Part II

Binary Searching in Trees

6
Introduction

Searching in ordered structures is a fundamental problem in theoretical computer science. In one of its most basic variants, the objective is to find a special element of a totally ordered set by making queries which iteratively narrow the possible locations of the desired element. This can be generalized to searching in more general structures which have only a partial order for their elements instead of a total order [CDK+04, LA95, BFN99b, OP06, MOW08].

In this work, we focus on searching in structures that lay between totally ordered sets and the most general posets: we wish to efficiently locate a particular node in a tree. More formally, as input we are given a tree $T = (V, E)$ which has a 'hidden' *marked* node and a function $w : V \to \mathbb{R}$ that gives the likelihood of a node being the one marked. For example, T could be modeling a network with one defective unit. In order to discover which node of T is marked, we can perform *edge queries*: after querying the arc (i, j) of T (j being a child of i)[1], we receive an answer stating that either the marked node is a descendant of j (called a **yes** answer) or that the marked node is not a descendant of j (called a **no** answer).

A search strategy is a procedure that decides the next query to be posed based on the outcome of previous queries. As an example, consider the strategy for searching the tree T of Figure 6.1.a represented by the decision tree D of Figure 6.1.b. A decision tree can be interpreted as a strategy in the following way: at each step we query the arc indicated by the node of D that we are currently located. In case of a **yes** answer, we move to the right child of the current node and we move to its left child otherwise. We proceed with these operations until the marked node is found. Let us assume that 4 is the marked node in Figure 6.1.a. We start at the root of D and query the arc $(3, 4)$ of T, asking if the marked node is a descendant of node 4 in T. Since the answer is **yes**, we move to the right child of $(3, 4)$ in D and we query the arc $(4, 6)$ in T. In this case, the outcome of the query $(4, 6)$ is **no** and then we move to node $(4, 5)$ of D. By querying this node we conclude that the marked node of T is

[1]Henceforth, when we refer to the arc (i, j), j is a child of i.

in indeed 4.

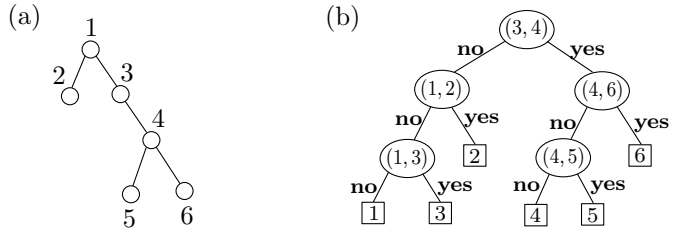

Figure 6.1: (a) Tree T. (b) Example of a decision tree for T; Internal nodes correspond to arcs of T and leaves to nodes of T

We define the average number of queries of a strategy $\mathcal{S}$ as $\sum_{v \in T} s_v w(v)$, where s_v is the number of queries needed to find the marked node when v is the marked node. Therefore, our optimization problem is to find the strategy with minimum expected number of queries. We make a more formal definition of a strategy by means of decision trees in the next section.

Besides generalizing a fundamental problem in theoretical computer science, searching in posets (and in particular in trees) also has practical applications such as in file system synchronization and software testing [MOW08]. We remark that although these applications were considered in the 'worst case' version of this problem, taking into account the likelihood that the elements are marked (for instance via code complexity measures in the latter example) may lead to improved searches.

Statement of the results. Our main result is a linear time algorithm that provides the first constant factor approximation for the problem of binary searching in trees. The algorithm is based on the decomposition of the input tree into special paths. A search strategy is computed for each of these paths and then combined to form a strategy for searching the original tree. This decomposition is motivated by the fact that the problem of binary searching in paths is easily reduced to the well-solved problem of searching in ordered lists with access probabilities.

As mentioned previously, the complexity of this 'average case' version of the problem of binary searching in trees remains open, which contrasts with its 'worst case' version that is polynomially solvable [BFN99b, OP06, MOW08].

Related work. Searching in totally ordered sets is a very well studied problem [Knuth98]. In addition, many variants have also been considered, such as when there is information about the likelihood of each element being the one marked [PS93], or where each query has a different fixed cost and the objective is to find a strategy with least total cost [Knight88, LMP99, LMP01]. As a

generalization of the latter, [NBB+00, SNB+03] considered the variant when the cost of each query depends on the queries posed previously.

The most general version of our problem when the input is a poset instead of a tree was first considered by Lipman and Abrahams [LA95]. Apart from introducing the problem, they present an optimized exponential time algorithm for solving it. In [KPB99], Kosaraju et. al. present a greedy $O(\log n)$-approximation algorithm. In fact, their algorithm handles more general searches, see [LN04, CPR+07] for other more general results. To the best of our knowledge, this $O(\log n)$-approximation algorithm is the only available result, with theoretical approximation guarantee, for the average case version of searching in trees. Therefore, our constant approximation represents a significant advance for this problem.

The variant of our problem where the goal is to minimize the number of queries in the worst case for searching in trees, instead of minimizing the average case, was first considered by Ben-Asher et. al. [BFN99b]. They have shown that it can be solved in $O(n^4 \log^3 n)$ via dynamic programming. Recent advances [OP06, MOW08] have reduced the time complexity to $O(n^3)$ and then $O(n)$. In contrast, the more general version where the input is a poset instead of a tree is NP-hard [CDK+04].