Partial Solution and Entropy

Tadao TAKAOKA

Department of Computer Science, University of Canterbury
Christchurch, New Zealand
Correspondence E-mail : tad@cosc.canterbury.ac.nz

Abstract. 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 $S(X) = (X_1, ..., X_k)$, where X is the entire data set, and each X_i is already solved. 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 shortest paths, X_i is an acyclic part in the given graph. For minimum spanning trees, X_i is interpreted as a partially obtained minimum spanning tree for a subgraph. The entropy measure, $H(S)$, is defined by regarding $p_i = |X_i|/|X|$ as a probability measure, that is, $H(S) = -n\Sigma_{i=1}^{k} p_i \log p_i$, where $n = \Sigma_{i=1}^{k}|X_i|$. Then we show that we can sort the input data $S(X)$ in $O(H(S))$ time, and solve the shortest path problem in $O(m + H(S))$ time where m is the number of edges of the graph. Finally we show that the minimum spanning tree is computed in $O(m + H(S))$ time.

Keywords:entropy, complexity, adaptive sort, minimal mergesort, ascending runs, shortest paths, nearly acyclic graphs, minimum spanning trees

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 [7], 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 [10]. 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.

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 $i = 1, , , , N$. 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

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

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

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

Let the state of data be given by a decomposition of a set X as $S(X) = (X_1, ..., X_k)$. 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

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

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 $H(S_0) \geq H(S_1) \geq ... \geq H(S_N)$. We use $-H(S)$ for the potential in equation (1). The accounting equation becomes $a_i = t_i - \Delta H(S_i)$, where $\Delta H(S_i) = H(S_{i-1}) - H(S_i)$. That is, the amortized time is the actual time minus the decrease of entropy at the i-th step. The entropy is regarded as a negative aspect of the data, i.e., the less entropy, the closer to the solution.

Let T and A be the actual total time and the amortized total time. Summing up a_i for $i = 1, ..., N$, we have $A = T + H(S_N) - H(S_0)$, or $T = A + H(S_0) - H(S_N)$. We call this process of summing up amortized times over the computational steps "summing-up". 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 plus the initial entropy. We also have a reasonable assumption that $-\sum_{i=1}^{k} p_i \log p_i > 0$ for the initial state, meaning $H(S_0) = \Omega(n)$.

Let $S'(X) = (X'_1, ..., X'_{k'})$ be a refinement of $S(X) = (X_1, ..., X_k)$, 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 $H(S) \leq H(S')$. As the entropy is a measure of uncertainty, we can say $S(X)$ is more solved than $S'(X)$.

The concept of amortized cost and actual cost is a relative one. That is, the actual time itself may be formulated as amortized time and actual time at a lower level of computation. Specifically, the actual time t_i in the accounting equation can be like $t_{ij} - (\Psi_{i,j-1} - \Psi_{i,j})$, where Ψ is another potential associated with the lower level computation. In such a case, summing-up takes place over indices i and j. Thanks to the linear property of the accounting equation, we can analyze amortized time on the upper level and lower level computation separately. We will see an example of a two-level amortized analysis in Section 5.

In later sections, we show three interpretations of X_i's. In sorting, X_i's are ascending runs which are regarded as solved. In shortest paths, X_i's are acyclic parts of the given graph, which can be processed without the effort of finding the minimum in the priority queue. In the minimum spanning tree problem, X_i is the set of vertices of a partially solved minimum spanning tree for a subgraph.

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

2 Application to adaptive sort

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 [3] 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 $a_1^{(i)} \leq \cdots \leq a_{n_i}^{(i)}$ 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 optimal under the measure of $H(S)$. Hence the measure $H(S)$ derived from runs-entropy 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 the next section is due to [10].

3 Minimal mergesort

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 $|X_{i_j}| \leq |X_{i_{j+1}}|$ $(j = 1, \cdots, k-1)$, 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 $O(n)$ 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. $First(L)$ is the first list in L.

ALGORITHM 1 (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;
 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
    {W is the sorted list}.
```

Lemma 1. *If W_2 is not an original X_i for any i at line 9, it holds that $|W_2| \leq \frac{2}{3}|W|$.*

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_2 or V_1 must have been merged with W_1 or a shorter list, a contradiction. ■

We measure the computing time by the number of key comparisons in the merge operation at line 9, where the straight-forward merging is done with $|W_1|+|W_2|-1$ key comparisons.

Lemma 2. *We slightly modify the definition of amortized time; it is the actual time minus constant times decrease of entropy. Then the amortized time for the i-th merge is not greater than zero.*

Proof. Let $n_i = |X_i|$ for $i = 1, ..., k$. In particular, $|W_1| = n_1$ and $|W_2| = n_2$. The change of entropy occurs only with n_1 and n_2. Thus, noting $n_1 \leq n_2 \leq 2n_1$, the decrease of entropy is

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

Thus

$$a_i = n_1 + n_2 - 1 - \Delta H / \log(3/2) \leq 0$$ ■

Theorem 1. *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 2 and the initial entropy is given by $H(S)$. ■

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 3. *Any sorting algorithm takes at least $\Omega(H(S))$ time when the entropy of ascending runs in $S(X)$ is $H(S)$ and $|X_i| \geq 2$ for $i = 1, \cdots, k$.*

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 different. Let $X_i = (a_1^{(i)}, \cdots, a_{n_i}^{(i)})$. Let $a_{n_i}^{(i)}$ $(i = 1, \cdots, k)$ 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 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 X'. Repeating this calculation yields the number of possibilities N as

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

Since the number of possible permutations is not fewer 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.

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

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

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

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 $T \geq \log(n-2k+2)$. Thus we have $T \geq H(S)/4 = \Omega(H(S))$. ∎

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 al 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 entropy measure. We can define entropy by decomposing the given sequence in non-consecutive portions. Minimal mergesort is not optimal under the entropy measure defined in this way. There are more entropy measures defined in [10].

4 Application to shortest paths for nearly acyclic graphs

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 $|E| = m$. 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 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 by a Fibonacci heap or 2-3 heap [9], we can show the single source shortest path 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. We assume the graph is connected from the source. 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 [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 2 $\{G = (V, E)$ is an acyclic graph.$\}$

1 Topologically sort V and assume without loss of generality); $V = \{v_1, \cdots, v_n\}$
  where $(v_i, v_j) \in E \Leftrightarrow i < j$;
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 4. *At the beginning of Line 5 in Algorithm 2, 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 [1] 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 3 {*Single source shortest paths with v_0 being the source*} *[1], [8]*

```
 1  for v ∈ V do if v = v_0 then d[v] := 0 else d[v] := ∞;
 2  Organize V in a priority queue Q with d[v] as key;
 3  S := ∅;
 4  while S ≠ 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 ∪ {v} ;
11    for w ∈ OUT(v) ∩ (V − S) do d[w] := min{d[w], d[v] + c(v, w)}
12  end.
```

It is shown in [1] 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 single source shortest path 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 $v_1, , , , , v_k$ 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, ..., 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. Note that the source is chosen by the first find-min. Then $S(V) = (V_1, ..., V_t)$ forms a decomposition of the set

V. We denote the entropy of this decomposition by $H(S)$. Lemma 6 in the next section shows that t find-min operations with $|V_1|+...+|V_t|$ deletes interleaved in Algorithm 3 (each i-th find-min followed by $|V_i|$ deletes) can be done in $O(H(S))$ 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 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 4. The time for building Q at line 2 is absorbed in $O(m)$. Thus we have the following theorem.

Theorem 2. *Algorithm 3 solves the single source shortest path problem in $O(m+H(S))$ time.*

5 Analysis of delete operations

We maintain the priority queue for the single source shortest path problem by a 2-3 heap [9]. In traditional priority queues, decrease-key, insert and delete-min 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 reshape of the work space. After destroying v, we merge the subtrees of v at the root level. The amortized time for a delete is proportional to the number of children, which is $O(\log n_v)$, where n_v is the number of descendants of node v to be deleted. A delete is defined on a Fibonacci heap in [1].

Let us delete nodes $v_j (j = 1, ..., k)$ in the batch mode from a 2-3 heap of size n. In the batch mode, we disconnect all children of all v_j and merge them at the root level. In other words, we do not process v_j one by one. Assume the number of descendants of v_j is n_j. The total amortized time T of deleting $v_1, ..., v_k$ in the batch mode is $T = O(\log n_1 + ... + \log n_k)$. Noting that $n_1 + ... + n_k \leq cn$ for some constant c, T is maximized as $T = O(k \log(n/k))$ when $n_1 = ... = n_k = cn/k$. Thus

Lemma 5. *k consecutive delete operations on a 2-3 heap of size n can be done in $O(k \log(n/k))$ 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 $T_i = O(k_i \log(n/k_i))$ by Lemma 5, we have the total time for all deletes bounded within a constant factor by

$k_1 \log(n/k_1) + ... + k_t \log(n/k_t) = n(\Sigma_{i=1}^{t}(k_i/n) \log(k_i/n)) = n(-\Sigma_{i=1}^{t} p_i \log p_i)$, 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 $O(t \log n + H(S))$, which is further simplified to $O(H(S))$ 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 $O(H(S))$.*

Proof. $H(S)$ is minimum when $k_1 = ... = k_{t-1} = 1$, 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. In terms of amortized analysis in Section 1, we can define a_i and t_i in the following way. Let $V^{(s)} = V_s \cup ... \cup V_t$. We define the state of data set, S_s, to be the data set $V^{(s)}$ decomposed as above. The initial state $S = S_1$ is given by $V_1 \cup ... \cup V_t$. The entropy of the state of data is defined for the beginning of the s-th find-min at line 8 using $V^{(s)}$ by

$$H(S_s) = \Sigma_{i=s}^{t} |V_i| \log(|V^{(s)}|/|V_i|)$$

Noting that $|V^{(s)}| \geq |V^{(s+1)}|$, the decrease of entropy at the beginning of the next find-min operation is

$$\begin{aligned} \Delta H(S_{s+1}) &= \Sigma_{i=s}^{t} |V_i| \log(|V^{(s)}|/|V_i|) - \Sigma_{i=s+1}^{t} |V_i| \log(|V^{(s+1)}|/|V_i|) \\ &\geq |V_s| \log(|V^{(s)}|/|V_s|) \end{aligned}$$

We define the actual time and amortized time to be those for the stage from the s-th find-min to the $(s+1)$-th. The actual time t_s for stage s is given by

$$t_s = O(m_s + |V_s| \log(|V^{(s)}|/|V_s|) + \log n).$$

This is because we inspect $O(m_s)$ edges, perform $O(V_s)$ deletes in the heap of size $|V^{(s)}|$, and spend $O(\log n)$ time for a find-min. We interpret the above formula as $t_s \leq c_s(m_s + |V_s| \log(|V^{(s)}|/|V_s|) + \log n)$ with some constant $c_s > 0$.

The amortized time for the s-th stage is slightly modified with constant c_s, and given as

$$\begin{aligned} &a_s = t_s - c_s \Delta H(S_{s+1}) \\ &\leq c_s(m_s + |V_s| \log(|V^{(s)}|/|V_s|) + \log n) - c_s |V_s| \log(|V^{(s)}|/|V_s|) \\ &\leq c_s(m_s + \log n) \end{aligned}$$

Using constant $c = max\{c_s\}$ and noting $H(S_t) = 0$, we have

$$T \leq A + c(H(S_1) - H(S_t)) \leq O(m + t \log n + H(S_1)) = O(m + H(S))$$

Note that when we perform summing-up over a_s, we also perform summing-up over actual and amortized times for operations on the 2-3 heap. In this sense, the above is a "two-level" amortized analysis.

Remark. In [1], $O(t \log n + H(S))$ is bounded by $O(n \log t)$. Thus our analysis of $O(t \log n + H(S)) \leq O(H(S))$ is sharper.

6 Relationship with 1-dominator

As a definition of near-acyclicity, the definition and algorithm for a 1-dominator decomposition is given in [8]. The decomposition is given by the set of disjoint sets, called 1-dominator sets, whose union is V. A 1-dominator set 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. Let $IN(v) = \{u|(u, v) \in E\}$. 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 [8] 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 1-dominator set are finalized through Algorithm 2 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 single source 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 4 is bounded by the entropy defined by the 1-dominator decomposition.

Theorem 3. *The decomposition by 1-dominator sets is a refinement of the decomposition defined by Algorithm 3.*

Proof. Suppose a vertex v is obtained by find-min at line 8 and v is not a trigger. Then v is inside some 1-dominator set. Since the distance to the corresponding trigger is smaller, the trigger must be 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 1-dominator set is subsequently deleted from the heap, and possibly more 1-dominator sets. Thus the 1-dominator decomposition is a refinemment of the decomposition $S(V)$. ■

The decomposition by Algorithm 3 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 3.

In [8], the single source 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, 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 3. However Algorithm 3 and the single source algorithm in [8] are not incompatible. We can run Algorithm 3 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.

7 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 $i = 1, ..., k$. The state of data $S(V)$ is defined by $S(V) = (V_1, ..., V_k)$, 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 4 {*To complete the MST problem*}
1 Let the sorted edge list L has been partially scanned
2 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 $i = 1, ..., k$
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 2. 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

$$\begin{aligned} &\Delta H(S_i) = n_1 \log(n/n_1) + n_2 \log(n/n_2) - (n_1 + n_2) \log(n/(n_1 + n_2)) \\ &= n_1 \log(1 + n_2/n_1) + n_2 \log(1 + n_1/n_2) \geq min\{n_1, n_2\} \end{aligned}$$

Noting that $t_i = min\{n_1, n_2\}$, amortized time becomes

$$a_i = t_i - \Delta H(S_i) \leq 0$$

The rest of work is bounded by $O(m)$. Thus the total time for the remaining work becomes $O(m + H(S))$, where $H(S)$ is the initial entropy at the beginning of line 4. Note that the condition for $a_i \leq 0$ is crucial. In the analysis of minimal merge sort, it is satisfied by the fact that the two shortest ascending runs are merged, whereas in the MST problem in this section, it is satisfied by merging the smaller tree to the larger tree.

8 Concluding Remarks

We captured computation as a process of reducing entropy, starting from some positive value and ending in zero. The amortized time for a step of the computation is the sum of 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 entropy will be useful for analysis. We showed that three specific problems of sorting, shortest paths and minimum spanning trees can be analyzed by this unified entropy analysis.

If the computation process is a merging process of two sets in the decomposition, 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.

Acknowledgment The author gratefully acknowledges many constructive comments given by the reviewers. This work was partially done at Kansai University.

References

1. Abuaiadh, D. and J.H. Kingston, Are Fibonacci heaps optimal? ISAAC'94, LNCS 834, pp. 442–450 (1994)
2. Dijkstra, E.W., A note on two problems in connection with graphs, Numer. Math. Vol. 1, pp. 269–271 (1959)
3. V. Estivill-Castro and D. Wood, A survey of adaptive sorting algorithms, *ACM Computing Surveys* 24, 441–476 (1992)
4. Fredman, M.L. and R.E. Tarjan, Fibonacci heaps and their use in improved network optimization problems, JACM, Vol. 34, No. 3, 596–615 (1987)
5. D. E. Knuth, "The Art of Computer Programming, Vol.3, Sorting and Searching," Addison-Wesley, Reading, Mass. (1974)
6. H. Mannila, Measures of presortedness and optimal sorting algorithms, *IEEE Trans. Comput. C-34*, 318–325 (1985)
7. Y. Nakagawa, A Difficulty Estimation Method for Multidimensional Nonlinear 0-1 Knapsack Problem Using Entropy, Transactions of the Institute of Electronics, Communication and Information, Vol. J87-A, No. 3, 406–408 (2004)
8. S. Saunders and T. Takaoka, Solving shortest paths efficiently on nearly acyclic directed graphs, Theoretical Computer Science, 370(1-3), 94-109 (2007)
9. T. Takaoka, Theory of 2-3 Heaps, Discrete Applied Math, Vol. 126, 115-128 (2003)
10. T. Takaoka, Entropy – Measure of Disorder, Proc. CATS (Computation: Australasian Theory Symposium), 77-85 (1998)
11. Tarjan, R.E., Data Structures and Network Algorithms, Regional Conference Series in Applied math. 44 (1983)