







A Nearly Optimal Randomized Algorithm for Explorable Heap Selection

Sander Borst¹ , Daniel Dadush¹ , Sophie Huiberts² ,
and Danish Kashaev¹ 

¹ Centrum Wiskunde & Informatica (CWI), Amsterdam, The Netherlands
{sander.borst,dadush,danish.kashaev}@cwi.nl

² Columbia University, New York, USA
sophie@huiberts.me

Abstract. Explorable heap selection is the problem of selecting the n^{th} smallest value in a binary heap. The key values can only be accessed by traversing through the underlying infinite binary tree, and the complexity of the algorithm is measured by the total distance traveled in the tree (each edge has unit cost). This problem was originally proposed as a model to study search strategies for the branch-and-bound algorithm with storage restrictions by Karp, Saks and Widgerson (FOCS '86), who gave deterministic and randomized $n \cdot \exp(O(\sqrt{\log n}))$ time algorithms using $O(\log(n)^{2.5})$ and $O(\sqrt{\log n})$ space respectively. We present a new randomized algorithm with running time $O(n \log(n)^3)$ against an oblivious adversary using $O(\log n)$ space, substantially improving the previous best randomized running time at the expense of slightly increased space usage. We also show an $\Omega(\log(n)n / \log(\log(n)))$ lower bound for any algorithm that solves the problem in the same amount of space, indicating that our algorithm is nearly optimal.

1 Introduction

Many important problems in theoretical computer science are fundamentally search problems. The objective of these problems is to find a certain solution from the search space. In this paper we analyze a search problem that we call *explorable heap selection*. The problem is related to the famous branch-and-bound algorithm and was originally proposed by Karp, Widgerson and Saks [13] to model node selection for branch-and-bound with low space-complexity. Furthermore, as we will explain later, the problem remains practically relevant to branch-and-bound even in the full space setting.

Due to space limitations, we have omitted several proofs. These can be found in [7]. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement QIP-805241).

The explorable heap selection problem¹ is an online graph exploration problem for an agent on a rooted (possibly infinite) binary tree. The nodes of the tree are labeled by distinct real numbers (the key values) that increase along every path starting from the root. The tree can thus be thought of as a min-heap. Starting at the root, the agent’s objective is to select the n^{th} smallest value in the tree while minimizing the distance traveled, where each edge of the tree has unit travel cost. The key value of a node is only revealed when the agent visits it, and the problem thus has an online nature. When the agent learns the key value of a node, it still does not know the rank of this value.

A simple selection strategy is to use the best-first rule, which repeatedly explores the unexplored node whose parent has the smallest key value. While this rule is optimal in terms of the number of nodes that it explores, namely $\Theta(n)$, the distance traveled by the agent can be far from optimal. In the worst-case, an agent using this rule will need to travel a distance of $\Theta(n^2)$ to find the n^{th} smallest value. A simple bad example for this rule is to consider a rooted tree consisting of two paths (which one can extend to a binary tree), where the two paths are consecutively labeled by all positive even and odd integers respectively.

Improving on the best-first strategy, Karp, Saks and Wigderson [13] gave a randomized algorithm with expected cost $n \cdot \exp(O(\sqrt{\log(n)}))$ using $O(\sqrt{\log(n)})$ working space. They also showed how to make the algorithm deterministic using $O(\log(n)^{2.5})$ space. In this work, our main contribution is an improved randomized algorithm with expected cost $O(n \log(n)^3)$ using $O(\log(n))$ space. Given the $\Omega(n)$ lower bound, our travel cost is optimal up to logarithmic factors. Furthermore we show that any algorithm for explorable heap selection that only uses s units of memory, must take at least $n \cdot \log_s(n)$ time in expectation. An interesting open problem is the question whether a superlinear lower bound also holds without any restriction on the memory usage.

To clarify the memory model, it is assumed that any key value and $O(\log n)$ bit integer can be stored using $O(1)$ space. We also assume that maintaining the current position in the tree does not take up memory. Furthermore, we assume that key value comparisons and moving across an edge of the tree require $O(1)$ time. Under these assumptions, the running times of the above algorithms are in fact proportional to their travel cost. Throughout the paper, we will thus use travel cost and running time interchangeably.

Motivation. The motivation to look at this problem comes from the branch-and-bound algorithm. This is a well-known algorithm that can be used for solving many types of problems. In particular, it is often used to solve integer linear programs (ILP), which are of the form $\arg \min \{c^\top x : x \in \mathbb{Z}^n, Ax \leq b\}$. In that setting, branch-and-bound works by first solving the linear programming (LP) relaxation, which does not have integrality constraints. The value of the solution to the relaxation forms a lower bound on the objective value of the original problem. Moreover, if this solution only has integral components, it is also optimal for the original problem. Otherwise, the algorithm chooses a component x_i for which the solution value \hat{x}_i is not integral. It then creates two new subproblems, by either adding the constraint $x_i \leq \lfloor \hat{x}_i \rfloor$ or $x_i \geq \lceil \hat{x}_i \rceil$. This

¹ [13] did not name the problem, so we have given a descriptive name here.

operation is called *branching*. The tree of subproblems, in which the children of a problem are created by the branching operation, is called the branch-and-bound tree. Because a subproblem contains more constraints than its parent, its objective value is greater or equal to the one of its parent.

At the core, the algorithm consists of two important components: the branching rule and the node selection rule. The branching rule determines how to split up a problem into subproblems, by choosing a variable to branch on. Substantial research has been done on branching rules, see, e.g., [2, 4, 14, 15].

The node selection rule decides which subproblem to solve next. Not much theoretical research has been done on the choice of the node selection rule. Traditionally, the best-first strategy is thought to be optimal from a theoretical perspective because this rule minimizes the number of nodes that need to be visited. However, to efficiently implement this rule the solver needs space proportional to the number of explored nodes, because all of them need to be kept in memory. In contrast to this, a simple strategy like depth-first search only needs to store the current solution. Unfortunately, performing a depth-first search can lead to an arbitrarily bad running time. This was the original motivation for introducing the explorable heap selection problem [13]. By guessing the number N of branch-and-bound nodes whose LP values are at most that of the optimal IP solution (which can be done via successive doubling), a search strategy for this problem can be directly interpreted as a node selection rule. The algorithm that they introduced can therefore be used to implement branch-and-bound efficiently in only $O\left(\sqrt{\log(N)}\right)$ space.

In practice, computers are usually able to store all explored nodes of the branch-and-bound tree in memory. However, many MIP-solvers still make use of a hybrid method that consists of both depth-first and best-first searches. This is not only done because depth-first search uses less memory, but also because it is often faster. Experimental studies have confirmed that the depth-first strategy is in many cases faster than best-first one [8]. This seems contradictory, because the running time of best-first search is often thought to be theoretically optimal.

In part, this contradiction can be explained by the fact that actual IP-solvers often employ complementary techniques and heuristics on top of branch-and-bound, which might benefit from depth-first searches. Additionally, a best-first search can hop between different parts of the tree, while a depth first search subsequently explores nodes that are very close to each other. In the latter case, the LP-solver can start from a very similar state, which is known as warm starting. This is faster for a variety of technical reasons [1]. For example, this can be the case when the LP-solver makes use of the LU-factorization of the optimal basis matrix [16]. Through the use of dynamic algorithms, computing this can be done faster if a factorization for a similar LP-basis is known [19]. Because of its large size, MIP-solvers will often not store the LU-factorization for all nodes in the tree. This makes it beneficial to move between similar nodes in the branch-and-bound tree. Furthermore, moving from one part of the tree to another means that the solver needs to undo and redo many bound changes, which also takes up time. Hence, the amount of distance traveled between nodes

in the tree is a metric that influences the running time. This can also be observed when running the academic MIP-solver SCIP [12].

The explorable heap selection problem captures these benefits of locality by measuring the running time in terms of the amount of travel through the tree. Therefore, we argue that this problem is still relevant for the choice of a node selection rule, even if all nodes can be stored in memory.

Related Work. The explorable heap selection problem was first introduced in [13]. Their result was later applied to prove an upper bound on the parallel running time of branch-and-bound [18].

When random access to the heap is provided at constant cost, selecting the n^{th} value in the heap can be done by a deterministic algorithm in $O(n)$ time by using an additional $O(n)$ memory for auxilliary data structures [11].

The explorable heap selection problem can be thought of as a *search game* [3] and bears some similarity to the *cow path problem*. In the cow path problem, an agent explores an unweighted unlabeled graph in search of a target node. The location of the target node is unknown, but when the agent visits a node they are told whether or not that node is the target. The performance of an algorithm is judged by the ratio of the number of visited nodes to the distance of the target from the agent’s starting point. In both the cow path problem and the explorable heap selection problem, the cost of backtracking and retracing paths is an important consideration. The cow path problem on infinite b -ary trees was studied in [9] under the assumption that when present at a node the agent can obtain an estimate on that node’s distance to the target.

Other explorable graph problems exist without a target, where typically the graph itself is unknown at the outset. There is an extensive literature on exploration both in graphs and in the plane. Models have been studied, in which one tried to minimize either the distance traveled or the amount of used memory. For more information we refer to [6, 20] and the references therein.

Outline. In Sect. 2 we formally introduce the explorable heap selection problem and any notation we will use. In Sect. 3 we introduce a new algorithm for solving this problem and provide a running time analysis. In Sect. 4 we give a lower bound on the complexity of solving explorable heap selection using a limited amount of memory.

2 The Explorable Heap Selection Problem

In this section we introduce the formal model for the explorable heap selection problem. The input to the algorithm is an infinite binary tree $T = (V, E)$, where each node $v \in V$ has an associated real value, denoted by $\text{val}(v) \in \mathbb{R}$. We assume that all the values are distinct and that for each node in the tree, the values of its children are larger than its own value. The binary tree T is thus a heap.

We want to find the n^{th} smallest value in this tree. This may be seen as an online graph exploration problem where an agent can move in the tree and learns the value of a node each time he explores it. At each time step, the agent resides

at a vertex $v \in V$ and may decide to move to either the left child, the right child or the parent of v (if it exists, i.e. if v is not the root of the tree). Each traversal of an edge costs one unit of time, and the complexity of an algorithm for this problem is thus measured by the total traveled distance in the binary tree. The algorithm is also allowed to store values in memory.

For a node $v \in V$, also per abuse of notation written $v \in T$, we denote by $T^{(v)}$ the subtree of T rooted at v . For a tree T and a value $\mathcal{L} \in \mathbb{R}$, we define the subtree $T_{\mathcal{L}} := \{v \in T \mid \text{val}(v) \leq \mathcal{L}\}$. We denote the n^{th} smallest value in T by $\text{SELECT}^T(n)$. This is the quantity that we are interested in finding algorithmically. We say that a value $\mathcal{V} \in \mathbb{R}$ is *good* for a tree T if $\mathcal{V} \leq \text{SELECT}^T(n)$ and *bad* otherwise. Similarly, we call a node $v \in T$ *good* if $\text{val}(v) \leq \text{SELECT}^T(n)$ and *bad* otherwise. We use $[k]$ to refer to the set $\{1, \dots, k\}$. When we write $\log(n)$, we assume the base of the logarithm to be 2.

We will often instruct the agent to move to an already discovered good vertex $v \in V$. The way this is done algorithmically is by saving $\text{val}(v)$ in memory and starting a depth first search at the root, turning back every time a value strictly bigger than $\text{val}(v)$ is encountered until finally finding $\text{val}(v)$. This takes at most $O(n)$ time, since we assume v to be a good node. If we instruct the agent to go back to the root from a certain vertex $v \in V$, this is simply done by traveling back in the tree, choosing to go to the parent of the current node at each step.

In later sections, we will often say that a subroutine takes a subtree $T^{(v)}$ as input. This implicitly means that we in fact pass it $\text{val}(v)$ as input, make the agent travel to $v \in T$ using the previously described procedure, call the subroutine from that position in the tree, and travel back to the original position at the end of the execution. Because the subroutine knows the value $\text{val}(v)$ of the root of $T^{(v)}$, it can ensure it never leaves the subtree $T^{(v)}$, thus making it possible to recurse on a subtree as if it were a rooted tree by itself.

We will sometimes want to pick a value uniformly at random from a set of values $\{\mathcal{V}_1, \dots, \mathcal{V}_k\}$ of unknown size that arrives in a streaming fashion, for instance when we traverse a part of the tree T by doing a depth first search. That is, we see the value \mathcal{V}_i at the i^{th} time step, but do not longer have access to it in memory once we move on to \mathcal{V}_{i+1} . This can be done by generating random values $\{X_1, \dots, X_k\}$ where, at the i^{th} time step, $X_i = \mathcal{V}_i$ with probability $1/i$, and $X_i = X_{i-1}$ otherwise. It is easy to check that X_k is a uniformly distributed sample from $\{\mathcal{V}_1, \dots, \mathcal{V}_k\}$.

3 A New Algorithm

The authors of [13] presented a deterministic algorithm that solves the explorable heap selection problem in $n \cdot \exp(O(\sqrt{\log(n)}))$ time and $O(n\sqrt{\log(n)})$ space. By replacing the binary search that is used in the algorithm by a randomized variant, they can decrease the space requirements. This way, they get a randomized algorithm with expected running time $n \cdot \exp(O(\sqrt{\log(n)}))$ and space complexity $O(\sqrt{\log(n)})$. Alternatively, the binary search can be implemented in a deterministic way by [17] to get the same running time with $O(\log(n)^{2.5})$ space.

We present a randomized algorithm with a running time $O(n \log(n)^3)$ and space complexity $O(\log(n))$. Unlike the algorithms mentioned before, our algorithm fundamentally relies on randomness to bound its running time. This bound only holds when the algorithm is run on a tree with labels that are fixed before the execution of the algorithm. That is, the tree must be generated by an adversary that is oblivious to the choices made by the algorithm. This is a stronger assumption than is needed for the algorithm that is given in [13], which also works against adaptive adversaries. An adaptive adversary is able to defer the decision of the node label to the time that the node is explored. Note that this distinction does not really matter for the application of the algorithm as a node selection rule in branch-and-bound, since there the node labels are fixed because they are derived from the integer program.

Theorem 1. *There exists a randomized algorithm that solves the explorable heap selection problem, with expected running time $O(n \log(n)^3)$ and $O(\log(n))$ space.*

The explorable heap selection problem can be seen as the problem of finding all n good nodes. Both our method and that of [13] function by first identifying a subtree consisting of only good nodes. The children of the leaves of this subtree are called “roots” and the subtree is extended by finding a number of new good nodes under these roots in multiple rounds.

In [13] this is done by running $O(c\sqrt{2\log(n)})$ different rounds, for some constant $c > 1$. In each round, the algorithm finds $n/c\sqrt{2\log(n)}$ new good nodes. These nodes are found by recursively exploring each active root and using binary search on the observed values to discover which of these values are good. Which active roots are recursively explored further depends on which values are good. The recursion in the algorithm is at most $O(\sqrt{\log(n)})$ levels deep, which is where the space complexity bound comes from.

In our algorithm, we take a different approach. We will call our algorithm consecutively with $n = 1, 2, 4, 8, \dots$. Hence, for a call to the algorithm, we can assume that we have already found at least $n/2$ good nodes. These nodes form a subtree of the original tree T . In each round, our algorithm chooses a random root under this subtree and finds every good node under it. It does so by doing recursive subcalls to the main algorithm on this root with values $n = 1, 2, 4, 8, \dots$. As soon as the recursively obtained node is a bad node, the algorithm stops searching the subtree of this root, since it is guaranteed that all the good nodes there have been found. The largest good value that is found can then be used to find additional good nodes under the other roots without recursive calls, through a simple depth-first search. Assuming that the node values were fixed in advance, we expect this largest good value to be greater than half of the other roots’ largest good values. Similarly, we expect its smallest bad value to be smaller than half of the other roots’ smallest bad values. By this principle, a sizeable fraction of the roots can, in expectation, be ruled out from getting a recursive call. Each round a new random root is selected until all good nodes have been found. This algorithm allows us to effectively perform binary search

on the list of roots, ordered by the largest good value contained in each of their subtrees in $O(\log n)$ rounds, and the same list ordered by the smallest bad values (Lemma 2). Bounding the expected number of good nodes found using recursive subcalls requires a subtle induction on two parameters (Lemma 1): both n and the number of good nodes that have been identified so far.

3.1 The Algorithm

The EXTEND procedure is the core of our algorithm. It finds the n^{th} smallest value in the tree, under the condition that the k^{th} smallest value \mathcal{L}_0 is provided to the algorithm for some $k \geq n/2$. Using this procedure, SELECT(n) can be solved by consecutively calling EXTEND($T, n_i, k_i, \mathcal{L}_i$) with parameters $(n_i, k_i) = (2^i, 2^{i-1})$ for $i \in \{1, \dots, \lceil \log(n) \rceil\}$.

Algorithm 1. The EXTEND procedure

```

1: Input:  $T$ : tree which is to be explored.
2:    $n \in \mathbb{N}$ : total number of good values to be found in the tree  $T$ , satisfying  $n \geq 2$ .
3:    $k \in \mathbb{N}$ : number of good values already found in the tree  $T$ , satisfying  $k \geq n/2$ .
4:    $\mathcal{L}_0 \in \mathbb{R}$ : value satisfying  $\text{DFS}(T, \mathcal{L}_0, n) = k$ .
5: Output: the  $n^{\text{th}}$  smallest value in  $T$ .
6: procedure EXTEND( $T, n, k, \mathcal{L}_0$ )
7:    $\mathcal{L} \leftarrow \mathcal{L}_0, \mathcal{U} \leftarrow \infty$ 
8:   while  $k < n$  do
9:      $r \leftarrow$  random element from  $\text{ROOTS}(T, \mathcal{L}_0, \mathcal{L}, \mathcal{U})$ 
10:     $\mathcal{L}' \leftarrow \max(\mathcal{L}, \text{val}(r))$ 
11:     $k' \leftarrow \text{DFS}(T, \mathcal{L}', n)$  // count the number of values  $\leq \mathcal{L}'$  in  $T$ 
12:     $c \leftarrow \text{DFS}(T^{(r)}, \mathcal{L}', n)$  // count the number of values  $\leq \mathcal{L}'$  in  $T^{(r)}$ 
13:     $c' \leftarrow \min(n - k' + c, 2c)$  // increase the number of values to be found in  $T^{(r)}$ 
14:    while  $k' < n$  do // loop until it is certified that  $\text{SELECT}^T(n) \leq \mathcal{L}'$ 
15:       $\tilde{\mathcal{L}}' \leftarrow \text{EXTEND}(T^{(r)}, c', c, \mathcal{L}')$ 
16:       $k' \leftarrow \text{DFS}(T, \tilde{\mathcal{L}}', n)$ 
17:       $c \leftarrow c'$ 
18:       $c' \leftarrow \min(n - k' + c, 2c)$ 
19:    end while
20:     $\tilde{\mathcal{L}}, \tilde{\mathcal{U}} \leftarrow \text{GOODVALUES}(T, T^{(r)}, \mathcal{L}', n)$  // find the good values in  $T^{(r)}$ 
21:     $\mathcal{L} \leftarrow \max(\mathcal{L}, \tilde{\mathcal{L}}), \mathcal{U} \leftarrow \min(\mathcal{U}, \tilde{\mathcal{U}})$ 
22:     $k \leftarrow \text{DFS}(T, \mathcal{L}, n)$  // compute the number of good values found in  $T$ 
23:  end while
24:  return  $\mathcal{L}$ 
25: end procedure

```

Let us describe a few invariants from the EXTEND procedure.

- \mathcal{L} and \mathcal{U} are respectively lower and upper bounds on $\text{SELECT}^T(n)$ during the whole execution of the procedure.
- The integer k counts the number of values $\leq \mathcal{L}$ in the full tree T .
- After an iteration of the inner while loop, \mathcal{L}' is set to the c^{th} smallest value in $T^{(r)}$. The variable c' then corresponds to the next value we would like to find

- in $T^{(r)}$ if we were to continue the search. Note that $c' \leq 2c$, enforcing that the recursive call to **EXTEND** satisfies its precondition, and that $c' \leq n - (k' - c)$ implies that $(k' - c) + c' \leq n$, which implies that the recursive subcall will not spend time searching for a value that is known in advance to be bad.
- k' always counts the number of values $\leq \mathcal{L}'$ in the full tree T . It is important to observe that this is a global parameter, and does not only count values below the current root. Moreover, $k' \geq n$ implies that we can stop searching below the current root, since it is guaranteed that all good values in $T^{(r)}$ have been found, i.e., \mathcal{L}' is larger than all the good values in $T^{(r)}$.

We now describe the subroutines used in the **EXTEND** procedure.

The Procedure DFS. The procedure **DFS** is a variant of depth first search. The input to the procedure is T , a cutoff value $\mathcal{L} \in \mathbb{R}$ and an integer $n \in \mathbb{N}$. The procedure returns the number of vertices in T whose value is at most \mathcal{L} .

It achieves that by exploring the tree T in a depth first search manner, starting at the root and turning back as soon as a node $w \in T$ such that $\text{val}(w) > \mathcal{L}$ is encountered. Moreover, if the number of nodes whose value is at most \mathcal{L} exceeds n during the search, the algorithm stops and returns $n + 1$.

The algorithm output is the following integer, whose value is at most \mathcal{L} :

$$\text{DFS}(T, \mathcal{L}, n) := \min \{|T_{\mathcal{L}}|, n + 1\}.$$

Observe that the **DFS** procedure allows us to check whether a node $w \in T$ is a good node, i.e. whether $\text{val}(w) \leq \text{SELECT}^T(n)$. Indeed, w is good if and only if $\text{DFS}(T, \text{val}(w), n) \leq n$.

This procedure visits only nodes in $T_{\mathcal{L}}$ or its direct descendants and its running time is thus $O(n)$. The space complexity is $O(1)$, since the only values needed to be stored in memory are \mathcal{L} , $\text{val}(v)$, where v is the root of the tree T , and a counter for the number of good values found so far.

The Procedure Roots. The procedure **ROOTS** takes as input a tree T as well as a lower bound $\mathcal{L}_0 \in \mathbb{R}$ on the value of $\text{SELECT}^T(n)$. We assume that the main algorithm has already found all the nodes $w \in T$ satisfying $\text{val}(w) \leq \mathcal{L}_0$. This means that the remaining values the main algorithm needs to find in T are all lying in the subtrees of the following nodes, that we call the \mathcal{L}_0 -roots of T (Fig. 1):

$$R(T, \mathcal{L}_0) := \{r \in T \setminus T_{\mathcal{L}_0} \mid r \text{ is a child of a node in } T_{\mathcal{L}_0}\}$$

In words, these are all the vertices in T one level deeper in the tree than $T_{\mathcal{L}_0}$. In addition to that, the procedure takes two other parameters $\mathcal{L}, \mathcal{U} \in \mathbb{R}$ as input, which correspond to (another) lower and upper bound on the value of $\text{SELECT}^T(n)$. These lower and upper bounds will be updated during the execution of the main algorithm. A key observation is that these bounds can allow us to remove certain roots in $R(T, \mathcal{L}_0)$ from consideration, in the sense that all the good values in that root's subtree will be certified to have already been found (Fig. 2):

$$\text{ROOTS}(T, \mathcal{L}_0, \mathcal{L}, \mathcal{U}) := \left\{ r \in R(T, \mathcal{L}_0) \mid \exists w \in T^{(r)} \text{ with } \text{val}(w) \in (\mathcal{L}, \mathcal{U}) \right\}$$

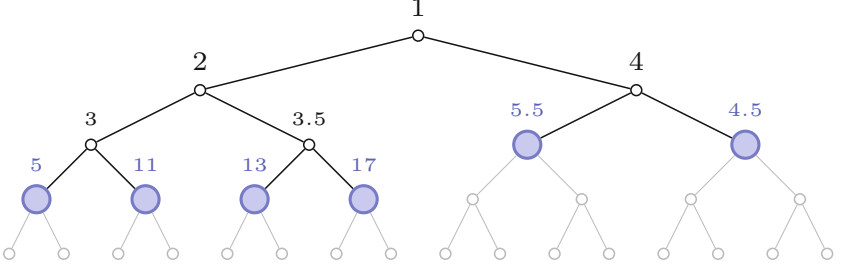


Fig. 1. An illustration of $R(T, \mathcal{L}_0)$ with $\mathcal{L}_0 = 4$. The number above each vertex is its value, the blue nodes are $R(T, \mathcal{L}_0)$, whereas the subtree above is $T_{\mathcal{L}_0}$. (Color figure online)

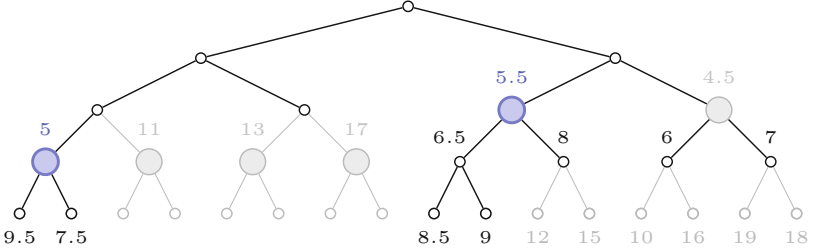


Fig. 2. An illustration of the ROOTS procedure with $\mathcal{L}_0 = 4$, $\mathcal{L} = 7$ and $\mathcal{U} = 10$. Only two active roots remain, and are both colored in blue. The other roots are considered killed since all the good values have been found in their subtrees. (Color figure online)

This subroutine can be implemented by running a depth first search starting at the root of T and exploring $T_{\mathcal{L}}$ with its direct descendants. Since \mathcal{L} is known to be good, the running time is bounded by $O(|T_{\mathcal{L}}|) = O(n)$. In the main algorithm, we will only need this procedure in order to select a root from $\text{ROOTS}(T, \mathcal{L}_0, \mathcal{L}, \mathcal{U})$ uniformly at random, without having to store the whole list in memory. This can then be achieved in $O(1)$ space, since one then only needs to store $\text{val}(v)$, \mathcal{L}_0 , \mathcal{L} and \mathcal{U} in memory, where v is the root of the tree T .

The Procedure GoodValues. The procedure `GOODVALUES` takes as input a tree T , a subtree $T^{(r)}$, a value $\mathcal{L}' \in \mathbb{R}_{\geq 0}$ and an integer $n \in \mathbb{N}$. The procedure then analyzes the set $S := \{\text{val}(w) \mid w \in T^{(r)}, \text{val}(w) \leq \mathcal{L}'\}$ and outputs both the largest good value and the smallest bad value in that set, that we respectively call \mathcal{L} and \mathcal{U} . If no bad values exist in S , the algorithm sets $\mathcal{U} = \infty$.

The procedure can be implemented using a randomized binary search on the values in S , where the procedure `DFS` is used to check whether a value is good. This makes the procedure have a running time of $O(n \log n)$.

The procedure only needs $O(1)$ space, since the only values necessary to be kept in memory are $\text{val}(v)$ (where v is the root of the tree T), $\text{val}(r)$, \mathcal{L} , \mathcal{U} and \mathcal{L}' , as well as the fact that every call to `DFS` also requires $O(1)$ space.

3.2 Proof of Correctness

Theorem 2. *At the end of the execution of Algorithm 1, \mathcal{L} is set to the n^{th} smallest value in T . Moreover, the algorithm is guaranteed to terminate.*

Proof sketch. The variable \mathcal{L} is always set to the first output of the procedure GOODVALUES, which is always the value of a good node, implying $\mathcal{L} \leq \text{SELECT}^T(n)$. The other inequality follows since the outer while loop ends when at least n good nodes have been found in T .

3.3 Running Time Analysis

The main challenge in analyzing the running time of the algorithm is dealing with the cost of the recursive subcalls in the EXTEND procedure. For this we rely on two important ideas.

Firstly, note that n is the index of the node value that we want to find, while k is the index of the node value that is passed to the procedure. So, the procedure needs to only find $n - k$ new good nodes. Our runtime bound for the recursive subcalls that are performed does not just depend on n , but also on $n - k$.

We will show that the amount of travel done in the non-recursive part of a call of EXTEND with parameters n and k is bounded by $O(n \log(n)^2)$. We will charge this travel to the parent call that makes these recursive calls. Hence, a parent call that does z recursive calls with parameters $(n_1, k_1), \dots, (n_z, k_z)$ will be charged a cost of $\sum_{i=1}^z n_i \log(n_i)^2$. In our analysis, we will show that this sum can be upper bounded by $(n - k) \log(n)^2$. So, for every recursive call with parameters n and k , a cost of at most $(n - k) \log(n)^2$ is incurred by the caller.

Now we just need to bound the sum over $(n - k) \log(n)^2$ for all calls with parameters n and k that are done. We do this by first considering a single algorithm call with parameters n and k that makes z recursive subcalls with parameters $(n_1, k_1), \dots, (n_z, k_z)$. For such a subcall, we would like to bound the sum $\sum_{i=1}^z (n_i - k_i) \log(n_i)^2$ by $(n - k) \log(n)^2$. However, this bound does not hold deterministically. Instead, we show that this bound does hold in expectation.

Now we know that every layer of recursion incurs an expected cost of at most $(n - k) \log(n)^2$. Because the parameter n will decrease by at least a constant factor in each layer of recursion, there can be at most $O(\log(n))$ layers. An upper bound of $O((n - k) \log(n)^3)$ on the expected running time of the EXTEND then follows for the recursive part.

Combining this with the upper bound of $O(n \log(n)^2)$ on the non-recursive part, we get a total running time of $O(n \log(n)^2) + O((n - k) \log(n)^3)$ for the EXTEND procedure, which then implies a running time of $O(n \log(n)^3)$ for the SELECT procedure.

Let us now prove these claims. We first show that the expectation of $\sum_{i=1}^z (n_i - k_i)$ is bounded.

Lemma 1. *Let z be the number of recursive calls that are done in the main loop of $\text{EXTEND}(T, n^*, k^*, \mathcal{L})$ with parameter $k \geq 1$. For $i \in [z]$, let n_i and k_i be the*

values of n and k that are given as parameters to the i th such subcall. Then:

$$\mathbb{E} \left[\sum_{i=1}^z n_i - k_i \right] \leq n^* - k^*.$$

Proof. Assume we have m roots, whose order is fixed. For $i \in [z]$, let $r_i \in [m]$ be such that the i th recursive subcall is done on the root with index r_i . For $t \in [m]$, let $s_t = \sum_{i=1}^z \mathbf{1}_{r_i=t}(n_i - k_i)$. From the algorithm we see that when $r_i = t$, all successive recursive calls will also be on root t , until all good nodes under this root have been found. The updated values of \mathcal{L} and \mathcal{U} ensure this root is never selected again after this, hence all iterations i with $r_i = t$ are consecutive. Now let a_t, b_t be variables that respectively denote the first and last indices i with $r_i = t$. When there is no iteration i with $r_i = t$, then $a_t = b_t = \infty$.

For two calls i and $i+1$ with $r_i = t = r_{i+1}$, observe that after call i already n_i good nodes under root t have been found. On line 15, c' corresponds to n_i and c corresponds to k_i , hence $k_{i+1} = n_i$. Therefore, the definition of s_t is a telescoping series and can be rewritten as $s_t = n_{b_t} - k_{a_t}$, when we define $k_\infty = n_\infty = 0$.

Let $p = n^* - k^*$ and let $W = \{w_1, \dots, w_p\}$ denote the p smallest values under T that are larger than \mathcal{L}_0 , in increasing order. Now each of these values in W will be part of a subtree generated by one of the roots. For the $j \in [p]$, let $d_j \in [m]$ be such that value w_j is part of the subtree of root d_j . Let $S_t = \{j \in [p] : d_j = t\}$. We will now show that for each root r_t , we have $\mathbb{E}[s_t] \leq |S_t|$. This will imply that $\mathbb{E}[\sum_{i=1}^z n_i - k_i] = \sum_{t=1}^m \mathbb{E}[s_t] \leq \sum_{t=1}^m |S_t| = n^* - k^*$.

First, consider a root t with $t \neq d_p$. On line 9, each iteration a random root is chosen. In every iteration root d_p will be among the active roots. So the probability that this root is chosen before root t is at least a half. In that case, after the iteration of root d_p , \mathcal{L} will be set to w_p . Then $\text{DFS}(T, \mathcal{L}, n)$ returns n , and the algorithm terminates. Since no subcalls are done on root t , $s_i = 0$.

If the algorithm does do subcalls i with $r_i = t$, then consider iteration b_t , the last iteration i that has $r_i = t$. Before this iteration, already k_{b_t} good nodes under the root have been found by the algorithm. It can be seen in the algorithm on lines 13 and 18 that $n_{b_t} \leq 2k_{b_t}$. Hence $s_t = n_{b_t} - k_{a_t} \leq n_{b_t} \leq 2k_{b_t} \leq 2|S_t|$. We therefore have $\mathbb{E}[s_t] \leq \frac{1}{2} \cdot 0 + \frac{1}{2} \cdot 2|S_t| = |S_t|$.

Now consider the root d_p . If $S_{d_p} = [p]$, then $s_p = n_{b_{d_p}} - k_{a_{d_p}} \leq n^* - k^* = |S_{d_p}|$, because $n_{b_{d_p}} \leq n^*$ and $k_{a_{d_p}} \geq k^*$. If $S_{d_p} \subsetneq [p]$, then there exists a j with $d_j \neq d_p$. Thus, we can define $j^* = \max\{j \in [p] : d_j \neq d_p\}$. With probability a half, root d_{j^*} is considered before root d_p . If this happens, \mathcal{L} will be equal to w_{j^*} when root d_p is selected by the algorithm. In particular, this means that $k_{a_{d_p}}$ will be equal to j^* . Recall the stated invariant that $c' \leq n^* - k^* = p$, and hence $n_{b_{d_p}} = c' \leq p$. Now we can see that $s_{d_p} = n_{b_{d_p}} - k_{a_{d_p}} \leq p - j^*$.

If root d_p is chosen before root d_{j^*} , then consider the last recursive call b_{d_p} to EXTEND that we do on root d_p . Define $A = [k' - k^*] \cap S_{d_p}$, i.e. the set of all good values under root d_p that have been found so far. We distinguish two cases.

If $k' - k^* \geq j^*$, i.e., when all good values under d_{j^*} have been found, then by definition of j^* , $[p] \setminus [k' - k^*] \subseteq [p] \setminus [j^*] \subseteq S_{d_p}$. Because A and $[p] \setminus [k' - k^*]$

are disjoint, we have $|A| + (n^* - k') = |A| + |[p] \setminus [k' - k^*]| \leq |S_{d_p}|$. Hence, we have $c' \leq n^* - k' + c = n^* - k' + |A| \leq |S_{d_p}|$. Therefore, $s_{d_p} \leq n_{b_{d_p}} = c' \leq |S_{d_p}|$.

If $k' - k^* < j^*$ at the time of subcall b_{d_p} , then the last good value under d_{j^*} has yet to be found, implying that $A \subseteq [j^*]$. From the definition of j^* we get $[p] \setminus [j^*] \subseteq S_{d_p}$. Hence, $|A| \leq |S_{d_p}| - |[p] \setminus [j^*]| = |S_{d_p}| - (p - j^*)$. Thus $c' \leq 2c = 2|A| \leq 2(|S_{d_p}| - (p - j^*))$. So, in this case we have $s_{d_p} \leq n_{b_{d_p}} = c' \leq 2(|S_{d_p}| - (p - j^*))$.

Collecting the three cases above, we find that

$$\begin{aligned} \mathbb{E}[s_{d_p}] &\leq \frac{1}{2} \cdot (p - j^*) + \frac{1}{2} \cdot \max(|S_{d_p}|, 2(|S_{d_p}| - (p - j^*))) \\ &\leq \max\left(\frac{1}{2}|S_{d_p}| + \frac{1}{2}(p - j^*), |S_{d_p}| - \frac{1}{2}(p - j^*)\right). \end{aligned}$$

Lastly, by definition of j^* we have $[p] \setminus [j^*] \subseteq S_{d_p}$, from which it follows that $p - j^* \leq |S_{d_p}|$. We finish the proof by observing that this implies

$$\max\left(\frac{1}{2}|S_{d_p}| + \frac{1}{2}(p - j^*), |S_{d_p}| - \frac{1}{2}(p - j^*)\right) \leq |S_{d_p}|,$$

which finishes the proof.

Lemma 2. *The expected number of times that the outermost while-loop (at line 8) is executed by the procedure EXTEND is at most $O(\log(n))$.*

Proof sketch. Let $A_\ell(\mathcal{L}) := \{r_j : \ell_j > \mathcal{L}\}$ and $A_u(\mathcal{U}) := \{r_j : u_j < \mathcal{U}\}$. Observe that $\text{ROOTS}(T, \mathcal{L}_0, \mathcal{L}, \mathcal{U}) = A_\ell(\mathcal{L}) \cup A_u(\mathcal{U})$ for any $\mathcal{L} \leq \mathcal{U}$. One can show that in each iteration the size of either $A_\ell(\mathcal{L})$ or $A_u(\mathcal{U})$ halves in expectation. Hence, in expectation at most $O(\log R)$ iterations are needed, where $\log R$ is the initial number of roots. Since $R \leq n$, the lemma follows.

By an elementary analysis of the algorithm and applying Lemma 2 we can prove the following lemma.

Lemma 3. *The expected running time of the non-recursive part of every call to EXTEND is $O(n \log(n)^2)$.*

Finally we are able to prove the running time bound.

Lemma 4. *Let $R(T, n, k)$ denote the running time of a call to EXTEND(T, n, k, \mathcal{L}_0). Then there exists $C > 0$ such that*

$$\mathbb{E}[R(T, n, k)] \leq 5C(n - k) \log(n)^3 + Cn \log(n)^2.$$

Proof. We will prove this with induction on $r := \lceil \log(n) \rceil$. For $r = 1$, we have $n \leq 2$. In this case R is constant, proving our induction base.

Now consider a call EXTEND(T, n, k, \mathcal{L}_0) and assume the induction claim is true when $\lceil \log(n) \rceil \leq r - 1$. By Lemma 3, we can choose C such that this running time is bounded by $C \cdot n \log(n)$.

Now we move on to the recursive part of the algorithm. All calls to $\text{EXTEND}(T, n, k, \mathcal{L}_0)$ with $k = 0$ will have $n = 1$, so each of these calls takes only $O(1)$ time. Hence we can safely ignore these calls.

Let z be the number of recursive calls to $\text{EXTEND}(T, n, k, \mathcal{L}_0)$ that are done from the base call with $k \geq 1$. Let T_i, k_i, n_i for $i \in [z]$ be the arguments of these function calls. Note that $n/2 \geq n_i \geq 2$ for all i . By the induction hypothesis the expectation of the recursive part of the running time is:

$$\begin{aligned} \mathbb{E} \left[\sum_{i=1}^z R(T_i, n_i, k_i) \right] &\leq \mathbb{E} \left[\sum_{i=1}^z 5C \log(n_i) (n_i - k_i) \log(n_i)^2 + C n_i \log(n_i)^2 \right] \\ &\leq 5C \log(n/2) \mathbb{E} \left[\sum_{i=1}^r n_i - k_i \right] \log(n)^2 + C \log(n)^2 \sum_{i=1}^r n_i \\ &\leq 5C (\log(n) - 1) (n - k) \log(n)^2 + 5C \log(n)^2 (n - k) \\ &\leq 5C (n - k) \log(n)^3. \end{aligned}$$

Here we used Lemma 1 as well as the fact that $\sum_{i=1}^r n_i \leq 4(n - k)$. To see this, consider an arbitrary root q with s good values under it. Now $\sum_{i=1}^z \mathbf{1}_{T_i=T(q)} n_i \leq \sum_{i=2}^{\lceil \log(s+1) \rceil} 2^i \leq 2^{\lceil \log(s+1) \rceil + 1} \leq 4s$. In total there are $n - k$ good values under the roots, and hence $\sum_{i=1}^z n_i \leq 4(n - k)$.

Adding the expected running time of the recursive and the non-recursive part, we see that

$$\mathbb{E}[R(T, n, k)] \leq 5C(n - k) \log(n)^3 + Cn \log(n)^2.$$

3.4 Space Complexity Analysis

We prove in this section the space complexity of our algorithm.

Theorem 3. *The procedure EXTEND runs in $O(\log(n))$ space.*

Proof. Observe that the subroutines DFS, ROOTS and GOODVALUES all require $O(1)$ memory, as argued in their respective analyses. Hence the space complexity of the non-recursive part of the EXTEND is $O(1)$. Any recursive subcall $\text{EXTEND}(T_i, n_i, k_i, \mathcal{L}_i)$ resulting from a call to $\text{EXTEND}(T, n, k, \mathcal{L})$, will have $n_i \leq n/2$. Hence, the depth of recursion is at most $O(\log(n))$, which implies that the same is true for the space complexity.

4 Lower Bound

In general, no lower bound is known for the running time of the selection problem. However, we will show that any algorithm with space complexity at most s , has a running time of at least $\Omega(n \log_s(n))$. The tree that is used for the lower bound construction is very simple: a root with two trails of length $O(n)$ attached to it.

We will make use of a variant of the communication complexity model. In this model there are two agents A and B , that both have access to their own

sets of values in S_A and S_B respectively. These sets are the input. We have $|S_A| = n + 1$ and $|S_B| = n$. Assume that all values S_A and S_B are different. Now consider the problem where player A wants to compute the median of $S_A \cup S_B$.

Because the players only have access to their own values, they need to communicate. They use a protocol, that can consist of multiple rounds. In every odd round, player A can do computations and send units of information to player B . In every even round, player B does computations and sends information to player A . We assume that sending one value from S_A or S_B takes up one *unit of information*. Furthermore, we assume that, except for comparisons, no operations can be performed on the values.

We can reduce median computation to the explorable heap selection problem.

Lemma 5. *If there is a algorithm that solves SELECT($3n$) in $f(n)n$ time and g space, then there is a protocol for median computation that uses $f(n)/2$ rounds in each of which at most g units of information are sent.*

By showing a lower bound on the number of necessary rounds for median computation we can now prove the lower bound.

Theorem 4. *The time complexity of any randomized algorithm for SELECT(n) with at most g units of storage is $\Omega(n \log_{g+1}(n))$.*

References

1. Achterberg, T.: Constraint Integer Programming. Ph.D. thesis, TU Berlin (2009)
2. Achterberg, T., Koch, T., Martin, A.: Branching rules revisited. *Oper. Res. Lett.* **33**(1), 42–54 (2005). <https://doi.org/10.1016/j.orl.2004.04.002>
3. Alpern, S., Gal, S.: The Theory of Search Games and Rendezvous, vol. 55. Springer, New York (2006). <https://doi.org/10.1007/b100809>
4. Balcan, M.F., Dick, T., Sandholm, T., Vitercik, E.: Learning to branch. In: ICML (2018)
5. Banerjee, S., Cohen-Addad, V., Gupta, A., Li, Z.: Graph searching with predictions, December 2022
6. Berman, P.: On-line searching and navigation. In: Fiat, A., Woeginger, G.J. (eds.) *Online Algorithms*. LNCS, vol. 1442, pp. 232–241. Springer, Heidelberg (1998). <https://doi.org/10.1007/BFb0029571>
7. Borst, S., Dadush, D., Huiberts, S., Kashaev, D.: A nearly optimal randomized algorithm for explorable heap selection, October 2022. <https://doi.org/10.48550/arXiv.2210.05982>
8. Clausen, J., Perregaard, M.: On the best search strategy in parallel branch-and-bound: best-first search versus lazy depth-first search. *Ann. Oper. Res.* **90**, 1–17 (1999)
9. Dasgupta, P., Chakrabarti, P.P., DeSarkar, S.C.: A near optimal algorithm for the extended cow-path problem in the presence of relative errors. In: Thiagarajan, P.S. (ed.) *FSTTCS 1995*. LNCS, vol. 1026, pp. 22–36. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-60692-0_38
10. Diks, K., Fraigniaud, P., Kranakis, E., Pelc, A.: Tree exploration with little memory. *J. Algorithms* **51**(1), 38–63 (2004). <https://doi.org/10.1016/j.jalgor.2003.10.002>

11. Frederickson, G.: An optimal algorithm for selection in a min-heap. *Inf. Comput.* **104**(2), 197–214 (1993). <https://doi.org/10.1006/inco.1993.1030>
12. Gleixner, A.M.: Personal communication, November 2022
13. Karp, R.M., Saks, M.E., Wigderson, A.: On a search problem related to branch-and-bound procedures. In: *FOCS*, pp. 19–28 (1986)
14. Linderoth, J.T., Savelsbergh, M.W.P.: A computational study of search strategies for mixed integer programming. *INFORMS J. Comput.* **11**(2), 173–187 (1999). <https://doi.org/10.1287/ijoc.11.2.173>
15. Lodi, A., Zarpellon, G.: On learning and branching: a survey. *TOP* **25**(2), 207–236 (2017). <https://doi.org/10.1007/s11750-017-0451-6>
16. Morrison, D.R., Jacobson, S.H., Sauppe, J.J., Sewell, E.C.: Branch-and-bound algorithms: a survey of recent advances in searching, branching, and pruning. *Discret. Optim.* **19**, 79–102 (2016). <https://doi.org/10.1016/j.disopt.2016.01.005>
17. Munro, J., Paterson, M.: Selection and sorting with limited storage. *Theoret. Comput. Sci.* **12**(3), 315–323 (1980). [https://doi.org/10.1016/0304-3975\(80\)90061-4](https://doi.org/10.1016/0304-3975(80)90061-4)
18. Pietracaprina, A., Pucci, G., Silvestri, F., Vandin, F.: Space-efficient parallel algorithms for combinatorial search problems. *J. Parallel Distrib. Comput.* **76**, 58–65 (2015)
19. Suhl, L.M., Suhl, U.H.: A fast LU update for linear programming. *Ann. Oper. Res.* **43**(1), 33–47 (1993). <https://doi.org/10.1007/BF02025534>
20. Kamphans, T.: Models and algorithms for online exploration and search. Ph.D. thesis, Rheinische Friedrich-Wilhelms-Universität Bonn (2006). <https://hdl.handle.net/20.500.11811/2622>