)$ by merging two shortest lists repeatedly. Formally we have the following. Let M and L be lists of lists, whereas W_i $(i = 1, 2)$ and W are ordinary lists. By the operation $M \Leftarrow L$, the leftmost list in L is moved to the rightmost part of M. By the operation $W_i \Leftarrow M$ $(i = 1, 2)$ the leftmost list of M is moved to W_i. By the operation $M \Leftarrow W$, W is moved to the rightmost part of M. M is regarded as a "meta-queue", consisting of a mixture of original X_i's and merged lists, sorted by the key of length. $First(L)$ is the first list in L. We assume L is not empty.

ALGORITHM 2 (Minimal mergesort)

```
1  Meta-sort S(X) into S'(X) by length of X_i;
2  Let L = S'(X);
3  M := ∅;
4  M ⇐ L;
5  if L ≠ ∅ then M ⇐ L else W ⇐ M;
6  for i := 1 to k − 1 do begin
7     W_1 ⇐ M;
8     W_2 ⇐ M;
9     W := merge (W_1, W_2);
10    while L ≠ ∅ and |W| > |first(L)| do M ⇐ L;
11    M ⇐ W
12 end
```

In line 9, we use the tree-based merge algorithm of $O(m \log(1+n/m))$ time by Brown and Tarjan [7], that merges two sorted lists of length m and n in balanced trees such that $m \leq n$. We use the array-based algorithm of $O(m+n-1)$ time in the next section. We mainly measure the computing time by the number of key comparisons in the merge operation at line 9.

Lemma 1 The amortized time for the i-th merge is not greater than zero.

Proof. Lemma follows from the fact that
$t_i \leq cn_1 \log(1+n_2/n_1) = O(n_1 \log(1+n_2/n_1) + n_2 \log(1+n_1/n_2))$
$a_i \leq cn_1 \log(1+n_2/n_1) - c\Delta H(S_i) \leq 0$ ■

Theorem 2 The algorithm minimal mergesort sorts $S(X) = (X_1, \cdots, X_k)$ where each X_i is an ascending sequence in $O(H(S))$ time.

Proof. Theorem follows from Lemma 1, the initial entropy being $H(S)$ and the final one being 0. The complexity of radix sort is absorbed in $H(S)$. ■

The initial scan for ascending runs including transformation to balanced trees takes $O(n)$ time, meaning the time including scanning becomes $O(n+H(S))$.

Example. Let $|X_1| = 2$, $|X_i| = 2^{i-1}$ $(i = 2, \cdots, k-1)$ and $n = 2^k$. Then minimal mergesort sorts $S(X)$ in $O(n)$ time, since $H(S) = O(n)$, whereas natural mergesort takes $O(n \log\log n)$ time to sort $S(X)$.

Lemma 2 For any sorting algorithm A and k integers n_1, ..., n_k such that $n_i \geq 2$ and $k \geq 1$, there exist k sorted lists X_1, ..., X_k such that $|X_i| = n_i$ and A runs in $\Omega(H(S))$ time when given $S = (X_1, \ldots, X_k)$ as input.

Proof. Sorting $S(X)$ into $S'(X) = (a'_1, \cdots, a'_n)$ where $a'_1 \leq \cdots \leq a'_n$ means that $S'(X)$ is a permutation of $S(X)$. To establish a lower bound, we can assume that all elements in X are distinct. Let $X_i = (a_1^{(i)}, \cdots, a_{n_i}^{(i)})$. Let $a_{n_i}^{(i)}$ $(i = 1, \cdots, k)$, the last element of X_i, be fixed to be the i-th largest element in X. Then there are $\binom{n-k}{n_1-1}$ possibilities of X_1 being scattered in $S'(X)$. Since the constraint of $a_{n_1}^{(1)} > a_1^{(2)}$ is satisfied by the choice of $a_{n_1}^{(1)}$, we have $\binom{n-k-n_1+1}{n_2-1}$ possibilities of X_2 being scattered in $S'(X)$. Repeating this calculation yields the number of possibilities N as

$$N = \frac{(n-k)!}{(n_1-1)!\,(n-k-n_1+1)!} \times \frac{(n-k-n_1+1)!}{(n_2-1)!\,(n-k-n_1-n_2+2)!}$$
$$\times \cdots \frac{(n_{k-1}-1)!}{(n_k-1)!\,0!} = \frac{n\,!}{n_1! \cdots n_k!} \cdot \frac{n_1 \cdots n_k}{n(n-1)\cdots(n-k+1)}\,.$$

Since the number of possible permutations is not smaller than this, we have the lower bound T on the computing time based on the binary decision tree model approximated by $T = \log N$. In the following we use natural logarithm for notational convenience. The result should be multiplied by $\log_2 e$. We use the following integral approximation.

$$n \log n - n + 1 \leq \sum_{j=1}^{n} \log j \leq n \log n - n + \log n$$

T is evaluated by using the first inequality for n and the second for n_i,

$$T = \log N \geq \log n! - \sum_{i=1}^{k} \log n_i! + \sum_{i=1}^{k} (\log n_i - \log(n-i+1))$$
$$\geq \sum_{i=1}^{k} n_i \log \frac{n}{n_i} - k \log n + 1$$

since $\sum n_i \log \frac{n}{n_i}$ is minimum when $n_1 = \cdots = n_{k-1} = 2$ and $n_k = n - 2(k-1)$,

$$\begin{aligned} 2T - H(S) &\geq \sum n_i \log \frac{n}{n_i} - 2k \log n + 2 \\ &\geq 2(k-1) \log \frac{n}{2} + (n - 2k + 2) \log \frac{n}{n - 2k + 2} - 2k \log n + 2 \\ &= (n - 2k) \log \frac{n}{n - 2k + 2} - 2 \log(n - 2k + 2) + 4 \\ &\geq -2 \log(n - 2k + 2), \end{aligned}$$

since $1 \leq k \leq n/2$. On the other hand we can show $T \geq \log(n - 2k + 2)$ from a combinatorial consideration, that is, when $n_1 = 2$, $a_1^{(1)}$ can go to any of $n - k$ places other than the last k places, meaning $N \geq n - k$. Thus we have $T \geq H(S)/4 = \Omega(H(S))$. ■

Remark. The above constant 1/4 may not be sharp. Indeed, if we use Stirling's approximation formula for factorial, we can improve this constant, although we cannot cover the extreme cases where some n_i are small despite a large n.

Theorem 3 When the lengths of ascending runs are 2 or greater, minimal mergesort is asymptotically optimal under the entropy measure.

If n_i=1 for some i, $a_{n_i}^{(i)}$ and $a_1^{(i+1)}$ form a part of a descending sequence. By reversing the descending sequences, we can guarantee that the sequence is decomposed into ascending runs of length at least 2. Let us extend minimal mergesort with this extra scanning in linear time, and define the entropy on the modified sequence. From this extension and the above lemma we see that minimal mergesort is asymptotically optimal for any sequence under the extended entropy measure, excluding scanning.

We can define entropy by decomposing the given sequence in non-ascending portions. Minimal mergesort is not optimal under the entropy measure defined in this way. There are more entropy measures defined in Ref. 2).

4. Use of the Array-based Merge Algorithm

We use the straight-forward merging with $|W_1| + |W_2| - 1$ key comparisons. This merge can be implemented on arrays whereby efficiency is increased. We define the adjusting term $f(i)$ to be the number of unmerged elements, where merged elements are those that have been compared in a merge operation at line 9. For the potential in equation (1), We set $\Phi(S_i) = -cH(S_i) - f(i)$. Here $f(i)$ is an adjusting term at step i and constant c is defined in the analysis below. The accounting equation becomes $a_i = t_i - c\Delta H(S_i) - \Delta f(i)$, where $\Delta H(S_i) = H(S_{i-1}) - H(S_i)$ and $\Delta f(i) = f(i-1) - f(i)$. That is, the amortized time is the actual time minus the decrease of entropy minus the decrease of f at the i-th step.

Let T and A be the actual total time and the amortized total time. Summing up a_i for $i = 1, \ldots, N$, we have $A = T + c(H(S_N) - H(S_0)) + f(N) - f(0)$, or $T = A + c(H(S_0) - H(S_N)) + f(0) - f(N)$.

Lemma 3 If W_2 is not an original X_i for any i, it holds that $|W_2| \leq 2|W_1|$.

Proof. Suppose to the contrary that $|W_2| > 2|W_1|$. Then for the previously merged lists V_1 and V_2, that is, $W_2 = \text{merge}\,(V_1, V_2)$, we have $|V_1| > |W_1|$ or $|V_2| > |W_1|$. Thus V_1 or V_2 must have been merged with W_1 or a shorter list, a contradiction. ■

Lemma 4 The amortized time for the i-th merge is not greater than zero.

Proof. Let $|W_1| = n_1$ and $|W_2| = n_2$. Note that $f(i)$ are initially n, non-negative and monotonically non-increasing. The change of entropy occurs only with n_1 and n_2. Let W_2 be a merged list and constant c be $c = 1/\log(3/2)$. Then, noting $n_1 \leq n_2 \leq 2n_1$, the decrease of entropy is

$$\begin{aligned} \Delta H(S_i) &= n_1 \log(1 + n_2/n_1) + n_2 \log(1 + n_1/n_2) \\ &\geq n_1 \log 2 + n_2 \log(3/2) \geq \log(3/2)(n_1 + n_2) \end{aligned}$$

Thus, noting that $\Delta f(i) \geq 0$,

$a_i = n_1 + n_2 - 1 - c\Delta H - \Delta f(i) \leq 0$, where $c = 1/\log(3/2) > 1$.

When W_2 is one of the original lists, that is, unmerged, we observe $\Delta f(i) \geq n_2$, and also that $\Delta H(S_i) \geq n_1$, since $n_1 \leq n_2$. Thus we have $a_i \leq n_1 + n_2 - 1 - (cn_1 + n_2) \leq 0$. ■

Theorem 4 The algorithm minimal mergesort sorts $S(X) = (X_1, \cdots, X_k)$ where each X_i is an ascending sequence in $O(n + H(S))$ time.

Proof. Theorem follows from Lemma 4 and the initial entropy is given by $H(S)$ and that $f(0) = n$. ■

The total time is the same as before, that is, $O(n + H(S))$.

5. Remaining Work for the MST Problem

Let $G = (V, E)$ be an undirected graph with edge cost function $c(u, v)$ for the edge (u, v). Let Kruskal's algorithm continue to work for the minimum (cost) spanning tree (MST) problem after the problem has been solved partially by the same algorithm. We estimate how much more time is needed to complete the work by using the concept of entropy. Let $G_1 = (V_1, E_1)$, ..., $G_k = (V_k, E_k)$ be subgraphs of G such that V_1, ..., V_k form a decomposition of V and G_i is the induced sub-graph from V_i. We assume the MST problem has been solved for G_i with spanning trees T_i for $i = 1, \ldots, k$. The state of data $S(V)$ is defined by $S(V) = (V_1, \ldots, V_k)$, and the entropy of the state is defined by (2) where X_i is interpreted as V_i.

The remaining work is to keep merging two trees by connecting them by the best possible edge. We use array *name* to keep track of names of trees to which vertices belong. If the two end points of an edge have different names, it connects distinct trees successfully. Otherwise it would form a cycle, not desirable situation, resulting in skipping the edge. The following algorithm completes the work from line 4.

ALGORITHM 3 {To complete the MST problem}
1 Let the sorted edge list L have been partially scanned
2 Let minimum spanning trees for G_1, ..., G_k have been obtained
3 Let $name[v] = i$ for $v \in V_i$ have been set for $i = 1, \ldots, k$
4 **while** $k > 1$ **do begin**
5 Remove the first edge (u, v) from L
6 **if** u and v belong to different subtrees T_1 and T_2 (without loss of generality)
7 **then begin**
8 Connect T_1 and T_2 by (u, v);
9 Change the names of the nodes in the smaller tree
to that of the larger tree;
10 $k := k - 1$;
11 **end**
12 **end**.

The analysis is similar to the proof of Lemma 1. We first analyze the time for name changes at line 9. Let t_i and a_i be the actual time and amortized time for the i-th operation that merges T_1 and T_2, where $|T_1| = n_1$ and $|T_2| = n_2$ and V_1 and V_2 are the sets of vertices corresponding to those spanning sub-trees. We measure the time by the number of name changes. Let the state of data after the i-th merge be S_i. The change of entropy occurs only with n_1 and n_2. Thus the decrease of entropy is

$$\Delta H(S_i) = n_1 \log(1 + n_2/n_1) + n_2 \log(1 + n_1/n_2) \geq \min\{n_1, n_2\}$$

Noting that $t_i = \min\{n_1, n_2\}$ and letting $c = 1$, amortized time becomes $a_i = t_i - \Delta H(S_i) \leq 0$. The rest of work is bounded by $O(m)$. We conclude this section by the following theorem.

Theorem 5 The remaining work becomes $O(m + H(S))$, where $H(S)$ is the initial entropy at the beginning of line 4.

6. Brief Review of 2-3 Heap and Delete Operations

In this section, we briefly review the 2-3 heap, which is a priority queue used for the single source shortest path algorithm in the next section. For details, the reader is referred to Ref. 8), and also Appendix. We use the 2-3 heap rather than the Fibonacci heap, since the former has a more rigid structure of dimensionality, and it is easier for describing the delete operation.

A 2-3 heap is a collection of heap ordered trees, which are formed through repeated formation of trunks. A trunk consists of 2 or 3 nodes (we call this the 2-3 condition) whose labels (keys) are heap-ordered. The number of trunks connected to the root of a heap ordered tree is called the degree of the tree. We make a heap-ordered tree of larger size by connecting the roots of trees of the same degree to form a trunk in such a way that their labels are heap-ordered and the trunk follows the 2-3 condition. We define the dimension of a trunk as follows. The dimension of a single node is 0. If we make a trunk of the roots of trees of the same degree, the dimension of the new trunk is one greater than the dimension of the highest-dimensional trunk of the trees, which is the degree of those trees.

A 2-3 heap is symbolically given by $P = \mathbf{a}_{k-1}T(k-1)+\ldots+\mathbf{a}_0T(0)$, where $T(i)$ is a tree of degree i. As there can be some $T(i)$'s, $T(i)$ can be said to be a tree type. The meaning of $\mathbf{a}_iT(i)$ is that 0, 1, or 2 roots of $T(i)$'s are connected in the trunk $\mathbf{a}_i$ where $\mathbf{a}_i$ is a 0-node trunk, 1-node trunk or 2-node trunk respectively. These trunks are expressed by boldface, i.e., **0**, **1** and **2**. We assume there is no tree if it is a 0-node trunk. The trunk $\mathbf{a}_i$ forms the highest-dimensional trunk of $\mathbf{a}_iT(i)$ and is called the main trunk. The main trunk does not fully follow the 2-3 condition.

If there are n nodes in the 2-3 heap, there are $O(\log n)$ trees of different degrees, that is, k is bounded by $O(\log n)$, and also the largest degree of trees is $O(\log n)$. The potential of a 2-node trunk is defined to be 1 and that of a 3-node trunk is 3. The potential of a 2-3 heap is the sum of the potentials of all trunks.

As the minimum node, which has the minimum label, is among the roots of the trees, find-min can be done in $O(\log n)$ time. Delete-min can be done in the following way after find-min is done. The minimum node, which is the root of a tree, is deleted, resulting in $O(\log n)$ trees broken apart, which were connected to the root. They are brought to the existing trees, and merged with them. Merging is done using the highest-dimensional trunks. If the highest-dimensional trunk has more than three nodes after merge, that trunk is cut into two; one with a 3-node trunk, which is a carry to the next position of trees of higher degree, and the remaining one. The carry is merged in the next position, etc. The underlying trees are moved together in this carry propagation. The amortized time for delete-min is shown to be $O(\log n)$.

Decrease-key is to decrease the key value of a node v, and recover the condition for the 2-3 heap. This is done by removing the tree rooted at v, and merge the tree with the existing trees. We call this the merging at the root level. Also we re-shape the work space if it does not satisfy the condition after removal. The work space is defined by the highest dimensional trunk $\mathbf{t}$ of dimension, say, i, on which v was sitting, and one or two trunks whose roots and that of $\mathbf{t}$ share the same trunk of dimension $i+1$. Using the potential defined above, we can show the amortized time for a decrease-key is $O(1)$.

We maintain the priority queue for the single source shortest path problem by a 2-3 heap. In traditional priority queues, decrease-key, insert and delete-min

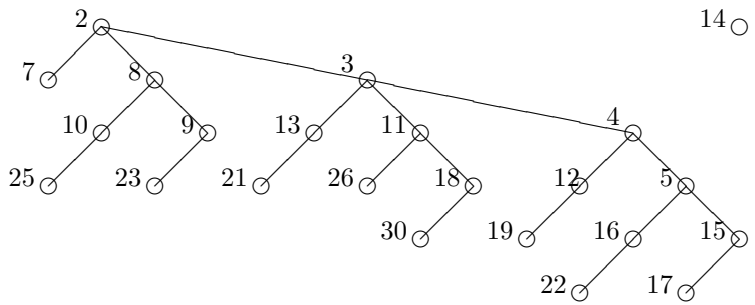


Fig. 1 The 2-3 heap.

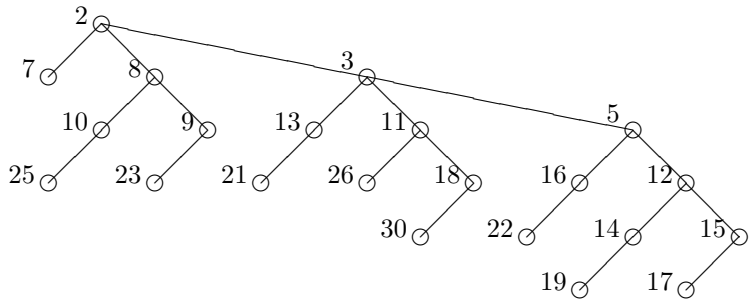


Fig. 2 The 2-3 heap after deleting 4.

operations are defined. We define a delete operation on a 2-3 heap. When we delete node v, we remove the subtree rooted at v similarly to decrease-key on v, entailing a re-shape of the work space. After destroying v, the subtrees of v are broken apart like delete-min. We merge them with the trees at the root level. The amortized time for a delete is proportional to the number of the subtrees, which is $O(\log n_v)$, where n_v is the number of descendants of node v to be deleted. The complexity $O(\log n_v)$, not $O(\log n)$, is crucial. A delete is defined on a Fibonacci heap in Ref. 9).

Example 1 Suppose we delete node(4) in **Fig. 1**. Then we have the following $T = \mathbf{1}T(3)$ given in **Fig. 2**.

Let us express trunks by tuples and the node with label x by $node(x)$. After $node(4)$ is deleted, we have a $\mathbf{2}T(0)$ with $(12, 19)$ as the main trunk and a $\mathbf{2}T(1)$ with $(5, 16, 22)$ and $(15, 17)$ connected in heap order. After (12, 14, 19) is formed at position 0, it is carried to $\mathbf{2}T(1)$, resulting in a $T(2)$, which is carried to the next position.

Let us delete nodes $v_j (j = 1, \ldots, k)$ in the batch mode from a 2-3 heap of size n. In the batch mode, we remove the sub-trees rooted at v_j for all j and bring them at the root level similarly to decrease-key. We process v_j one by one from lower dimensions with necessary changes for re-shape. After all v_j are processed, we destroy v_j, disconnect all children of all v_j and merge them at root level. Assume the number of descendants of v_j immediately before merge is n_j. The total amortized time T of deleting $v_1, \ldots, v_k$ in the batch mode is $T = O(\log n_1 + \ldots + \log n + k)$. Noting that $n_1 + \ldots + n_k \leq n$ in the batch mode, T is maximized as $T = O(k(\log(n/k) + 1))$ when $n_1 = \ldots = n_k = n/k$. Thus

Lemma 5 k consecutive delete operations on a 2-3 heap of size n can be done in $O(k(\log(n/k) + 1)$ time.

Now we perform t batches of delete operations. Assume the i-th batch has k_i delete operations. Let the time for the i-th batch of delete operations be denoted by T_i. Since $T_i = O(k_i(\log(n/k_i) + 1))$ by Lemma 5, we have the total time for all deletes bounded within a constant factor by

$$\begin{aligned} k_1(\log(n/k_1) + 1) + \ldots + k_t(\log(n/k_t) + 1) &= n(\Sigma_{i=1}^{t}(k_i/n)(\log(n/k_i) + 1)) \\ &= n + n(-\Sigma_{i=1}^{t} p_i \log p_i), \end{aligned}$$

where $p_i = k_i/n$. We define $H(S) = -n\Sigma_{i=1}^{t} p_i \log p_i$.

Let us perform those t batches of delete operations after t find-min operations; each batch after each find-min. One find-min operation can be done in $O(\log n)$ time. Thus the total time becomes $O(t \log n + n + H(S))$, which is further simplified to $O(n + H(S))$ by the following lemma.

Lemma 6 For $t \geq 2$, $t \log n \leq O(H(S))$. Thus the time for heap operations described above is bounded by $O(n + H(S))$.

Proof. $H(S)$ is minimum when $k_1 = \ldots = k_{t-1} = 1$, and $k_t = n - t + 1$. Thus $2H(S) \geq 2(t-1)\log n + 2(n - t + 1)\log(n/(n - t + 1)) \geq t \log n$ ■

Remark. The operation defined above can be viewed as a generalization of delete-min=(find-min, delete) to (find-min, delete, ..., delete).

7. Application to Shortest Paths for Nearly Acyclic Graphs

If the given directed graph with non-negative edge costs is nearly acyclic, we can solve the single source shortest path (SSSP) problem faster than a general graph as noted in Refs. 9), 10), etc. The main purpose of this section is to investigate the possibility of measuring the degree of acyclicity of the given graph by entropy. As the degree of acyclicity is determined by the SSSP algorithm itself, or in other words, dynamically, we may well wish to have a measure derived from the structure of the graph. This issue will be addressed in the next section. Apart from the analysis of data structures, we analyze the computing time directly in this section, not by amortised analysis of Section 2.

Let $G = (V, E)$ be a directed graph where V is the set of vertices with $|V| = n$ and E is the set of edges with $|E| = m$. We assume every vertex is reachable from the source so that $m \geq n$. The non-negative cost of edge (v_i, v_j) is denoted by $c(v_i, v_j)$. Let $OUT(v)$ (also $IN(v)$) be the list of edges from (to) v expressed by the set of the other end points of edges from (to) v. A brief description of Dijkstra's algorithm [13] follows. Let S be the solution set, to which shortest distances have been established by the algorithm. The vertices in $V - S$ have tentative distances that are those of the shortest paths that go through S except for the end points. We take a vertex in $V - S$ that has the minimum distance, finalize it, and update the distances to other vertices in $V - S$ using edge list $OUT(v)$. If we organize $Q = V - S$ by a Fibonacci heap [14] or 2-3 heap [8], we can show the SSSP problem can be solved in $O(m + n \log n)$ time. We call this algorithm with one of those priority queues the standard single source algorithm. Note that we use the same symbol S for the state of data and the solution set, hoping this is not a source of confusion.

We give the following well known algorithm [11] and its correctness for acyclic graphs for the sake of completeness. See Ref. 11) for the proof. It runs in $O(m)$ time, that is, we do not need an operation of finding the minimum in the priority queue.

ALGORITHM 4 {$G = (V, E)$ is an acyclic graph.}
1 Topologically sort V and assume without loss of generality $V = \{v_1, \cdots, v_n\}$, where $(v_i, v_j) \in E \Rightarrow i < j$;
2 $d[v_1] := 0$; {v_1 is the source}
3 **for** $i := 2$ **to** n **do** $d[v_i] := \infty$;
4 **for** $i := 1$ **to** n **do**
5 **for** v_j such that $(v_i, v_j) \in E$ **do**
6 $d[v_j] := \min\{d[v_j], d[v_i] + c(v_i, v_j)\}$.

Lemma 7 At the beginning of Line 5 in Algorithm 4, the shortest distances from v_1 to v_j $(j < i)$ are computed. Also at the beginning of line 5, distances computed in $d[v_j]$ $(j \geq i)$ are those of shortest paths that lie in $\{v_1, \cdots, v_{i-1}\}$ except for v_j. Thus at the end shortest distances $d[v_i]$ are computed correctly for all $i (1 \leq i \leq n)$.

Abuaiadh and Kingston [9] gave a result by restricting the given graph to being nearly acyclic. When they solve the single source problem, they distinguish between two kinds of vertices in $V - S$. One is the set of vertices, "easy" ones, to which there are no edges from $V - S$, e.g., only edges from S. The other is the set of vertices, "difficult" ones, to which there are edges from $V - S$. To expand S, if there are easy vertices, those are included in S and distances to other vertices in $V - S$ are updated. If there are no easy vertices, the vertex with minimum tentative distance is chosen to be included in S. If the number of such delete-minimum operations is t, the authors show that the single source problem can be solved in $O(m + n \log t)$ time with use of a Fibonacci heap. That is, the second term of the complexity is improved from $n \log n$ to $n \log t$. If the graph is acyclic, $t = 1$ and we have $O(m + n)$ time. Since we have $O(m + n \log n)$ when $t = n$, the result is an improvement of Fredman and Tarjan with use of the new parameter t. The authors claim that if the given graph is nearly acyclic, t is expected to be small and thus we can have a speed up.

The definition of near acyclicity and the estimate of t under it is not clear, however. We will show that the second term can be bounded by the entropy derived from a structural property of the given graph.

ALGORITHM 5 {Single source shortest paths with v_0 being the source}[9),10)]
1 **for** $v \in V$ **do if** $v = v_0$ **then** $d[v] := 0$ **else** $d[v] := \infty$;
2 Organize V in a priority queue Q with $d[v]$ as key;
3 $S := \emptyset$;
4 **while** $S \neq V$ **do begin**
5 **if** there is a vertex v in $V - S$ with no incoming edge from $V - S$ **then**
6 Choose v
7 **else**
8 Choose v from $V - S$ such that $d[v]$ is minimum;
9 Delete v from Q;
10 $S := S \cup \{v\}$;
11 **for** $w \in OUT(v) \cap (V - S)$ **do** $d[w] := min\{d[w], d[v] + c(v, w)\}$
12 **end**.

It is shown in Ref. 9) that a sequence of n delete, m decrease-key and t find-min operations is processed in $O(m + n \log t)$ time, meaning that the SSSP problem can be solved in the same amount of time.

We use the 2-3 heap for priority queue Q with the additional operation of delete. Let $v_1, \ldots, v_k$ be deleted between two consecutive find-min operations such that v_1 is found at a find-min operation at line 8, and v_k is found at line 6 immediately before the next find-min. Each induced subgraph from them forms an acyclic graph, and they are topologically sorted in the order in which vertices are chosen at line 6. Thus they can be deleted from the heap without the effort of find-min operations. Let $V_1, \ldots, V_t$ be the sets of vertices such that V_i is the acyclic set chosen following the i-th find-min operation and just before the next find-min. We call this set the i-th acyclic set. We assume $|IN(v_0)| > 1$ so that the source is chosen by the first find-min. Then $S(V) = (V_1, \ldots, V_t)$ forms a decomposition of the set V, called acyclic decomposition. We denote the entropy of this decomposition by $H(S)$. Lemma 6 shows that t find-min operations with $|V_1| + \ldots + |V_t|$ deletes interleaved in Algorithm 5 (each i-th find-min followed by $|V_i|$ deletes) can be done in $O(n + H(S))$ time. Let m_s be the number of edges examined at line 11 between the s-th find-min operation and the $(s+1)$-th find-min operation. Between these operations, $O(m_s)$ amortized time (in the

context of data structures) is spent at line 11. The total time for line 11 becomes $O(m)$. When $t = 1$, the whole graph is acyclic, and we can solve the single source problem in $O(m)$ time by Lemma 7. The time for building Q at line 2 is absorbed in $O(m)$. Thus we have the following theorem.

Theorem 6 Algorithm 5 solves the SSSP problem in $O(m + H(S))$ time.

Remark. In Ref. 9), $O(t \log n + H(S))$ is bounded by $O(n \log t)$. Thus our analysis of $O(t \log n + H(S)) \leq O(H(S))$ is sharper.

8. Relationship with 1-dominator

As a definition of near-acyclicity, the definition and algorithm for a 1-dominator decomposition is given in Ref. 10). The decomposition is given by the set of disjoint sets, called maximal acyclic structures, whose union is V. A maximal acyclic structure dominated by a trigger v, A_v, is the maximal set of vertices w such that any path from outside A_v to w must go through v, and the subgraph induced by A_v is an acyclic graph. A_v is formally defined by the maximal set satisfying the following formula for any w.

The induced graph from A_v is acyclic, $v \in A_v$ and

$(w \in A_v)\&(w \neq v) \rightarrow (IN(w) \neq \phi)\&(IN(w) \subseteq A_v)$

In Ref. 10) it is shown V is uniquely decomposed into several A_v's, and the time for this decomposition is $O(m)$. The 1-dominator decomposition is used for identifying the set of the triggers, R. Only triggers are maintained in the heap. Once the distance to a trigger is finalized, the distances to members of the corresponding acyclic structure are finalized through Algorithm 4 in time proportional to the number of edges in the set. At the border of the set. the distances to other triggers are updated. The time for the SSSP problem becomes $O(m + r \log r)$, where r is the number of triggers, that is, $r = |R|$.

We show that the entropy $H(S)$ in Section 7 is bounded by the entropy defined by the 1-dominator decomposition.

Theorem 7 The 1-dominator decomposition is a refinement of the decomposition defined by Algorithm 5.

Proof. Suppose a vertex v is obtained by find-min at line 8 and v is not a trigger. Then v is inside some maximal acyclic structure. Since the distance to the corresponding trigger is smaller, the trigger must have been included in the solution set earlier, and v must have subsequently been deleted from the heap, a contradiction. Thus v is a trigger. Then the maximal acyclic structure is subsequently deleted from the heap, and possibly more maximal acyclic structures. Thus the 1-dominator decomposition is a refinement of the decomposition $S(V)$. ■

The decomposition by Algorithm 5 is dynamically defined, i.e., it cannot be defined statically before the algorithm starts. On the other hand, the algorithm based on the 1-dominator decomposition is more predictable as the preprocessing can reveal the 1-dominator decomposition and its entropy, which bounds the entropy defined by Algorithm 5.

In Ref. 10), the SSSP algorithm is given in a slightly different way. It maintains only triggers in the heap, and the distances between triggers are given by those of pseudo edges and associated costs, which are defined between triggers through the intervening acyclic part and obtained in $O(m)$ time. In other words, the standard single source algorithm runs on this reduced graph. Thus the time becomes $O(m + r \log r)$, which may be better than $O(m + H(S))$ of Algorithm 5. However Algorithm 5 and the SSSP algorithm in Ref. 10) are not incompatible. We can run Algorithm 5 on the reduced graph obtained through the 1-dominator decomposition. Then the time will become $O(m + H(S'))$, where $H(S')$ is the entropy defined by the algorithm run on the reduced graph. We note that $O(H'(S))$ is bounded by $O(H(S))$ and $O(r \log r)$).

9. Dual Entropy

We defined entropy for the algorithm based on merging. In this section we define entropy for a generic algorithm based on partitioning. We capture computation as a partitioning process. Specifically if the state of data X is given by $S(X) = (X_1, \ldots, X_k)$, where each X_i is unsolved and the order in which they are placed in $S(X)$ gives the meaning of partial solution. We remove a $W = X_i$, partition W into W_1 and W_2 and put them back to the location of X_i in the order of W_1 and W_2.

We define the entropy of S(X) by

$$H(S) = \Sigma_{i=1}^{k} |X_i| \log |X_i|$$

We call this $H(S)$ the dual entropy, and the entropy in Section 2 the primary entropy in contrast. We note that $H(S) \geq n \log(n/k)$.

The first example is quicksort for $S(X)$ where X_i satisfy $X_1 \leq X_2 \leq \ldots \leq X_k$. The meaning of $X \leq Y$ is for all x in X and y in Y, $x \leq y$.

Algorithm 6 Generic quicksort
$L = S(X)$;
while $|L| < n$ do
  W:=any member of L such that $|W| > 1$;
  partition W into W_1 and W_2 of equal size;
  put W_1 and W_2 in place of W in this order;
end

Remark. If $|W|$ is odd, we partition W into W_1, x and W_2 such that $W_1 \leq x \leq W_2$. Also we will have $n_1 + n_2 + 1$ instead of $n_1 + n_2$ in the following formula.

Let $|W_1| = n_1$ and $|W_2| = n_2$. The decrease of entropy is given by

$$(n_1 + n_2)\log(n_1 + n_2) - n_1 \log n_1 - n_2 \log n_2$$
$$= n_1 \log(1 + n_2/n_1) + n_2 \log(1 + n_1/n_2)$$

If we use the median selection algorithm of linear time, $O(cn)$, for the partition, we can have $n_1 = n_2$, meaning the decrease of entropy is $n_1 + n_2$. Thus the amortized time $a_i = c(n_1 + n_2) - c\Delta H(S_i) \leq 0$, and the computing time of the above quicksort becomes $O(H(S))$.

Obviously if we use any sorting algorithm of $O(n_i \log n_i)$ time for each $|X_i|$, we have the same time complexity. The analysis here is to highlight the meaning of dual entropy.

The next example is from the single source shortest path algorithm for a nearly acyclic graph.

In Section 8, we decomposed the graph into acyclic parts, where an acyclic part is a subgraph which is acyclic and has a vertex through which other members of the subgraph are reached from outside. If we have one such subgraph, the whole graph is acyclic and the SSSP problem can be solved in $O(m)$ time by Algorithm 4 [11]. Let those subgraphs be $G_1 = (V_1, E_1), \ldots, G_k = (V_k, E_k)$. If k is small, we can say the graph is nearly acyclic. We showed that the SSSP problem can be solved in $O(m+H(S))$ time where $H(S)$ is defined by the primary entropy for $S(V) = (V_1, \ldots, V_k)$.

The other nearly acyclic graph in this section is defined by decomposing the set V in a different way. Let us decompose the set of vertices into strongly connected (sc) components $V_1, \ldots, V_k$. Let $G_1 = (V_1, E_1), \ldots, G_k = (V_k, E_k)$ be subgraphs such that G_i is the subgraph induced from the sc-component V_i. If the maximum size, r, of the sc-components is small, we can say the graph is nearly acyclic. When $r = 1$, the graph is acyclic. We assume a meta graph $\tilde{G} = (\tilde{V}, \tilde{E})$ such that vertices are given by $V_1, \ldots, V_k$ and there is an edge from V_i to V_j if there is an edge from a vertex in V_i to a vertex in V_j. Tarjan's algorithm for sc-components is based on depth-first search and $V_1, \ldots, V_k$ are available in topological order of the meta graph, which is acyclic. This decomposition is regarded as dual to the acyclic decomposition in which each component graph is acyclic.

The generalized single source shortest path (GSS) algorithm is to generalize the SSSP problem in such a way that we initialize distances arbitrarily over vertices. Note that the initial distribution of distances in the traditional SSSP is given by 0 for the source and infinity for others.

We expand Algorithm 4 to the following algorithm with line numbers expanded with dots.

Algorithm 7 {Solve the SSSP problem for graph $G = (V, E)$ and source v_0}
1 Compute sc-components $V_1, V_2, \cdots, V_k$ in topological order in meta graph;
2.1 **for** $v \in V$ **do** $d[v] := \infty$;
2.2 $d[v_0] := 0$; {For source v_0 let $v_0 \in V_1$ without loss of generality}
3 **for** $i := 2$ **to** k **do for** $v \in V_i$ **do** $d[v] := \infty$;
4.1 **for** $i := 1$ **to** k **do begin**
4.2 Solve the GSS for G_i;
5 **for** V_j such that $(V_i, V_j) \in \tilde{E}$ **do**
6.1 **for** $v \in V_i$ and $w \in V_j$ such that $(v, w) \in E$ **do**
6.2 $d[w] := \min\{d[w], d[v] + c(v, w)\}$
7 **end** of for at line 4.1.

Line 4.2 is to obtain shortest distances within sc-components whereas lines 6.1 and 6.2 are to update distances through edges between sc-components.

The correctness of this algorithm is given in Ref. 12), in which the time complexity is given by $O(m+n\log r)$. We give a sharper analysis by the dual entropy. The time for line 1 is $O(m)$. The time spent at line 6.2 is $O(m)$ in total. The time for the GSS for G_i at line 4.2 is $O(m_i+n_i\log n_i)$ where $n_i=|V_i|$. Summing up those complexities yields $O(m+H(S))$ where $H(S)$ is the dual entropy of $S(V)=(V_1,\ldots,V_k)$.

It is the topological order in which the sc-components are placed that gives the meaning of partial solution, whereas in the acyclic decomposition, the acyclic structures can be regarded as partially solved. This relation is similar to that between minimal mergesort and quicksort given earlier. Algorithm 5 is regarded as a process of merging acyclic parts into the solution set whereas Algorithm 7 is regarded as that of partitioning process of sc-components into the solution set. The solution is given by the shortest path spanning tree in both cases. In the former this is regarded as the solved acyclic structure, which is actually a tree. In the latter this is regarded as a set of singleton sc-components.

10. Concluding Remarks

We captured computation as a process of reducing entropy, starting from some positive value and ending in zero. We showed that three specific problems of sorting, shortest paths and minimum spanning trees can be analyzed by primary entropy. The amortized time for a step of the computation is the actual time minus the reduction of entropy. If the analysis of a single amortized time is easier than the analysis of the total actual time, this method by amortization will be useful for analysis.

We also defined the dual concept of the above mentioned entropy that can be used for algorithms based on partitioning. The analysis is rather straightforward, but can shed light on the algorithmic process and reduction of entropy.

If the computation process is a merging process of two sets in the decomposition, or partitioning process of a set into two, our method may be used. The definition of entropy and actual time needs care depending on the specifics of each problem. It remains to be seen if more difficult problems can be analyzed by this method.

The primary and dual entropy are defined in the worst case. They can be defined in the average case, and may be used for a sharper analysis of expected times of some other problems.

The motivation of this paper is an intuitive notion of partial solution. If the given problem is likely to be partially solved, or easier to solve than the general case in other words, we want to minimize our effort to complete the solution using that information. More formal treatments of those concepts will need to be done in future research.

References

1) Nakagawa, Y.: A Difficulty Estimation Method for Multidimensional Nonlinear 0-1 Knapsack Problem Using Entropy, *Trans. Institute of Electronics, Communication and Information*, Vol.J87-A, No.3, pp.406–408 (2004).
2) Takaoka, T.: Entropy–Measure of Disorder, *Proc. CATS* (*Computation: Australasian Theory Symposium*), pp.77–85 (1998).
3) Takaoka, T.: Partial Solution and Entropy, *MFCS 2009, LNCS 5734*, pp.700–711 (2009).
4) Estivill-Castro, V. and Wood, D.: A survey of adaptive sorting algorithms, *ACM Computing Surveys* 24, pp.441–476 (1992).
5) Knuth, D.E.: The Art of Computer Programming, Vol.3, Sorting and Searching, *Reading, Mass.*, Addison-Wesley (1974).
6) Mannila, H.: Measures of presortedness and optimal sorting algorithms, *IEEE Trans. Comput. C-34*, pp.318–325 (1985).
7) Brown, M.R. and Tarjan, R.E.: A Fast Merging Algorithm, *Journal of the ACM*, Vol.26, No.2, pp.211–226 (1979).
8) Takaoka, T.: Theory of 2-3 Heaps, *Discrete Applied Math*, Vol.126, pp.115–128 (2003).
9) Abuaiadh, D. and Kingston, J.H.: Are Fibonacci heaps optimal?, *ISAAC'94, LNCS 834*, pp.442–450 (1994).
10) Saunders, S. and Takaoka, T.: Solving shortest paths efficiently on nearly acyclic directed graphs, *Theoretical Computer Science*, 370(1-3), pp.94–109 (2007).
11) Tarjan, R.E.: Data Structures and Network Algorithms, *Regional Conference Series in Applied Math.*, 44 (1983)
12) Takaoka, T.: Shortest path algorithms for nearly acyclic directed graphs, Theoretical Computer Science, Vol.203, pp.143–150 (1998).
13) Dijkstra, E.W.: A note on two problems in connection with graphs, *Numer.Math.*, Vol.1, pp.269–271 (1959).

14) Fredman, M.L. and Tarjan, R.E.: Fibonacci heaps and their use in improved network optimization problems, *JACM*, Vol.34, No.3, pp.596–615 (1987).

Appendix: Review of 2-3 heap

In this appendix we review the 2-3 heap as it is used in the shortest path algorithm discussed earlier. We first define a polynomial queue, which has a rigid structure based on dimensionality. Then we define a 2-3 heap, which is a more flexible version of a 3-nomial queue. That is, a 2-3 heap allows some deformation of the tree structure within some tolerance bound, which makes possible an $O(1)$ amortized time for decrease-key and insert. For the full description, readers are referred to Ref. 8).

We define algebraic operations on trees. We deal with rooted trees in the following. A tree consists of nodes and branches, each branch connecting two nodes. The root of tree T is denoted by $root(T)$. A linear tree, called a trunk, of size r is a linear list of r nodes such that its first element is regarded as the root and a branch exists from a node to the next. The $r-1$ nodes below the root are the first, second, ..., $(r-1)$-th child. The linear tree of size r is expressed by bold face $\mathbf{r}$. Thus a single node is denoted by $\mathbf{1}$, which is an identity in our tree algebra. The empty tree is denoted by $\mathbf{0}$, which serves as the zero element. A product of two trees S and T, $P = ST$, is defined in such a way that every node of S is replaced by a copy of T and every branch in S connecting two nodes u and v now connects the roots of the trees substituted for u and v in S. Note that $\mathbf{2} * \mathbf{2} \neq \mathbf{4}$, for example, and also that $ST \neq TS$ in general. The symbol "$*$" is used to avoid ambiguity.

The number of first children of node v is called the degree of v and denoted by $deg(v)$. The degree of tree T, $deg(T)$, is defined by $deg(root(T))$. The sum of two trees S and T, denoted by $S+T$, is just the collection of two trees S and T. A polynomial of trees is defined next. Since the operation of product is associative, we use the notation of $\mathbf{r}^i$ for the products of i $\mathbf{r}$'s. Note that $deg(\mathbf{r}^i) = i$. An r-ary polynomial of trees of degree $k-1$, P, is defined by

$$P = \mathbf{a}_{k-1}\mathbf{r}^{k-1} + \ldots + \mathbf{a}_1\mathbf{r} + \mathbf{a}_0 \qquad (3)$$

where $\mathbf{a}_i$ is a linear tree of size a_i and called a coefficient in the polynomial. Let $|P|$ be the number of nodes in P and $|\mathbf{a}_i| = a_i$. Then we have $|P| = a_{k-1}r^{k-1} + \ldots + a_1 r + a_0$. We choose a_i to be $0 \leq a_i \leq r-1$, so that n nodes can be expressed by the above polynomial of trees uniquely, like the k digit radix-r expression of n is unique with $k = \lceil \log_r(n+1) \rceil$. The term $\mathbf{a}_i\mathbf{r}^i$ is called the i-th term. We call $\mathbf{r}^i$ the complete tree of degree i. Let the operation "$\bullet$" be defined by the tree $L = S \bullet T$ for trees S and T. The tree L is made by linking S and T in such a way that $root(T)$ is connected as a child of $root(S)$. This operation is not associative. The product $\mathbf{r}^i = \mathbf{r}\mathbf{r}^{i-1}$ is expressed by

$$\mathbf{r}^i = \mathbf{r}^{i-1} \bullet \ldots \bullet \mathbf{r}^{i-1} (r-1 \bullet\text{'s are evaluated right to left}) \qquad (4)$$

The whole operation in Eq. (4) is to link r trees, called an i-th r-ary linking. The path of length $r-1$ created by the r-ary linking is called the i-th trunk of the tree $\mathbf{r}^i$, which defines the i-th dimension of the tree in a geometrical sense. If two trunks $\mathbf{a}$ and $\mathbf{b}$ of the same dimension share the same trunk of dimension higher by 1 in the sense that their roots are sitting on the trunk, we say $\mathbf{a}$ and $\mathbf{b}$ are siblings.

The j-th $\mathbf{r}^{i-1}$ in Eq. (4) is called the j-th subtree on the trunk. Let the dimension of the highest-dimensional trunk on which a node v is sitting be i. The dimension of node v is defined to be $i-1$. The path created by linking a_i trees of $\mathbf{r}^i$ in Eq. (3) is called the main trunk of the tree corresponding to this term. A polynomial of trees is regarded as a collection of trees of distinct degrees connected by their main trunks.

We next define a polynomial queue. An r-nomial queue is an r-ary polynomial of trees with a label $label(v)$ attached to each node v such that if u is a parent or elder sibling of v, $label(u) \leq label(v)$. A binomial queue is a 2-nomial queue.

Example 2 A polynomial queue with $r = 3$ and an underlying polynomial of trees $P = \mathbf{2} * \mathbf{3}^2 + \mathbf{2} * \mathbf{3} + \mathbf{2}$ is given in **Fig. 3**.

Each term $\mathbf{a}_i\mathbf{r}^i$ in Eq. (3) is a tree of degree $i+1$ if $a_i > 1$. One additional degree is caused by the coefficient. The merging of two linear trees $\mathbf{r}$ and $\mathbf{s}$ is to merge the two lists by their labels. The result is denoted by the sum $\mathbf{r}+\mathbf{s}$. The merging of two terms $\mathbf{a}_i\mathbf{r}^i$ and $\mathbf{a'}_i\mathbf{r}^i$ is to merge the main trunks of the two trees by their labels, spending $a_i + a'_i - 1$ comparisons. When the roots are merged, the trees underneath are moved accordingly. If $a_i + a'_i < r$, we have the merged tree with coefficient $\mathbf{a}_i + \mathbf{a}'_i$. Otherwise we have a carry tree $\mathbf{r}^{i+1}$ and the remaining

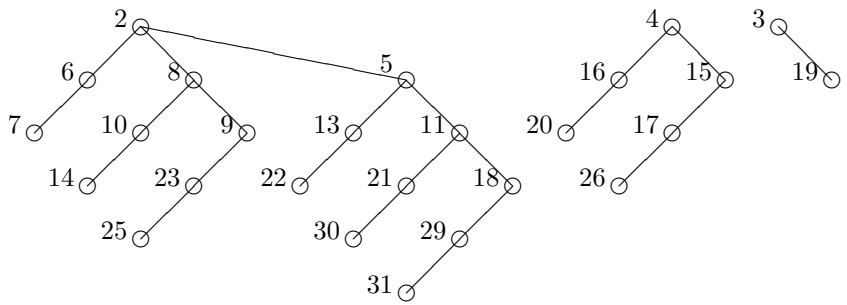


Fig. 3 A polynomial queue with $r = 3$.

tree with the main trunk of length $a_i + a'_i - r$. The sum of two polynomial queues P and Q is made by merging two polynomial queues in a very similar way to the addition of two radix-r numbers. We start from the 0-th term. Two i-th terms from both queues and a carry if $i > 0$ are merged, causing a possible carry to the $(i+1)$-th terms. Then we proceed to the $(i+1)$-th term.

An insertion of a key into a polynomial queue is to merge a single node with the label of the key into the 0-th term, taking $O(r \log_r n)$ time for possible propagation of carries to higher terms. Thus n insertions will form a polynomial queue P in $O(nr \log_r n)$ time. The value of k in Eq. (3) is $O(\log_r n)$ when we have n nodes in the polynomial of trees.

Delete-min is done by finding the minimum node, which has the minimum key, and deleting the node. Suppose the node is in $\mathbf{a}_i\mathbf{r}^i$. Deleting the root produces $\mathbf{r-1}$, $(\mathbf{r-1})\mathbf{r}$, ..., $(\mathbf{a_i-1})\mathbf{r}^i$, which can be regarded as another polynomial queue and merged into the existing polynomial queue. Here $\mathbf{a-1}$ is a trunk of size $a-1$. We can take n successive minima from the queue by deleting the minimum in some tree T, adding the resulting polynomial queue Q to $P-T$, and repeating this process. This will sort the n numbers in $O(nr \log_r n)$ time after the queue is made. Thus the total time for sorting is $O(nr \log_r n)$. In the sorting process, we do not change key values. If the key values are updated frequently, however, this structure of polynomial queue is not flexible. The best we can do for decrease-key is to keep swapping the node with siblings or parents after the key is decreased until finding a proper position, spending $O(\log_r n)$ time. Also we cannot afford to spend $O(n \log n)$ time for build-heap for shortest path algorithms. The following data structure can accommodate those issues.

A 2-3 heap is similar to 3-nomial queue, but the size of each trunk is limited to 2 or 3, except for main trunks whose size can be 0, 1 or 2. The potential of a 2-3 heap is the sum of potentials of all trunks. The potential of a two-node trunk is 1, and that of a 3-node trunk is 3. The amortized cost of an operation is the number of comparisons used minus the increase of potential.

A 2-3 heap is a collection of heap-ordered trees of different degrees, given in a general form below.

$P = \mathbf{a}_{k-1}T(k-1) + \ldots + \mathbf{a}_0 T(0)$

Here $T(i)$ is a tree of degree i and $T(0) = \mathbf{1}$. We express various trees of degree i by $T(i)$, meaning that $T(i)$ is a type of a tree. We do not distinguish between a tree and tree type. We call $\mathbf{a}_i$ the main trunk at position i.

In the following, comparisons between nodes mean comparisons between their keys.

The merge process at the root level is now described. When a tree $T'(i)$, whose root is v, is removed from a bigger tree, it is brought to $\mathbf{a}_i T(i)$ and merged. Merge means that v is merged into $\mathbf{a}_i$ together with the underlying trees of $T'(i)$. If $\mathbf{a}_i$ is $\mathbf{1}$, v is compared with one node and placed above or below. If $\mathbf{a}_i$ is $\mathbf{2}$, v is compared with two nodes, and merged, that is, inserted into appropriate place spending two comparisons. As the increase of potential and the number of comparisons are equal, the amortized cost is zero.

Insert and delete-min are similar to those for trinomial queue, but we can show insert is done in $O(1)$ amortized time and delete-min is in $O(\log n)$ amortized time. Thus heap initialization for n items can be done in $O(n)$ time.

A decrease-key is to decrease the key value of the node, and remove the tree rooted at v and take it to the root level. At the root level, the tree is merged with the existing tree of the same degree. After the tree rooted at v is gone, the near-by trees are reconnected to recover the requirement of the 2-3 tree. This recovery process is done within the work space of v, which is defined by the trunk on which v is sitting and one or two sibling trunks. Including v, the size of the work space is between 4 and 9. If the size is 5 or greater, we can recover the structure by moving trees within the work place. If it is 4, one tree is moved to form a 3-node trunk, and recovery process continues to a work space of higher

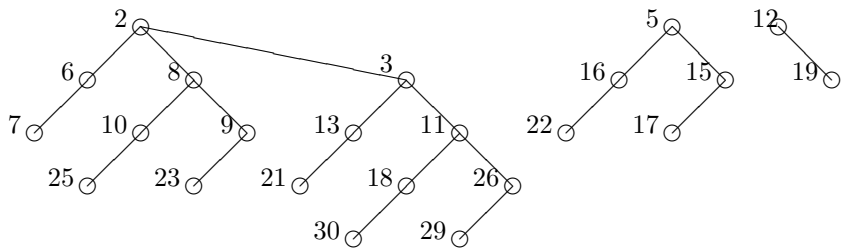


Fig. 4 A 2-3 heap.

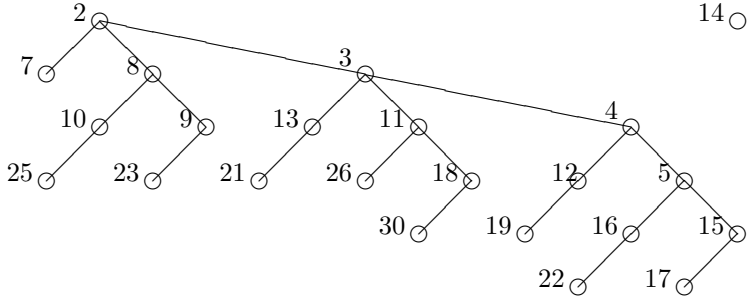


Fig. 5 The 2-3 heap after two decrease-key operations.

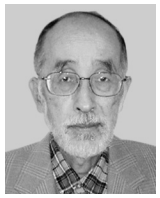
dimension, since a 2-node trunk cannot shrink further.

The amortized time for a decrease-key operation is shown to be $O(1)$. The re-adjustment with near-by trees of the tree at v takes $O(1)$ amortized time. The amortized cost for the merge process at the root level is zero.

Example 3 See $P = \mathbf{2}T(2) + \mathbf{2}T(1) + \mathbf{2}T(0)$ in **Fig. 4**. In examples, we name the node with label x by $node(x)$. The first tree has three trunks of dimensions, 1, 2 and 3. $node(6)$ and $node(7)$ of dimension 0 are siblings on the same trunk of dimension 1, and children of $node(2)$. Similarly $node(8)$ and $node(9)$ are of dimension 1. $node(10)$ and $node(25)$ are children of $node(8)$ of dimension 0. Trunks (2, 6, 7) and (8, 10, 25) are siblings.

Example 4 Suppose we decrease labels 6 to 4, and 29 to 14 in Fig. 4. Then we have the following $T = \mathbf{1}T(3)+\mathbf{1}T(0)$ in **Fig. 5**. We first remove $node(6)$, causing the move of $node(7)$ to the child position of $node(2)$. The new node $node(4)$ is inserted into $2T(0)$, resulting in $T(1)$ with $node(12)$ and $node(19)$ and carrying over to $2T(1)$ to cause insertion. Then the newly formed $T(2)$ will be carried to $2T(2)$, resulting in $T(3)$. For the second decrease-key, we have a new node $node(14)$. Since the trunk cannot shrink further, the two links from $node(11)$ to $node(18)$ and $node(26)$ are swapped.

(Received December 23, 2009)
(Accepted June 3, 2010)
(Original version of this article can be found in the Journal of Information Processing Vol.18, pp.227–241.)



Tadao Takaoka graduated from Kyoto University, and got a Doctor of Engineering degree in Applied Mathematics and Physics. After a three year experience at NTT Laboratory, he joined Ibaraki University. He had part-time and full-time teaching positions at University of Alabama, University of Tsukuba, University of Electro-Communications, Kansai University and Osaka University. He is now a Professor of Computer Science at University of Canterbury, New Zealand. His research areas are mainly in theoretical computer science in general and design and analysis of algorithms in particular. He served as a guest editor for Algorithmica, the Organinzing Chair and Program Co-chair of ISAAC 2001, and is now on the Editorial Board for the on-line journal, Algorithms.

Yuji Nakagawa is a Professor of the faculty of Informatics of Kansai University. He received his B.E. in Electronics Engineering from the Himeji Institute of Technology, Japan 1973, and M.E. and D. Eng. from Kyoto University, Japan in 1976, 1979. His research interests are in sort and search algorithms, discrete optimization, multi-objective optimization.