Algorithm A.2 Maximum weight spanning tree in an undirected graph

Procedure Max-Weight-Spanning-Tree (
$\mathcal{H} = (\mathcal{N}, \mathcal{E})$
$\{w_{ij} \ : \ (X_i, X_j) \in \mathcal{E}\}$
)
$\mathcal{N}_T \leftarrow \{X_1\}$
$\mathcal{E}_T \leftarrow \emptyset$
while $\mathcal{N}_T \neq \mathcal{X}$
$\mathcal{E}' \leftarrow \{(i,j) \in \mathcal{E} \ : \ X_i \in \mathcal{N}_T, X_j \notin \mathcal{N}_T\}$
$(X_i, X_j) \leftarrow \arg\max_{(X_i,X_j)\in\mathcal{E}'} w_{ij}$
// (X_i, X_j) is the highest-weight edge between a node in T and a node out of T
$\mathcal{N}_T \leftarrow \mathcal{N}_T \cup \{X_j\}$
$\mathcal{E}_T \leftarrow \mathcal{E}_T \cup \{(X_i, X_j)\}$
return $(\mathcal{E}_T)$

A.3.2 Analysis of Algorithmic Complexity

A key step in evaluating the usefulness of an algorithm is to analyze its computational cost: the amount of time it takes to complete the computation and the amount of space (memory) required. To evaluate the algorithm, we are usually not interested in the cost for a particular input, but rather in the algorithm's performance over a set of inputs. Of course, we would expect most algorithms to run longer when applied to larger problems. Thus, the complexity of an algorithm is usually measured in terms of its performance, as a function of the size of the input given to it. Of course, to determine the precise cost of the algorithm, we need to know exactly how it is implemented and even which machine it will be run on. However, we can often determine the scalability of an algorithm at a more abstract level, without worrying about the details of its implementation. We now provide a high-level overview of some of the basic concepts underlying such analysis.

Consider an algorithm that takes a list of n numbers and adds them together to compute their sum. Assuming the algorithm simply traverses the list and computes the sum as it goes along, it has to perform some fixed number of basic operations for each element in the list. The precise operations depend on the implementation: we might follow a pointer in a linked list, or simply increment a counter in an array. Thus, the precise cost might vary based on the implementation. But, the total number of operations per list element is some fixed constant factor. Thus, for any reasonable implementation, the running time of the algorithm will be bounded by $C \cdot n$ for some constant C. In this case, we say that the *asymptotic complexity* of the algorithm is $O(n)$, where the $O()$ notation makes implicit the precise nature of the constant factor, which can vary from one implementation to another. This idea only makes sense if we consider the running time as a function of n. For any fixed problem size, say up to 100, we can always find a constant C (for instance, a million years) such that the algorithm takes time no more than C. However, even if we are not interested in problems of unbounded size, evaluating the way in which the running time varies as a function of the problem size is the first step to understanding how well it will

asymptotic complexity