On Finding a Hamiltonian Path in a Tournament Using Semi-Heap (Part 1: Sequential Solution) *

Jie Wu

Department of Computer Science and Engineering

Florida Atlantic University

Boca Raton, FL 33431

jie@cse.fau.edu

Abstract

The problem of sorting an intransitive total ordered set, a generalization of regular sorting, is considered. This generalized sorting is based on the fact that there exists a special linear ordering (also called a generalized sorting sequence) for any intransitive total ordered set, or equivalently, the existence of a Hamiltonian path in a tournament. A new data structure called semi-heap is proposed to construct an optimal $\Theta(n \log n)$ sorting algorithm. In [7] we propose a cost-optimal parallel algorithm using semi-heap. The run time of this algorithm is $\Theta(n)$ with $\Theta(\log n)$ processors under the EREW PRAM model.

Key words: Data structure, directed graph, Hamiltonian path, heap, sorting, total order

*This work was supported in part by NSF grants CCR 9900646 and ANI 0073736. A preliminary version of this paper appeared in the Proceedings of the 14th International Parallel & Distributed Processing Symposium.

1 Introduction

Sorting is one of the fundamental problems in computer science for which different solutions have been proposed [4]. Given a sequence of n numbers $(n_1, n_2, ..., n_n)$ as an input, a sorting algorithm generates a permutation (reordering) $(n_1^{'}, n_2^{'}, ..., n_n^{'})$ of the input sequence such that $n_1^{'} \geq n_2^{'} \geq ... \geq n_n^{'}$.

We consider a generalization of the sorting problem by replacing $\geq$ with $\succ$, where $\succ$ is a total order without the transitive property, i.e., it is intransitive. That is, if $n_i \succ n_j$ and $n_j \succ n_k$, it is not necessary that $n_i \succ n_k$. The total order requires that for any two elements n_i and n_j, either $n_i \succ n_j$ or $n_j \succ n_i$, but not both (antisymmetric).

The set N of n elements exhibiting intransitive total order can be represented by a directed graph, where $n_i \succ n_j$ represents a directed edge from vertex n_i to vertex n_j. The underlying graph is a complete graph. This graph is also called a *tournament* [1], representing a tournament of n players where every possible pair of players plays one game to decide the winner (and the loser) between them. Sorting on N corresponds to finding a Hamiltonian path (also called a generalized sorting sequence, or simply, a sorting sequence) in the tournament. The existence of a Hamiltonian path in any tournament was first proved in [5]. Other properties related to tournament can be found in [6].

Hell and Rosenfeld [3] proved that the bound of finding a Hamiltonian path is $\Theta(n \log n)$, the same bound as the regular sorting. They also considered bounds on finding some generalized Hamiltonian paths. It is easy to prove that many regular sorting algorithms can be used to find a Hamiltonian path in a tournament, such as bubble sort, insertion sort, binary insertion sort, and merge sort.

In this paper, we propose a new data structure called *semi-heap*, which is an extension of a regular heap structure. We introduce an optimal $\Theta(n \log n)$ algorithm to determine a Hamiltonian path in a tournament based on the semi-heap structure. In [7], we propose a cost-optimal parallel algorithm based on the semi-heap structure that takes $\Theta(n)$ in run time using $\Theta(\log n)$ processors in the EREW PRAM model. An implementation of the cost-optimal parallel algorithm in the network model with a linear order of processors is also shown.

The rest of the paper is organized as follows: Section 2 shows a constructive proof of the existence of a Hamiltonian path in any given tournament, and then, proposes the semi-heap structure. Section 3 demonstrates why the regular heapsort cannot be directly applied to the semi-heap structure, and then, presents an optimal generalized sorting algorithm using semi-heap. Section 4

summaries our results.

2 Semi-Heap Data Structure

In this section, we first show the existence of a Hamiltonian path in any given tournament, and then, propose the semi-heap data structure. Unlike proofs presented in many textbooks of graph theory, we provide a constructive proof which serves as the base for insertion sort.

Proposition [5]: *Consider a set N ($|N| = n$) with any two elements n_i and n_j, either $n_i \succ n_j$ or $n_j \succ n_i$, but not both. Then elements in N can be arranged in a linear order $n_1^{'} \succ n_2^{'} \succ ... \succ n_{n-1}^{'} \succ n_n^{'}$.*

Proof: We prove this theorem by induction. When $n = 1$, the result is obvious. Assume that the theorem holds for $n = k$, i.e.,

$$n_1^{'} \succ n_2^{'} \succ ... \succ n_k^{'}$$

When $n = k + 1$, any k elements can be arranged in a linear order as above. We then insert the $(k+1)$th element $n_{k+1}^{'}$ in front of $n_i^{'}$, where i is the largest index such that $n_{k+1}^{'} \succ n_i^{'}$. That is,

$$n_1^{'} \succ n_2^{'} \succ ... \succ n_{k+1}^{'} \succ n_i^{'} ... \succ n_k^{'}$$

If such an index i does not exist, $n_{k+1}^{'}$ is placed as the last element in the linear order:

$$n_1^{'} \succ n_2^{'} \succ ... \succ n_k^{'} \succ n_{k+1}^{'}$$

■

The proposition states that a Hamiltonian path may exist in any given tournament, but not necessary for a Hamiltonian circle. That is, we can always arrange n players in a linear order from left to right such that each player beats the one to its right. Figure 1 shows a directed graph with five vertices. One sorting sequence is $n_3 \succ n_4 \succ n_2 \succ n_5 \succ n_1$. When $\succ$ is transitive, the sorting sequence arrangement is reduced to a regular sorting problem. Unlike the regular sorting problem, more than one solution may exist for the generalized sorting problem. For example, $n_1 \succ n_3 \succ n_2 \succ n_5 \succ n_4$ is another sorting sequence for the example of Figure 1. The insertion sort with a complexity of $\Theta(n^2)$ can be easily constructed based on the above proof. In the following, we propose the semi-heap data structure, and then, present a sorting algorithm with a complexity of $\Theta(n \log n)$ based on the semi-heap.

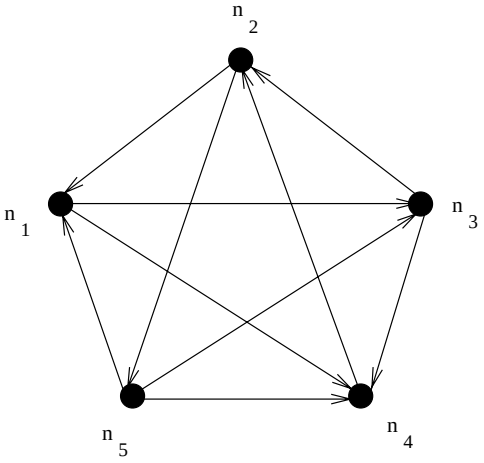

Figure 1: A directed graph with a complete underlying graph.

Consider three elements n_1, n_2, n_3 in N, denote $n_1 = \max\{n_1, n_2, n_3\}$ if $n_1 \succ n_2$ and $n_1 \succ n_3$. Note that in a total order without the transitive property, the maximum element may not exist among three elements. For example, if $n_1 \succ n_2$, $n_2 \succ n_3$, and $n_3 \succ n_1$, $\max\{n_1, n_2, n_3\}$ does not exist. Next we introduce a new concept of the maximum element based on $\succ$.

Definition 1: $n_1 = \max_{\succ}\{n_1, n_2, n_3\}$ *if both* $n_2 = \max\{n_1, n_2, n_3\}$ *and* $n_3 = \max\{n_1, n_2, n_3\}$ *are false.*

Note that when $n_i = \max\{n_1, n_2, n_3\}$ are false for all $i = 1, 2, 3$, every n_i is a maximum element.

A *semi-heap* is any array object that can be viewed as a complete binary tree, like a regular heap. A complete binary tree of height h is a binary tree that is full down to level $h-1$, with level h filled in from left to right. However, the regular heap property is changed. Let $L(n^{'})$ and $R(n^{'})$ represent left and right child nodes of $n^{'}$, respectively. When a child, say $R(n^{'})$, does not exist, the relation $n^{'} \succ R(n^{'})$ automatically holds.

Definition 2: *A semi-heap for a given intransitive total order* $\succ$ *is a complete binary tree. For every node* $n^{'}$ *in the tree,* $n^{'} = \max_{\succ}\{n^{'}, L(n^{'}), R(n^{'})\}$.

When an array A is used to represent a semi-heap, $l(i)$ and $r(i)$ are used as indices of the left and right child nodes of i; they can be computed simply by $l(i) = 2i$ and $r(i) = 2i+1$. Figure 2 (a) shows a semi-heap with 10 elements. A semi-heap can be viewed as a set of overlapping triangles, with each triangle consisting of $A[i]$, $A[l(i)]$, $A[r(i)]$. Figure 3 shows four possible configurations

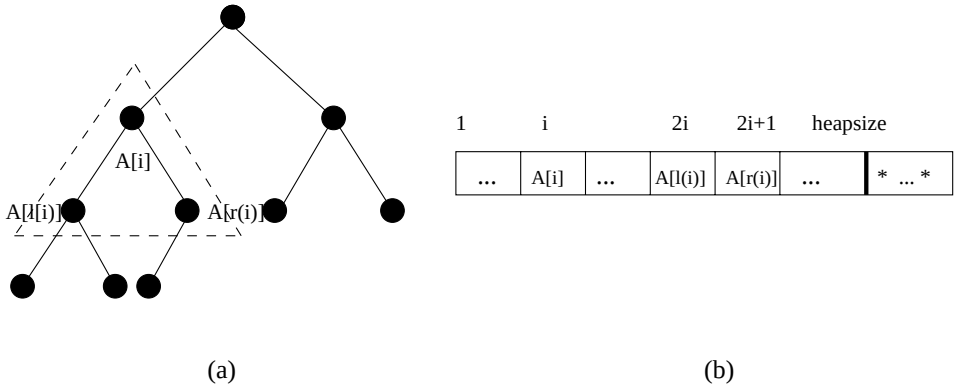

Figure 2: A semi-heap structure as a set of overlapping triangles.

of a triangle under relation $\succ$. In this figure, if $A[i] \succ A[l(i)]$ is true, a directed edge is drawn from $A[i]$ to $A[l(i)]$. Note that $A[i] = \max_{\succ}\{A[i], A[l(i)], A[r(i)]\}$ for all cases. In cases (a) and (b) condition $A[i] = \max\{A[i], A[l(i)], A[r(i)]\}$ also holds.

To simplify the presentation, we fill in a special symbol $*$ representing a smaller value than any one in the semi-heap for entries that are outside the semi-heap. That is, $A[i] \succ A[j]$ is true for any i inside the semi-heap and any j outside the semi-heap. Specifically, $A[i]$ is an element of the semi-heap if $1 \leq i \leq \mathit{heapsize}$ (see Figure 2 (b)). $A[j]$ is an element outside the semi-heap if $j > \mathit{heapsize}$.

3 Generalized Sorting Using Semi-Heap

Although a semi-heap resembles a heap, the traditional heapsort algorithm cannot be directly applied to a semi-heap to generate a generalized sorted sequence. Recall that with the transitive property, root $A[1]$ of the heap is always the maximum element in the heap, i.e., the player at the root "beats" all the other players in the tournament. When we "discard" the root, it is "replaced" by the last element $A[n]$ in the heap, and then, the heap is reconstructed by pushing $A[n]$ down in the heap so that the new root is the maximum element among the remaining elements. However, in a semi-heap, we may face a situation in which $A[n]$ beats all $A[1]$, $A[2]$, and $A[3]$, which is an impossible situation in a regular heap. $A[n]$, the new root, cannot be selected (and be removed from the semi-heap) in the next round to be placed after $A[1]$, the previously selected element, because $A[n]$ beats $A[1]$. On the other hand, because $A[n]$ beats $A[2]$, its left child, and $A[3]$, its

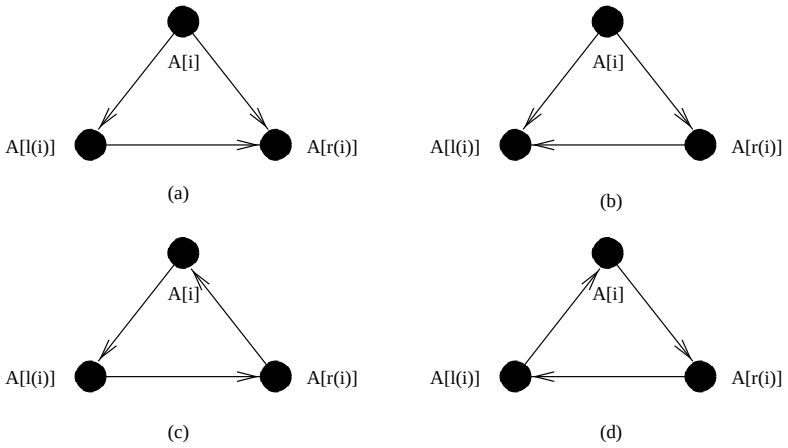

Figure 3: Four possible configurations of a triangle in a semi-heap.

right child, $A[n]$ cannot be pushed down in the semi-heap. Therefore, a different strategy has to be developed for semi-heap.

We follow closely the notation used in Cormen, Leiserson, and Rivest's book [2]. The sorting using semi-heap consists of four modules: SEMI-HEAPIFY(A, i), BUILD-SEMI-HEAP(A), REPLACE(A, i), and SEMI-HEAP-SORT(A). SEMI-HEAPIFY(A, i) constructs a semi-heap rooted at $A[i]$, provided that binary trees rooted at $A[l(i)]$ and $A[r(i)]$ are semi-heaps (see Figure 3). The cost of SEMI-HEAPIFY is the height of node $A[i]$, measured by the number of edges on the longest simple downward path from the node to a leaf. That is, the cost of SEMI-HEAPIFY is $\Theta(\log n)$, where $n = heapsize$. BUILD-SEMI-HEAP uses the procedure SEMI-HEAPIFY in a bottom-up manner to convert an arbitrary array A into a semi-heap. The cost of BUILD-SEMI-HEAP is $\Theta(n)$, which is the same cost of building a regular heap.

Generalized sorting is done through SEMI-HEAP-SORT by repeatly printing and removing the root of the binary tree (which is initially a semi-heap). The root is replaced by either its leftchild or rightchild through REPLACE. The selected child is replaced by one of its child nodes. The process continues until reaching one of the leaf nodes and the entry for that leaf node is replaced by $*$, i.e., that leaf node is removed from the tree. A new tree derived is no longer a semi-heap; however, each overlapping triangle in the tree still meets the maximum element requirement in Definition 2. The cost of REPLACE is the height of the current tree, which is bounded by the height of the original semi-heap, $\Theta(\log n)$. Therefore, the cost of SEMI-HEAP-SORT is $\Theta(n \log n)$. Without loss of generality, we assume that $n \geq 1$.

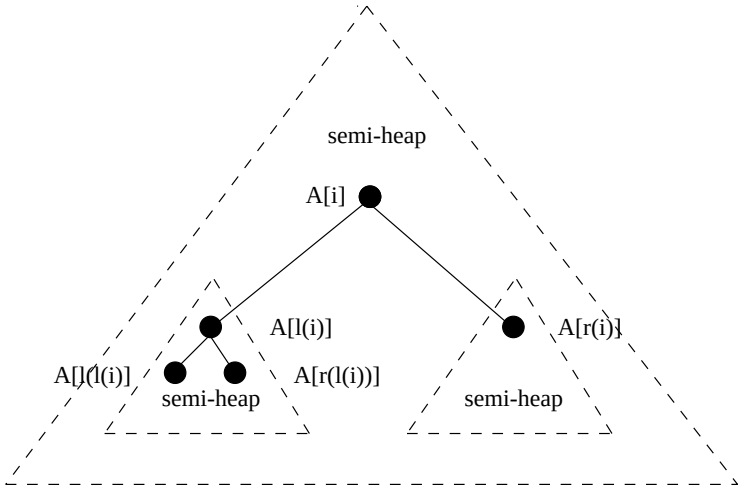

Figure 4: The construction of a semi-heap using SEMI-HEAPIFY.

SEMI-HEAPIFY(A, i)

```
1  if A[i] ≠ max_≻{A[i], A[l(i)], A[r(i)]}
2    then find winner such that A[winner] ⟵ max{A[i], A[l(i)], A[r(i)]}
3         exchange A[i] ⟷ A[winner]
4         SEMI-HEAPIFY(A, winner)
```

BUILD-SEMI-HEAP(A)

```
1  for  i ⟵ ⌊heapsize/2⌋ downto 1
2       do SEMI-HEAPIFY(A, i)
```

REPLACE(A, i)

```
1  if  (A[l(i)] = *) ∧ (A[r(i)] = *)
2    then A[i] ⟵ *
3    else  if (A[i] ≻ A[l(i)]) ∧ (A[l(i)] ≻ A[r(i)])
4            then A[i] ⟵ A[l(i)]
5                 REPLACE(A, l[i])
6            else  A[i] ⟵ A[r(i)]
7                 REPLACE(A, r[i])
```

SEMI-HEAP-SORT(A)

```
1 BUILD-SEMI-HEAP(A)
2 while (A[l(1)] ≠ *) ∨ (A[r(1)] ≠ *)
3       do print(A[1])
4          REPLACE(A, 1)
5 print(A[1])
```

Theorem 1: *BUILD-SEMI-HEAP constructs a semi-heap for any given complete binary tree.*

Proof: The procedure BUILD-SEMI-HEAP goes through nodes that have at least one child node and runs SEMI-HEAPIFY on these nodes. The order in which these nodes are processed guarantees that the subtrees rooted at child nodes of $A[i]$ are semi-heap before SEMI-HEAPIFY runs at $A[i]$.

When SEMI-HEAPIFY is called at $A[i]$, if $A[i]$ is the maximum element among $A[i]$, $A[l(i)]$, and $A[r(i)]$ based on $\succ$, the binary tree rooted at $A[i]$ is automatically a semi-heap. Otherwise and without loss of generality, one of the child nodes, say $A[l(i)]$, is the winner among the three, i.e., $A[l(i)]$ beats both $A[i]$ and $A[r(i)]$. In this case, $A[l(i)]$ is swapped with $A[i]$, which ensures that node $A[i]$ and its child nodes satisfy the semi-heap property. However, node $A[l(i)]$ now has the original $A[i]$, and thus, the subtree rooted at $A[l(i)]$ may violate the semi-heap property. Therefore, SEMI-HEAPIFY must be called recursively on that subtree.

A new problem (that does not appear in the original heap structure) is how to ensure that the resultant root $A[l(i)]$, after applying SEMI-HEAPIFY at $A[l(i)]$, will not violate the semi-heap property among $A[i]$, $A[l(i)]$, and $A[r(i)]$. In a regular heap, $A[i]$ is the maximum element in the tree rooted at $A[i]$, the heap property among $A[i]$, $A[l(i)]$, and $A[r(i)]$ automatically holds. In a semi-heap, we need to prove that the newly selected root $A[l(i)]$ (other than the original value $A[i]$), which is either $A[l(l(i))]$ or $A[r(l(i))]$ in the original tree, cannot beat both $A[i]$ (the original $A[l(i)]$) and $A[r(i)]$. In fact, we prove that $A[i]$ (the original $A[l(i)]$) always beats the newly selected $A[l(i)]$ (the original $A[l(l(i))]$ or $A[r(l(i))]$). We consider the following two cases in the original tree with a semi-heap rooted at $A[l(i)]$ (see Figure 4):

- If $A[l(i)]$ beats both $A[l(l(i))]$ and $A[r(l(i))]$. The problem is solved because in the resultant tree node $A[l(i)]$ becomes $A[i]$ and either $A[l(l(i))]$ or $A[r(l(i))]$ becomes $A[l(i)]$.

- If $A[l(i)]$ beats only one child node, then without loss of generality, we assume that $A[l(i)]$ (which is now $A[i]$) beats $A[l(l(i))]$, $A[l(l(i))]$ beats $A[r(l(i))]$, and $A[r(l(i))]$ beats $A[l(i)]$. To select a winner among the original $A[i]$ (now $A[l(i)]$), $A[l(l(i))]$, $A[r(l(i))]$, other than $A[l(i)]$, $A[l(l(i))]$ is the only choice (since $A[r(l(i))]$ has lost to $A[l(l(i))]$). Consequently, $A[l(l(i))]$

becomes the newly selected root of the left subtree of $A[i]$, based on the assumption, $A[i]$ (the original $A[l(i)]$) beats $A[l(i)]$ (the original $A[l(l(i))]$) in the resultant tree. ■

Consider a complete binary tree with eight vertices, i.e., $heapsize = 8$. The initial configuration of array A is n_1, n_2, n_3, n_4, n_5, n_6, n_7, and n_8. The tournament is represented by an 8×8 matrix M given below, where $M[i,j] = 1$ if n_i beats n_j (i.e., $n_i \succ n_j$) and $M[i,j] = 0$ if n_i is beaten by n_j (i.e., $n_j \succ n_i$). $M[i,i] = -$ represents an impossible situation. Note that $M[i,j] = 1$ if and only if $M[j,i] = 0$.

$$M = \begin{pmatrix} - & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 1 & - & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & - & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & - & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & - & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & - & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & - & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & - \end{pmatrix}$$

Figure 5 (a) shows the initial configuration of this complete binary tree in array A, where the corresponding tree structure is represented by a set of overlapping triangles. Three edges among three vertices in each triangle represent tournament results between three pairs of players in the triangle. That is, an edge directed from n_i to n_j exists if $M[i,j] = 1$ in matrix M. Relationships between two vertices from different triangles are not shown in the figure. Figure 5 (b) shows the resultant semi-heap after applying BUILD-SEMI-HEAP. $A[j]$ is filled with $*$ for $j \geq 8$. Actually, it is sufficient to define the size of A to be $2 \times heapsize$. A step-by-step application of REPLACE$(A, 1)$ to the example of Figure 5 is shown in Figure 6, where the selected (printed) elements are placed beside the root in a left-to-right order. In this example, the final output sequence is $n_1 \succ n_7 \succ n_3 \succ n_2 \succ n_4 \succ n_5 \succ n_8 \succ n_6$. Once all elements are printed, all entries in array A are filled with $*$. The correctness of this result can be easily verified through the given matrix M.

Note that although the REPLACE process destroys the semi-heap structure (since the resultant tree is no longer a complete binary tree), each overlapping triangle in the corresponding binary tree still maintains one of the four possible configurations of a semi-heap as shown in Figure 3. Therefore,

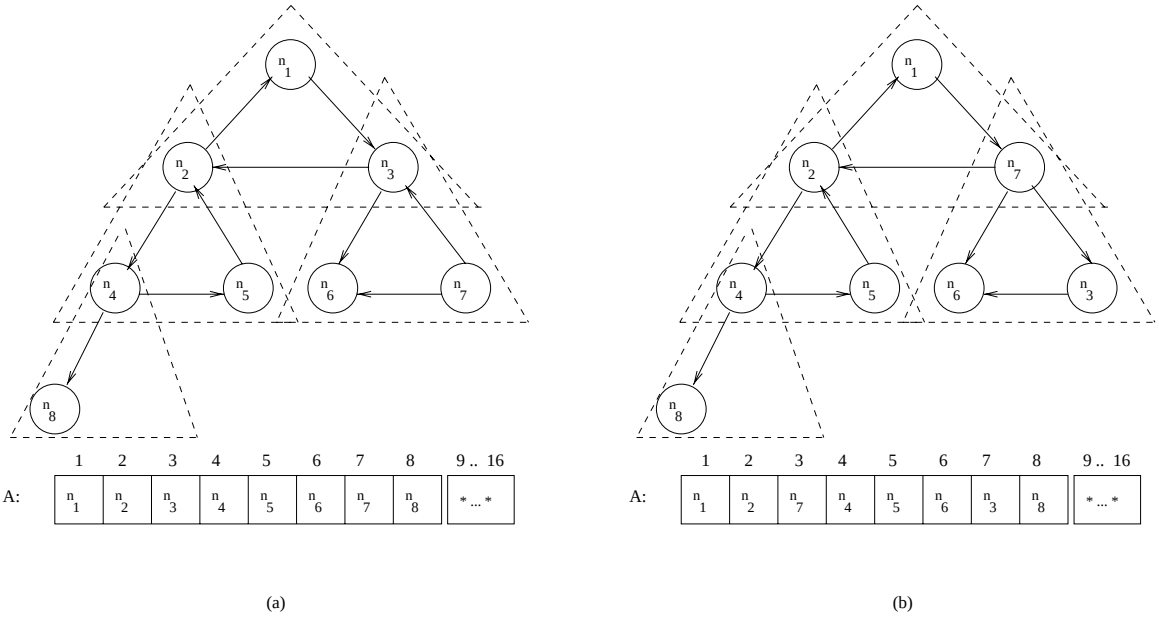

Figure 5: An example tree: (a) the initial configuration, (b) the semi-heap configuration, after applying BUILD-SEMI-HEAP.

it always generates a generalized sorted sequence for any given semi-heap.

Theorem 2: *For any given semi-heap, SEMI-HEAP-SORT generates a generalized sorted sequence.*

Proof: It suffices to show that REPLACE always replaces the current root by an element beaten by the root. In addition, each overlapping triangle in the binary tree is still one of the four possible configurations of a triangle in a semi-heap, i.e., the root of each triangle is the maximum element based on $\succ$ in the triangle. Based on the definition of REPLACE, the current root $A[i]$ is replaced by $A[l(i)]$ for cases (a) and (c) and by $A[r(i)]$ for cases (b) and (d) of Figure 3. The replacing element, say $A[l(i)]$, is itself replaced by an element in the triangle rooted at $A[l(i)]$. This process continues iteratively down the semi-heap. In addition, the new root $A[i]$ beats both of its child nodes (if any). This property ensures when a child node is missing (i.e., the corresponding triangle contains only two nodes), $A[i]$ can still be replaced by another child node without causing any problem. Therefore, the root of each triangle is still the maximum element based on $\succ$ in the triangle. ■

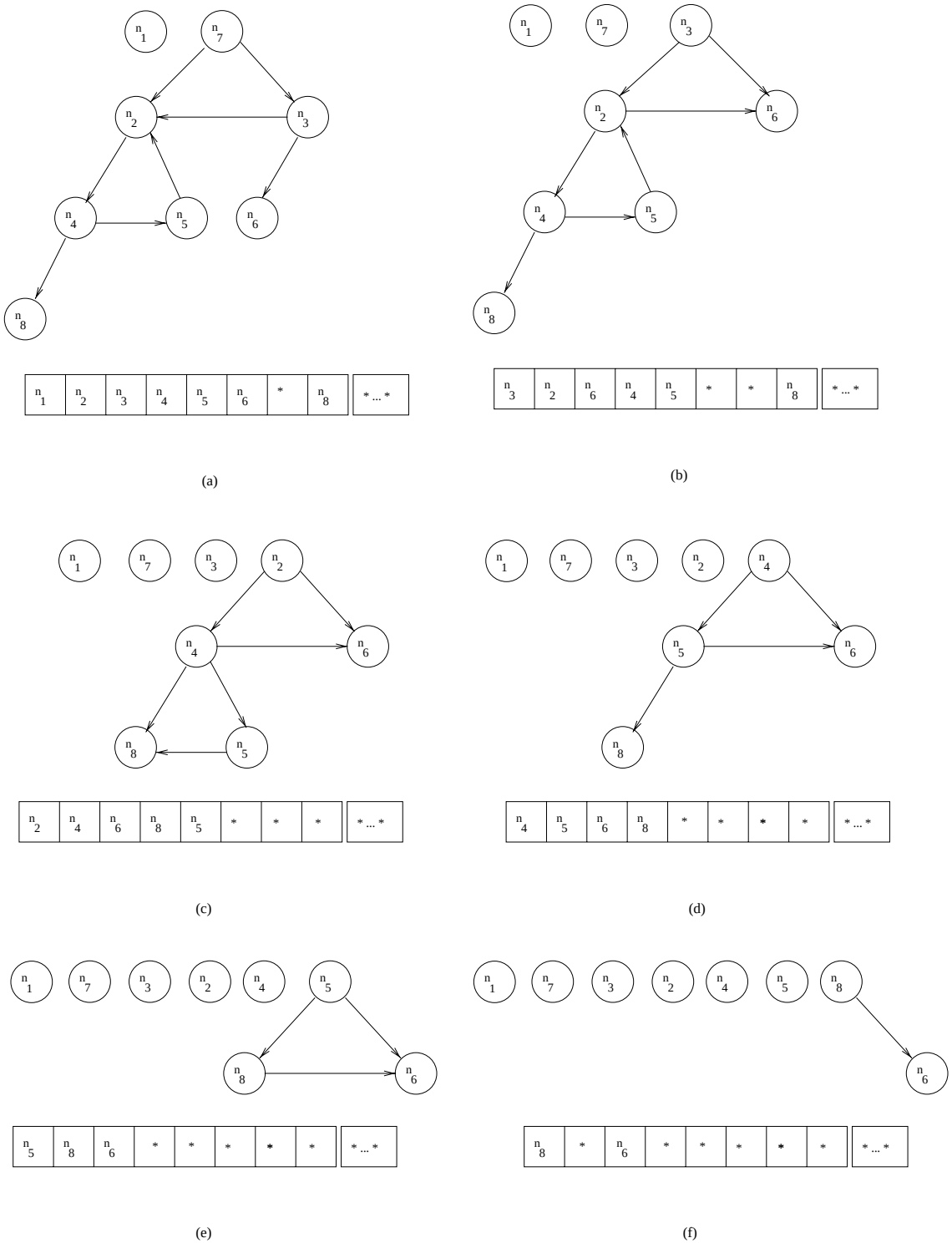

Figure 6: A step-by-step application of REPLACE(A, i) in the example of Figure 5.

4 Conclusions

We have proposed a data structure called semi-heap which is a generalization of the traditional heap structure. The semi-heap structure is used to solve a generalized sorting problem. We have shown that the generalized sorting problem can be solved optimally using semi-heap. In [7] we show a parallel solution based on the semi-heap structure.

References

[1] J. A. Bondy and U.S.R. Murthy. *Graph Theory and Applications.* The Macmillan Press. 1976.

[2] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press. 1994.

[3] P. Hell and M. Rosenfeld. The complexity of finding generalized paths in tournaments. *Journal of Algorithms.* 1983, 4, 303-309.

[4] D. Knuth. *The Art of Computer Programming, Vol 3, Sorting and Searching.* Addison-Wesley Publishing Company, second edition. 1998.

[5] H. G. Landau. On dominance relations and the structure of animal societies, III: The condition for score structure. *Bull. Math. Biophys.* 15, 1953, 143-148.

[6] D. B. West. *Introduction to Graph Theory.* Prentice Hall, Inc. 1996.

[7] J. Wu. On finding a Hamiltonian path in a tournament using semi-heap (Part 2: Parallel solution).