Mathematical Programming
https://doi.org/10.1007/s10107-024-02145-5

FULL LENGTH PAPER

Series B

A nearly optimal randomized algorithm for explorable heap selection

Sander Borst[1] · **Daniel Dadush**[1,2] · **Sophie Huiberts**[3] · **Danish Kashaev**[1]

Received: 29 July 2023 / Accepted: 5 September 2024

Abstract

Explorable heap selection is the problem of selecting the nth 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.

Keywords Graph exploration · Branch and bound · Node selection · Online algorithm

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)
An early version of work was published in 'Integer Programming and Combinatorial Optimization' [1].

✉ Daniel Dadush
d.n.dadush@uu.nl; dadush@cwi.nl

Sander Borst
sander.borst@cwi.nl

Sophie Huiberts
sophie@huiberts.me

Danish Kashaev
danish.kashaev@cwi.nl

1 Centrum Wiskunde & Informatica (CWI), Amsterdam, The Netherlands

2 Utrecht University, Utrecht, Netherlands

3 Columbia University, New York, USA

Published online: 05 November 2024

Mathematics Subject Classification 90C57 · 90C57

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, Saks and Widgerson [2] 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.

The explorable heap selection problem[1] 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 thus the problem has an online nature. When the agent learns the key value of a node, it still does not know the rank of this value.

The selection problem for ordinary heaps, which allow for random access (i.e., jumping to arbitrary nodes in the tree for "free"), has also been studied. In this model, it was shown by [3] that selecting the n^{th} minimum can be achieved deterministically in $O(n)$ time using $O(n)$ workspace. We note that in both models, $\Omega(n)$ is a natural lower bound. This is because verifying that a value $\mathcal{L}$ is the n^{th} minimum requires $\Theta(n)$ time—one must at least inspect the n nodes with value at most $\mathcal{L}$—which can be done via straightforward depth-first search.

A simple selection strategy is to use the best-first rule,[2] 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. Moreover, the space complexity becomes $\Omega(n)$ in general when using the best-first rule, because essentially all the explored nodes might need to be kept in memory. We note that irrespective of computational considerations on the agent, either in terms of working memory or running time restrictions, minimizing the total travel distance in explorable heap selection remains a challenging online problem.

Improving on the best-first strategy, Karp, Saks and Wigderson [2] gave a randomized algorithm with expected cost $n \cdot \exp(O(\sqrt{\log(n)}))$ using $O(\sqrt{\log(n)})$ working

[1] [2] did not give the problem a name, so we have attempted to give a descriptive one here.

[2] Frederickson's algorithm [3] is in fact a highly optimized variant of this rule.

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 uses only 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 happen to be 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 (IPs), 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 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. The algorithm can also be used to solve mixed-integer linear programs (MIPs), where some of the variables are allowed to be continuous.

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., [4–7].

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, a disadvantage of this rule is that searches using it might use 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

[2]. 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.

Nowadays, computers have a lot of memory available. This usually makes it feasible 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 [9]. For example, this can be the case when the LP-solver makes use of the LU-factorization of the optimal basis matrix [10]. Through the use of dynamic algorithms, computing this can be done faster if a factorization for a similar LP-basis is known [11]. 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 [2]. Their result was later applied to prove an upper bound on the parallel running time of branch-and-bound [13].

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 [3].

The explorable heap selection problem can be thought of as a *search game* [14] 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 [15] 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 [16, 17]. In some of the used models the objective is to minimize the distance traveled [?], [18–20]. Other models are about minimizing the amount of used memory [21]. What distinguises the explorable heap selection problem from these problems is the information that the graph is a heap and that the ordinal of the target is known. This can allow an algorithm to rule out certain locations for the target. Because of this additional information, the techniques used here do not seem to be applicable to these other problems.

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

We introduce in this section 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. Moreover, for each node in the tree, the values of its children are larger than its own value. Hence, for every $v_1, v_2 \in V$ such that v_1 is an ancestor of v_2, we have that $\text{val}(v_2) > \text{val}(v_1)$. The binary tree T is thus a heap.

The algorithmic problem we are interested in is finding 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 they explore 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.

We now introduce a few notations used throughout the paper.

- 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 $\textsf{SELECT}^T(n)$. 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 $\mathcal{V} \leq \mathsf{SELECT}^T(n)$ and *bad* otherwise. Similarly, we call a node $v \in T$ *good* if $\text{val}(v) \leq \mathsf{SELECT}^T(n)$ and *bad* otherwise.
- We will use $[k]$ to refer to the set $\{1, \ldots, k\}$.
- When we write $\log(n)$, we assume the base of the logarithm to be 2.

For a given value $\mathcal{V} \in \mathbb{R}$, it is easy to check whether it is good in $O(n)$ time: start a depth first search at the root of the tree, turning back each time a value strictly greater than $\mathcal{V}$ is encountered. In the meantime, count the number of values below $\mathcal{V}$ found so far and stop the search if more than n values are found. If the number of values below $\mathcal{V}$ found at the end of the procedure is at most n, then $\mathcal{V}$ is a good value. This procedure is described in more detail later in the DFS subroutine.

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 write the subtree $T^{(v)}$ as part of the input for simplicity of presentation.

We will sometimes want to pick a value uniformly at random from a set of values $\{\mathcal{V}_1, \ldots, \mathcal{V}_k\}$ 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 $\{X_1, \ldots, X_k\}$ 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 $\{\mathcal{V}_1, \ldots, \mathcal{V}_k\}$.

3 A new algorithm

The authors of [2] 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 are able to decrease the space requirements. This way, they obtain 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 using a deterministic routine by [22] to achieve 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 fun-

damentally 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 [2], 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 and branching rule.

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.*

As mentioned above, checking whether a value v is good can be done in $O(n)$ time by doing a depth-first search with cutoff value $\text{val}(v)$ that returns when more than n good nodes are found. For a set of k values, we can determine which of them are good in $O(\log(k)n)$ time by performing a binary search.

The explorable heap selection problem can be seen as the problem of finding all n good nodes. Both our method and that of [2] 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. Importantly, the term 'good node' is always used with respect to the current call to EXTEND. So, a node might be good in one recursive call, but not good in another.

In [2] 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 $n = 1, 2, 4, 8, \ldots$. 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 $n = 1, 2, 4, 8, \ldots$. 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.

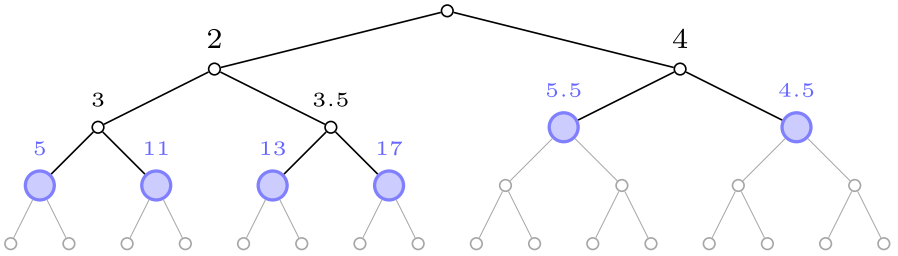


Fig. 1 An illustration of $R(T, \mathcal{L}_0)$ with $\mathcal{L}_0 = 4$. The number above each vertex is its value, the blue nodes are $R(T, \mathcal{L}_0)$, whereas the subtree above is $T_{\mathcal{L}_0}$

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 Subroutines

We first describe three subroutines that will be used in our main algorithm.

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.

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

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

This algorithm visits only nodes in $T_{\mathcal{L}}$ or its direct descendants and its running time is $O(n)$. 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 an initial fixed lower bound $\mathcal{L}_0 \in \mathbb{R}$ on the value of $\textsf{SELECT}^T(n)$. 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

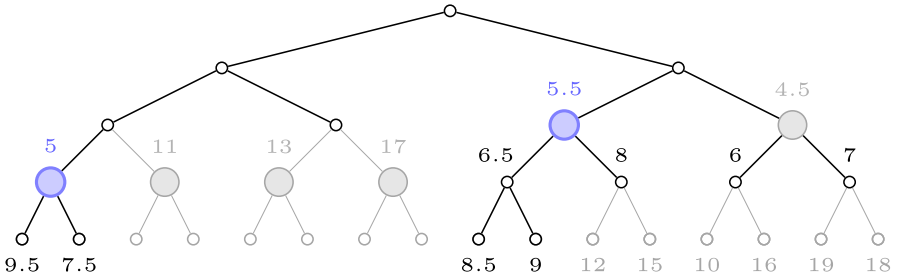


Fig. 2 An illustration of the ROOTS procedure with $\mathcal{L}_0 = 4$, $\mathcal{L} = 7$ and $\mathcal{U} = 10$. Only two active roots remain, and are both colored in blue. The other roots are considered killed since all the good values have been found in their subtrees

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:

$$R(T, \mathcal{L}_0) := \left\{ r \in T \backslash T_{\mathcal{L}_0} \;\middle|\; r \text{ is a child of a node in } T_{\mathcal{L}_0} \right\}$$

In other words, these are all the vertices in T one level deeper in the tree than $T_{\mathcal{L}_0}$, see Fig. 1 for an illustration. 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 $\mathsf{SELECT}^T(n)$. These bounds $\mathcal{L}$ and $\mathcal{U}$ will be variables being updated during the execution of the main algorithm, where $\mathcal{L}$ will be increasing and $\mathcal{U}$ will be decreasing. More precisely, $\mathcal{L}$ will be the largest value that the main algorithm has certified being at most $\mathsf{SELECT}^T(n)$, whereas $\mathcal{U}$ will be the smallest value that the algorithm has certified being at least that. A key observation is that these lower and upper 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. The only roots that the main algorithm needs to consider, when $\mathcal{L}$ and $\mathcal{U}$ are given, are thus the following.

$$\textsc{Roots}(T, \mathcal{L}_0, \mathcal{L}, \mathcal{U}) := \left\{ r \in R(T, \mathcal{L}_0) \mid \exists w \in T^{(r)} \text{ with } \mathrm{val}(w) \in (\mathcal{L}, \mathcal{U}) \right\} \quad (1)$$

This subroutine can be implemented as follows. Run a depth first search starting at the root of T. Once a node $r \in T$ with $\mathrm{val}(r) > \mathcal{L}_0$ is encountered, the subroutine marks that vertex r as belonging to $R(T, \mathcal{L}_0)$. The depth first search continues deeper in the tree until finding a node $w \in T^{(r)}$ with $\mathrm{val}(w) > \mathcal{L}$. At this point, if $\mathrm{val}(w) < \mathcal{U}$, then the search directly returns to r without exploring any additional nodes in $T^{(r)}$ and adds r to the output. If however $\mathrm{val}(w) \geq \mathcal{U}$, then the search continues exploring $T^{(r)}_{\mathcal{L}}$ (and its direct descendants) by trying to find a node w with $\mathrm{val}(w) \in (\mathcal{L}, \mathcal{U})$. In case the algorithm explores all of $T^{(r)}_{\mathcal{L}}$ with its direct descendants, and it turns out that no such node exists (i.e. every direct descendant w of $T^{(r)}_{\mathcal{L}}$ satisfies $\mathrm{val}(w) \geq \mathcal{U}$), then r is not added to the output.

This procedure takes time $O(|T_{\mathcal{L}}|)$, i.e. proportional to the number of nodes in T with value at most $\mathcal{L}$. Since the procedure is called only on values $\mathcal{L}$ which are known to be good, the 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 $\textsc{Roots}(T, \mathcal{L}_0, \mathcal{L}, \mathcal{U})$ 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 $\textsc{GoodValues}$ takes as input a tree T, a subtree $T^{(r)}$ for a node $r \in T$, a value $\mathcal{L}' \in \mathbb{R}_{\geq 0}$ and an integer $n \in \mathbb{N}$. The procedure then analyzes the set

$$S := \left\{ \text{val}(w) \;\middle|\; w \in T^{(r)}, \text{val}(w) \leq \mathcal{L}' \right\}$$

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 $\mathcal{U} = \infty$. Notice that this output determines, for each value in S, whether it is good or not. Indeed, any $\mathcal{V} \in S$ is good if and only if $\mathcal{V} \leq \mathcal{L}$, and is bad if and only if $\mathcal{V} \geq \mathcal{U}$.

The implementation is as follows. Start by initializing the variables $\mathcal{L} = -\infty$ and $\mathcal{U} = \mathcal{L}'$. These variables correspond to lower and upper bounds on $\mathsf{SELECT}^T(n)$. Loop through the values in

$$S' := \left\{ \text{val}(w) \mid w \in T^{(r)},\ \mathcal{L} < \text{val}(w) < \mathcal{U} \right\}$$

using a depth first search starting at r and sample one value $\mathcal{V}$ uniformly randomly from that set. Check whether $\mathcal{V}$ is a good value by calling $\text{DFS}(T, \mathcal{V}, n)$. If it is good, update $\mathcal{L} = \mathcal{V}$. If it is bad, update $\mathcal{U} = \mathcal{V}$. Continue this procedure until S' is empty, i.e. $|S'| = 0$. If, at the end of the procedure, $\mathcal{L} = \mathcal{L}' = \mathcal{U}$, then set $\mathcal{U} = \infty$. The output is thus:

$$\textsc{GoodValues}(T, T^{(r)}, \mathcal{L}', n) := \{\mathcal{L}, \mathcal{U}\}$$

where

$$\mathcal{L} := \max \left\{ \mathcal{V} \in S \mid \mathcal{V} \leq \mathsf{SELECT}^T(n) \right\},$$
$$\mathcal{U} := \min \left\{ \mathcal{V} \in S \mid \mathcal{V} > \mathsf{SELECT}^T(n) \right\}.$$

Sampling a value from S' takes $O(|S|)$ time. Checking whether a sampled valued is good takes $O(n)$ time. In expectation, the number of updates before the set S' is empty is $O(\log(|S|))$, leading to an expected total running time of $O((|S| + n)\log(|S|))$. As we will later see in the proof of Lemma 3, we will only end up making calls $\textsc{GoodValues}(T, T^{(r)}, \mathcal{L}', n)$ with parameters $T^{(r)}$ and $\mathcal{L}'$ satisfying $\text{DFS}(T^{(r)}, \mathcal{L}') = O(n)$. Since $|S| = \text{DFS}(T^{(r)}, \mathcal{L}')$, this leads to an expected running time of $O(n \log(n))$.

The procedure can be implemented in $O(1)$ space, since the only values needed to be kept in memory are val(v) (where v is the root of the tree T), 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 The main algorithm

We now present our main algorithm. This algorithm is named SELECT and outputs the n^{th} smallest value in the tree T. A procedure used in SELECT is the EXTEND algorithm, described below, which assumes that at least $n/2$ good nodes have already been found in the tree, and also outputs the n^{th} smallest one.

Algorithm 1 The SELECT procedure

```
1: Input : n ∈ ℕ
2: Output : SELECT(n), the n^th smallest value in the heap T.
3: procedure SELECT(n)
4:     k ← 1
5:     L ← val(v)  // v is the root of the tree T
6:     while k < n do
7:         if k < n/2 then
8:             k' ← 2k
9:         else
10:            k' ← n
11:        end if
12:        L ← EXTEND(T, k', k, L)
13:        k ← k'
14:    end while
15:    return L
16: end procedure
```

Let us describe a few invariants from the EXTEND procedure.

- $\mathcal{L}$ and $\mathcal{U}$ are respectively lower and upper bounds on $\mathsf{SELECT}^T(n)$ during the whole execution of the procedure. More precisely, $\mathcal{L} \leq \mathsf{SELECT}^T(n)$ and $\mathcal{U} > \mathsf{SELECT}^T(n)$ at any point, and hence $\mathcal{L}$ is good and $\mathcal{U}$ is bad. The integer k counts the number of values $\leq \mathcal{L}$ in the full tree T.
- No root can be randomly selected twice. This is ruled out by the updated values of $\mathcal{L}$ and $\mathcal{U}$, and the proof can be found in Theorem 2.
- 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.
- From the definition of k' and c one can see that $k' \geq k + c$. Combined with the previous invariant, we see that $c' \leq n - k$.
- 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

Algorithm 2 The EXTEND procedure

Input: T: tree which is to be explored.
$n \in \mathbb{N}$: total number of good values to be found, satisfying $n \geq 2$.
$k \in \mathbb{N}$: number of good values already found, satisfying $k \geq n/2$.
$\mathcal{L}_0 \in \mathbb{R}$: value satisfying $\text{DFS}(T, \mathcal{L}_0, n) = k$.
Output: the n^{th} smallest value in T.

procedure EXTEND($T, n, k, \mathcal{L}_0$)
  $\mathcal{L} \leftarrow \mathcal{L}_0$
  $\mathcal{U} \leftarrow \infty$
  while $k < n$ **do**
    $r \leftarrow$ random element from ROOTS($T, \mathcal{L}_0, \mathcal{L}, \mathcal{U}$)
    $\mathcal{L}' \leftarrow \max(\mathcal{L}, \text{val}(r))$
    $k' \leftarrow \text{DFS}(T, \mathcal{L}', n)$ // count the number of values $\leq \mathcal{L}'$ in T
    $c \leftarrow \text{DFS}(T^{(r)}, \mathcal{L}', n)$ // counting the number of values $\leq \mathcal{L}'$ in $T^{(r)}$
    $c' \leftarrow \min(n - k' + c, 2c)$ // increase the number of values to be found in $T^{(r)}$
    while $k' < n$ **do** // loop until it is certified that $\text{SELECT}^T(n) \leq \mathcal{L}'$
      $\mathcal{L}' \leftarrow$ EXTEND($T^{(r)}, c', c, \mathcal{L}'$)
      $k' \leftarrow \text{DFS}(T, \mathcal{L}', n)$
      $c \leftarrow c'$
      $c' \leftarrow \min(n - k' + c, 2c)$
    end while
    $\tilde{\mathcal{L}}, \tilde{\mathcal{U}} \leftarrow$ GOODVALUES($T, T^{(r)}, \mathcal{L}', n$) // find the good values in $T^{(r)}$
    $\mathcal{L} \leftarrow \max(\mathcal{L}, \tilde{\mathcal{L}})$
    $\mathcal{U} \leftarrow \min(\mathcal{U}, \tilde{\mathcal{U}})$
    $k \leftarrow \text{DFS}(T, \mathcal{L}, n)$ // compute the number of good values found in T
  end while
  return $\mathcal{L}$
end procedure

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 $T^{(r)}$.

3.3 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 We show $\mathcal{L} = \text{SELECT}^T(n)$ holds at the end of Algorithm 2, i.e. the EXTEND procedure. Correctness of Algorithm 1, i.e. the SELECT procedure, then clearly follows from that. First, notice that $\mathcal{L}$ is always set to the first output of the procedure GOODVALUES, which is always the value of a good node in T, implying

$$\mathcal{L} \leq \text{SELECT}^T(n)$$

at any point during the execution of the algorithm. Since the outer while loop ends when at least n good nodes in T have value at most $\mathcal{L}$, we get

$$\mathcal{L} \geq \text{SELECT}^T(n),$$

which implies that when the algorithm terminates it does so with the correct value.

It remains to prove that the algorithm terminates. We observe that every recursive call $\mathcal{L}' \leftarrow \text{EXTEND}(T^{(r)}, c', c, \mathcal{L}')$ strictly increases the value of $\mathcal{L}'$, ensuring that at least one extra value in T is under the increased value. This implies that k' strictly increases every iteration of the inner while loop, thus ensuring that this loop terminates.

To see that the outer loop terminates, we observe that after each iteration the set $\text{ROOTS}(T, \mathcal{L}_0, \mathcal{L}, \mathcal{U})$ shrinks by at least one element. As soon as this set is empty, there will be no more roots with unexplored good values in their subtrees, so $k = n$ and the algorithm terminates. □

3.4 Running time analysis

In order to prove a $O(n \log(n^3))$ running time bound for the $\text{SELECT}(n)$ procedure, we will show that the running time of the EXTEND procedure with parameters n and k is $O((n-k)\log(n)^3) + O(n \log(n)^2)$.

The main challenge in analyzing the running time of EXTEND is in dealing with the cost of the recursive subcalls on line 16. For this we rely on an important idea, formalized in Lemma 1, stating that if the parent call with parameters n and k makes $z \in \mathbb{N}$ recursive calls with parameters $(n_1, k_1), \ldots, (n_z, k_z)$, then $\sum_{i=1}^{z}(n_i - k_i) \leq n - k$ in expectation over the random choices of the algorithm.

A second insight is that the outermost while loop on line 9 is executed at most $O(\log(n))$ times in expectation, which is shown in Lemma 2. The first lemma allows to show that the running time of the EXTEND procedure on the recursive part is $O((n-k)\log(n)^3)$, through an induction proof. The second lemma helps to show that the running time of the EXTEND procedure on the non-recursive part is $O(n \log(n)^2)$. The running time analysis of EXTEND is formally done in Lemma 3. Finally, the running time of $O(n \log(n^3))$ for the $\text{SELECT}(n)$ procedure then follows in Theorem 3.

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

Lemma 1 *Let z be the number of recursive calls with $k \geq 1$ that are done in the main loop of* $\text{EXTEND}(T, n, k, \mathcal{L}_0)$*. For every $i \in [z]$, let n_i and k_i be the values that are given as second and third parameters to the ith such subcall. It holds that:*

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

Proof For simplicity of notation, let us denote the set of roots at the beginning of the execution of the algorithm by $\mathcal{R} := \text{ROOTS}(T, \mathcal{L}_0, \mathcal{L}, \mathcal{U})$, where $\mathcal{L} = \mathcal{L}_0$ and $\mathcal{U} = \infty$ at initialization. An important observation is that, once a root $r \in \mathcal{R}$ is randomly selected on line 10, all the recursive calls under it (i.e. with its subtree $T^{(r)}$ as first parameter) on line 16 are consecutive. The last such recursive call ensures that all the good values in $T^{(r)}$ are found and sets $\mathcal{L}$ and $\mathcal{U}$ to respectively be the largest good value and smallest bad value in $T^{(r)}$. From then on, this root leaves the updated set $\text{ROOTS}(T, \mathcal{L}_0, \mathcal{L}, \mathcal{U})$ by (1) and will thus never be again considered in the random choice on line 10. For every $r \in \mathcal{R}$, let us define the set:

$$C(r) = \Big\{ i \in [z] \text{ s.t. the } i\text{th recursive call is under root } r \Big\}$$

and let us denote by $S_r \in \mathbb{N}$ the total number of good values in its subtree $T^{(r)}$. Our goal is to show that:

$$\mathbb{E}\Big[\sum_{i \in C(r)} (n_i - k_i) \Big] \leq S_r \qquad \forall r \in \mathcal{R}. \tag{2}$$

Clearly, this would imply the lemma, since the total number of good values to be found is $\sum_{r \in \mathcal{R}} S_r = n-k$. For convenience, we define this number to be $p := n-k$. We now order the good values to be found and denote them as follows: $\mathcal{V}_1 < \mathcal{V}_2 < \cdots < \mathcal{V}_p$. Each value $\mathcal{V}_k$ is to be found in the subtree of a certain root that we denote by $r(\mathcal{V}_k) \in \mathcal{R}$.

We first show that the claim (2) holds for any root $r \in \mathcal{R}$ such that $r \neq r(\mathcal{V}_p)$. Let us thus fix such a root $r \neq r(\mathcal{V}_p)$. The key observation is that, since the random choice on line 10 is uniform, and since $r(\mathcal{V}_p)$ will always be among the active roots, the subtree of the root $r(\mathcal{V}_p)$ will be explored before the subtree of root r with probability at least a half. In that case, no recursive calls will be made under root r. This holds since the updated values $\mathcal{L}$ and $\mathcal{U}$ after the iteration of $r(\mathcal{V}_p)$ ensure that r leaves ROOTS$(T, \mathcal{L}_0, \mathcal{L}, \mathcal{U})$ by (1) and is thus not considered in the random choice in later iterations. If the root r is however considered before $r(\mathcal{V}_p)$, which happens with probability at most a half, then $\sum_{i \in C(r)} (n_i - k_i) \leq 2S_r$, since the sum is telescoping and the parameters k_i and n_i at most double at each step on line 19 until all good values in $T^{(r)}$ are found. Hence, we get that

$$\mathbb{E}\Big[\sum_{i \in C(r)} (n_i - k_i) \Big] \leq \frac{1}{2}\, 0 + \frac{1}{2}\, 2S_r \leq S_r. \tag{3}$$

It remains to show that claim (2) holds for the root $r(\mathcal{V}_p)$ under which the largest good value lies. In that case, let us denote by $\mathcal{V}_j$ the largest good value lying in a subtree of a different root $r(\mathcal{V}_j) \neq r(\mathcal{V}_p)$. We also denote by $\{r(\mathcal{V}_j) \prec r(\mathcal{V}_p)\}$ the probabilistic event that $r(\mathcal{V}_j)$ is considered before $r(\mathcal{V}_p)$ in the random choices of the algorithm. By our choice of $\mathcal{V}_j$ and $\mathcal{V}_p$, this event happens with probability exactly a half. Moreover, if this event happens, all the good values outside of $T^{(r(\mathcal{V}_p))}$ will have been found after exploring $T^{(r(\mathcal{V}_j))}$. This means that, when the algorithm considers $r(\mathcal{V}_p)$, it knows that there remain at most $p-j$ values to be found. That is, we will have $C(r(\mathcal{V}_p)) = \{t, \ldots, z\}$ for some t, such that $k_t \geq S_{r(\mathcal{V}_p)} - (p-j)$ and $n_z \leq S_{r(\mathcal{V}_p)}$, leading to

$$\mathbb{E}\Big[\sum_{i \in C(r(\mathcal{V}_p))} (n_i - k_i) \;\Big|\; r(\mathcal{V}_j) \prec r(\mathcal{V}_p) \Big] \leq S_{r(\mathcal{V}_p)} - \big(S_{r(\mathcal{V}_p)} - (p-j)\big) = p - j, \tag{4}$$

where we have again used the fact that the sum is telescoping.

We now consider the event $\{r(\mathcal{V}_p) \prec r(\mathcal{V}_j)\}$ and distinguish two cases. Suppose that the penultimate call $i \in C(r(\mathcal{V}_p))$ finds a good value which is bigger than $\mathcal{V}_j$.

By a similar argument as above, the algorithm does not double in the last step, but truncates due to line 19, meaning that $\sum_{i\in C(r(\mathcal{V}_p))}(n_i - k_i) = S_{r(\mathcal{V}_p)}$ holds in this case. Combining this with (4) and using the fact that the last $p - j$ values are under root $r(\mathcal{V}_p)$, we get

$$\mathbb{E}\Big[\sum_{i\in C(r(\mathcal{V}_p))}(n_i - k_i)\Big] \leq \frac{1}{2}(p-j) + \frac{1}{2}S_{r(\mathcal{V}_p)} \leq S_{r(\mathcal{V}_p)}.$$

Suppose now that the penultimate call $i \in C(r(\mathcal{V}_p))$ finds a good value which is smaller than $\mathcal{V}_j$. This means that the number of good values found in $T^{(r(\mathcal{V}_p))}$ is at most $S_{r(\mathcal{V}_p)} - (p - j)$ at that point. The last call $i \in C(r(\mathcal{V}_p))$ then doubles the parameters, meaning that $\sum_{i\in C(r(\mathcal{V}_p))}(n_i - k_i) \leq 2\,(S_{r(\mathcal{V}_p)} - (p - j))$ holds, due to the fact that the sum is telescoping. Combining this with (4) leads to

$$\mathbb{E}\Big[\sum_{i\in C(r(\mathcal{V}_p))}(n_i - k_i)\Big] \leq \frac{1}{2}(p-j) + S_{r(\mathcal{V}_p)} - (p-j) \leq S_{r(\mathcal{V}_p)}.$$

□

We now bound the expected number of iterations of the outermost while-loop.

Lemma 2 *The expected number of times that the outermost while-loop (at line 9) is executed by the procedure* EXTEND *is at most* $O(\log(n))$.

Proof Let $r_1, \ldots, r_m$ denote the roots returned by ROOTS$(T, \mathcal{L}_0, \mathcal{L}_0, \infty)$. For $j \in [m]$, let ℓ_j and u_j respectively denote the largest good value and the smallest non-good value under root r_j. Let $A_\ell(\mathcal{L}) := \{r_j : \ell_j > \mathcal{L}\}$ and $A_u(\mathcal{U}) := \{r_j : u_j < \mathcal{U}\}$. Observe that ROOTS$(T, \mathcal{L}_0, \mathcal{L}, \mathcal{U}) = A_\ell(\mathcal{L}) \cup A_u(\mathcal{U})$ for any $\mathcal{L} \leq \mathcal{U}$.

Let $\mathcal{L}_i$ and $\mathcal{U}_i$ denote the values of $\mathcal{L}$ and $\mathcal{U}$ at the start of the ith iteration. After an iteration i in which root r_j was selected, the algorithm updates $\mathcal{L}$ and $\mathcal{U}$ such that $\mathcal{L}_{i+1} = \max(\mathcal{L}, \ell_j)$ and $\mathcal{U}_{i+1} = \min(\mathcal{U}, u_j)$. Observe that $\mathcal{L}_i$ is nondecreasing and that $\mathcal{U}_i$ is nonincreasing.

We will now show that if a root from $A_\ell(\mathcal{L}_i)$ is selected in iteration i, then the expected size of $A_\ell(\mathcal{L}_{i+1})$ is at most half that of $A_\ell(\mathcal{L}_i)$. This will imply that in expectation only $\log(n)$ iterations are needed to make $|A_\ell(\mathcal{L})| = 1$.

Let $\mathcal{F}_i$ be the filtration containing all information up until iteration i. Let X_i be a random variable denoting the value of $|A_\ell(\mathcal{L}_i)|$. Let $(s_k)_{k\geq 1}$ be the subsequence consisting of iteration indices i in which a root from $A_\ell(\mathcal{L}_i)$ is selected. Because roots are selected uniformly at random, we have $\mathbb{E}[X_{s_{k+1}} \mid \mathcal{F}_{s_k}] \leq \frac{1}{2}X_{s_k}$.

Let $Y_i = \max(\log(X_i), 0)$. Note that when $Y_{s_k} \geq 1$, we have $\mathbb{E}[Y_{s_{k+1}} \mid \mathcal{F}_{s_k}] = \mathbb{E}[\log(X_{s_{k+1}}) \mid \mathcal{F}_{s_k}] \leq \log(\mathbb{E}[X_{s_{k+1}} \mid \mathcal{F}_{s_k}]) \leq Y_{s_k} - 1$. Let τ be the smallest k such that $Y_{s_k} = 0$. Note that τ is the number of iterations i in which a root from $A_\ell(\mathcal{L}_i)$ is selected, and hence $\tau \leq n$. The sequence $(Y_{s_k} + k)_{k=1,\ldots,\tau}$ is therefore a supermartingale and τ is a stopping time. By the martingale stopping theorem [23, Theorem 12.2], we have $\mathbb{E}[\tau] = \mathbb{E}[Y_{s_\tau} + \tau] \leq \mathbb{E}[Y_{s_1} + 1] = \log(m) + 1$.

Now we have shown that in expectation at most $\log(m) + 1$ iterations i are needed in which roots from $A_\ell(\mathcal{L}_i)$ are considered. The same argument can be repeated for

$A_u(\mathcal{U})$. Since in every iteration a root from $A_\ell(\mathcal{L})$ or $A_u(\mathcal{U})$ is selected, the expected total number of iterations is at most $2\log(m) + 2$. This directly implies the lemma as $m \leq |T_{\mathcal{L}}| + 1 \leq n + 1$. □

We are now able to prove the running time bound for the EXTEND procedure.

Lemma 3 *Let $R(T, n, k)$ denote the running time of a call* 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 $\lceil \log(n) \rceil \leq r - 1$. Let p be the number of iterations of the outer-most while-loop that are executed.

We will first consider the running time induced by the base call itself, excluding any recursive subcalls. Note that all of this running time is incurred by the calls to the procedures DFS, ROOTS and GOODVALUES, plus the cost of moving to the corresponding node before each of these calls. In the base call, the procedure will only move between nodes that are among the ones with the n smallest values, or the nodes directly below them. For this reason, we can upper bound the cost of each movement action by a running time of $O(n)$.

- On line 12, 13, 24 each call DFS incurs a running time of at most $O(n)$. Each of these lines will be executed p times, incurring a total running time of $O(pn)$.
- On line 17 each call DFS$(T, \mathcal{L}', n)$ incurs a running time of at most $O(n)$. The code will be executed $O(p\log(n))$ times since c' doubles after every iteration of the inner loop and never grows larger than n, thus incurring a total running time of $O(pn\log(n))$.
- The arguments $T^{(r)}$ and $\mathcal{L}'$ of the call to GOODVALUES on line 21 satisfy DFS$(T^{(r)}, \mathcal{L}') = c \leq c' \leq n$. Hence, the running time of this procedure is $O(n\log(n))$ time. The line is executed at most p times, so the total running time incurred is $O(pn\log(n))$.

Adding up all the running times listed before, we see that the total running time incurred by the non-recursive part is $O(pn\log(n))$. By Lemma 2, $\mathbb{E}[p] \leq \log(n)$. Hence, we can choose C such that the expected running time of the non-recursive part is bounded by

$$Cn\log(n)^2.$$

Now we move on to the recursive part of the algorithm. All calls to 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 of recursive calls to 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 $n/2 \geq n - k \geq n_i \geq 2$ for all i. By the induction hypothesis, the

expectation of the recursive part of the running time is:

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

Here we used Lemma 1 as well as the fact that $\sum_{i=1}^{z} n_i \leq 4(n-k)$. To see why the latter inequality is true, consider an arbitrary root r that has S_r values under it that are good (with respect to the base call). Now $\sum_{i=1}^{z} \mathbf{1}_{\{T_i = T^{(r)}\}} n_i \leq \sum_{i=2}^{\lceil \log(S_r+1)\rceil} 2^i \leq 2^{\lceil \log(S_r+1)\rceil + 1} \leq 4S_r$. 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

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

□

This now implies the desired running time for the procedure SELECT.

Theorem 3 *The procedure* SELECT(n) *runs in expected* $O(n\log(n)^3)$ *time.*

Proof The key idea is that SELECT calls EXTEND$(T, k', k, \mathcal{L})$ at most $\lceil \log(n)\rceil$ times with parameters $(k', k) = (2^i, 2^{i-1})$ for $i \in \{1, \ldots, \lceil \log(n)\rceil\}$. By Lemma 3, the running time of SELECT can thus be upper bounded by

$$\begin{aligned}
\sum_{i=1}^{\lceil \log(n)\rceil} \mathbb{E}[R(T, 2^i, 2^{i-1})] &\leq 5C\log(n)^3 \sum_{i=1}^{\lceil \log(n)\rceil} (2^i - 2^{i-1}) + \sum_{i=1}^{\lceil \log(n)\rceil} Cn\log(n)^2 \\
&= O(n\log(n)^3).
\end{aligned}$$

□

3.5 Space complexity analysis

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

Theorem 4 *The procedure* SELECT(n) *runs in* $O(\log(n))$ *space.*

Proof Observe that it is enough to prove that the statement holds for EXTEND$(T, n, k, \mathcal{L})$ with $k \geq n/2$, since the memory can be freed up (only keeping the returned value in memory) after every call to EXTEND in the SELECT(n) algorithm.

Moreover, observe that the subroutines DFS, ROOTS and GOODVALUES all require $O(1)$ memory, as argued in their respective analyses. Any call EXTEND$(T, n, k, \mathcal{L})$ only makes recursive calls EXTEND$(T^{(r)}, \hat{n}, \hat{k}, \hat{\mathcal{L}})$ with $1 \leq \hat{n} \leq n - k \leq \frac{1}{2}n$. So the depth of the recursion is at most $\log(n)$, and the space complexity of the algorithm is $O(\log(n))$. □

4 Lower bound

No lower bound is known for the running time of the selection problem on explorable heaps. However, we will show that any (randomized) algorithm with space complexity at most s, has a running time of at least $\Omega(n \log_s(n))$. Somewhat surprisingly, 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 a totally ordered set W is given, which is partitioned into (S_A, S_B). There are two agents A and B, that have access to the sets of values in S_A and S_B respectively. 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, that is the $(n + 1)$th smallest value of W.

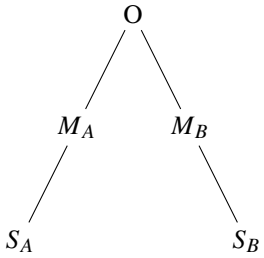
Because the players only have access to their own values, they need to communicate. For this purpose they use a protocol, that can consist of multiple rounds. In every odd round, player A can do computations and send units of information to player B. In every even round, player B does computations and 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. For example, the algorithm cannot do addition or multiplication on the values.

We will now reduce the median computation problem to the explorable heap selection problem.

Lemma 4 *If there is a randomized algorithm that solves* SELECT$(3n)$ *in* $f(n)n$ *time and* g *space, then there is a randomized protocol for median computation that uses* $f(n)/2$ *rounds in each of which at most* g *units of information are sent.*

Proof Consider arbitrary sets S_A and S_B with $|S_A| = n + 1$ and $|S_B| = n$ and $S_A \cap S_B = \emptyset$. Introduce a new element O, such that $O < x$ for all $x \in S_A \cup S_B$. Let M_A and M_B be two sets with $|M_A| = |M_B| + 1 = n$ and $O < y < x$ for all $y \in M_A \cup M_B$ and $x \in S_A \cup S_B$.

Let us write $S_A = \{a_1, \ldots, a_{n+1}\}$. Now consider a subtree for which the root node has value a_1. For every $i \in \{1, \ldots, n\}$, let every node that has value a_i have a child with value a_{i+1} and another child with some value that is larger than any value in $S_A \cup S_B \cup M_A \cup M_B$. We will call this a *trail* of S_A.



Now we will construct a labeled tree in the following way: create a tree with a root node of value O. Attach a trail of M_A as the left child of this root and a trail of M_B as the right child. Attach a trail of S_A as a child of the largest node in M_A and do the same for a trail of M_B under the largest node of S_B. The resulting tree will now look as shown in the above picture.

Observe that the $3n$th smallest value in this tree is the median of $S_A \cup S_B$. Now we can view the selection algorithm as an algorithm for median computation if we consider moving between S_A and S_B in the tree as sending the g units of information that are in memory to the other player. Because moving between the two sets takes at least $2n$ steps, the number of rounds of rounds in the corresponding communication protocol is at most $\frac{f(n)n}{2n} = f(n)/2$, proving the statement. □

We now move on to proving a lower bound for the median computation problem. The following lemma will play a key part in the proof.

Lemma 5 *Let $S \subseteq [n]$ be a randomly distributed subset of $[n]$ with size $|S| \leq k \leq n$. Then for $\ell \leq \frac{n}{8k}$ there exists a length-ℓ interval $\subseteq [n]$ (i.e. $I = \{i, i+1, \ldots, i+\ell-1\}$) such that:* $\Pr[S \cap I \neq \emptyset] \leq \frac{1}{4}$.

Proof Let $\mathcal{I}_\ell$ be the set of length-ℓ intervals in $[n]$. We have $|\mathcal{I}_\ell| = n - \ell + 1$. Observe that any value in $[n]$ is contained in at most ℓ elements of $\mathcal{I}_\ell$. Hence, for any set S of size at most k, there are at most $k \cdot \ell$ elements of $\mathcal{I}_\ell$ that contain any of the elements of S. That is: $|\{I \in \mathcal{I}_\ell : I \cap S \neq \emptyset\}| \leq k \cdot \ell$. This implies that for a randomly distributed set $S \subseteq [n]$ we also have:

$$\begin{aligned}\sum_{I \in \mathcal{I}_\ell} \Pr_S[I \cap S \neq \emptyset] &= \sum_{I \in \mathcal{I}_\ell} \mathbb{E}_S[\mathbf{1}_{I \cap S \neq \emptyset}] = \mathbb{E}_S\left[\sum_{I \in \mathcal{I}_\ell} \mathbf{1}_{I \cap S \neq \emptyset}\right] \\ &= \mathbb{E}_S[|\{I \in \mathcal{I}_\ell : I \cap S \neq \emptyset\}|] \leq k \cdot \ell.\end{aligned}$$

So there must be an $I \in \mathcal{I}_\ell$ with:

$$\Pr_S[I \cap S \neq \emptyset] \leq \frac{k \cdot \ell}{|\mathcal{I}_\ell|} = \frac{k \cdot \ell}{n - \ell + 1} \leq \frac{k \cdot \frac{n}{8k}}{\frac{1}{2}n} = \frac{1}{4}.$$

□

Theorem 5 *Any randomized protocol for median computation that sends at most g units of info per round, takes at least $\Omega(\log_{g+1}(n))$ rounds in expectation.*

Proof We will instead prove the following result for a symmetric version of median computation, because this makes the proof a bit easier. In this setting, we have $|S_A| = |S_B| = n$ and the objective is to find both the nth and the $(n+1)$th smallest element of $S_A \cup S_B$. We will call the set consisting of these two values the 2-*median* of $S_A \cup S_B$ and we will denote it by $\mathsf{2median}(S_A \cup S_B)$. Because this problem can be trivially solved by appending two rounds to any median-computation algorithm, proving a lower bound for this case is sufficient.

Let $g' = g + 1$. We can assume that $g \geq 1$, and hence $g' \geq 2$. We will prove with induction on n that the expected number of rounds to compute the median is at least $\frac{1}{10}\log_{g'}(n) - 9$. For $n < 2^8(g')^2$, this is trivial. Now let $n \geq 2^8(g')^2$. Assume that the claim is true for values strictly smaller than n. We will now prove the claim for n.

Consider an arbitrary randomized algorithm. Let $V_i \subseteq [n]$ be the set of indices of the values that are emitted during round i by one of the two players. Observe that the distribution of the set V_1 does not depend on the input, because player A only has access to his own set of n values that he can compare to each other. Order the values in S_A by increasing order of their values $x_1, \ldots, x_n$. Order the values of S_B in decreasing order as $y_1, \ldots, y_n$. We now describe below how the relative ordering of the x_i's with respect to the y_i's is decided adversarially.

Let $\ell = \lfloor \frac{n}{8g} \rfloor$. From Lemma 5 it follows that there exists an interval $I = \{a, \ldots, a+\ell-1\} \subseteq [n]$ such that $\Pr[V_1 \cap I \neq \emptyset] \leq \frac{1}{4}$. Now let $L = \{1, \ldots, a-1\}$ and $U = \{a+\ell, \ldots, n\}$. Observe that $\{L, I, U\}$ forms a partition of $[n]$. We now order the values in the sets such that we have $y_u < x_l < y_i < x_u < y_l$ for all $l \in L, u \in U, i \in I$. Note that this fixes the ordinal index of any element in $S_A \cup S_B$, except for the elements x_i and y_i for $i \in I$.

Condition on the event that $I \cap V_1 = \emptyset$. Observe that in this case, the results of all comparisons that player 2 can do in the second round have been fixed. Hence, V_2 will be a random subset of $[n]$, whose distribution will not depend on the order of the values $x_a, \ldots, x_{a+\ell-1}$ with respect to $y_1, \ldots, y_n$.

We now do a similar argument for the second player. Let $\ell' = \lfloor \frac{\ell}{8g} \rfloor$. From Lemma 5, there exists an interval $I' = \{a', \ldots, a' + \ell' - 1\} \subseteq I$ such that $\Pr[I' \cap V_2 \neq \emptyset \mid I \cap V_1 = \emptyset] \leq \frac{1}{4}$. Define $L' = \{a, \ldots, a' - 1\}$ and $U' = \{a' + \ell', \ldots, a + \ell - 1\}$. Observe that $\{L', I', U'\}$ forms a partition of I. We now order the values in the sets such that we have $y_u < x_l < y_i < x_u < y_l$ for all $l \in L', u \in U', i \in I'$. Note that we have now fixed the ordinal index of any element in $S_A \cup S_B$, except for the elements x_i and y_i for $i \in I'$.

Because $I' \subseteq I$, we have

$$\Pr[I' \cap (V_1 \cup V_2) \neq \emptyset] \leq \Pr[I \cap V_1 \neq \emptyset] + \Pr[I' \cap V_2 \neq \emptyset \mid S \cap V_1 = \emptyset] \leq \frac{1}{4} + \frac{1}{4} = \frac{1}{2}.$$

Now, let R be the number of rounds that the algorithm takes and define $S'_A = \{x_i : i \in I'\}$ and $S'_B = \{y_i : i \in I'\}$. Observe that $\mathsf{2median}(S_A \cup S_B) = \mathsf{2median}(S'_A \cup S'_B)$. So the algorithm can now be seen as an algorithm to compute the 2-median of $S'_A \cup S'_B$. Let R' be the number of rounds in which elements from the set $S'_A \cup S'_B$ are transmitted. With probability $\phi := Pr[I' \cap (V_1 \cup V_2) = \emptyset] \geq \frac{1}{2}$, no information about S'_A and S'_B

is transmitted in the first two rounds, meaning that

$$\mathbb{E}[R'] \leq \phi\,\mathbb{E}[R-2] + (1-\phi)\,\mathbb{E}[R] = \mathbb{E}[R] - 2\phi \leq \mathbb{E}[R] - 1.$$

Moreover, by our induction hypothesis it follows that R' satisfies:

$$\begin{aligned}\mathbb{E}[R'] &\geq \frac{1}{10}\log_{g'}(|S'_B|) - 9 = \frac{1}{10}\log_{g'}(\ell') - 9 \geq \frac{1}{10}\log_{g'}\left(\frac{n}{(8g)^2} - 2\right) - 9 \\ &\geq \frac{1}{10}(\log_{g'}(n) - 2\log_{g'}(8g') - 2) - 9 \geq \frac{1}{10}\log_{g'}(n) - 10.\end{aligned}$$

The second inequality follows from the definition of ℓ'. The third inequality follows from the fact that $\log_{g'}(x-2) \geq \log_{g'}(x) - 2$ for $x \geq 3$. The last inequality follows from $g' \geq 2$. Consequently, we get that $\mathbb{E}[R] \geq \mathbb{E}[R'] + 1 \geq \frac{1}{10}\log_{g'}(n) - 9$. □

Combining Theorem 5 and Lemma 4 now implies the following.

Theorem 6 *The time complexity of any randomized algorithm for* SELECT(n) *with at most g units of storage is* $\Omega(\log_{g+1}(n)n)$.

Funding 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).

Declarations

Conflict of interest We have no Conflict of interest to disclose.

References

1. Borst, S., Dadush, D., Huiberts, S., Kashaev, D.: A Nearly Optimal Randomized Algorithm for Explorable Heap Selection. In: Integer Programming and Combinatorial Optimization. Lecture Notes in Computer Science, vol. 13904, pp. 29–43. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-32726-1_3
2. Karp, R.M., Saks, M.E., Wigderson, A.: On a search problem related to branch-and-bound procedures. In: 27th Annual Symposium on Foundations of Computer Science, pp. 19–28. IEEE Computer Society, Washington, DC (1986)
3. Frederickson, G.N.: An optimal algorithm for selection in a min-heap. Inf. Comput. **104**(2), 197–214 (1993). https://doi.org/10.1006/inco.1993.1030
4. Linderoth, J.T., Savelsbergh, M.W.P.: A computational study of search strategies for mixed integer programming. INFORMS J. Comput. **2**, 173–187 (1999). https://doi.org/10.1287/ijoc.11.2.173

5. 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
6. 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
7. Balcan, M.-F., Dick, T., Sandholm, T., Vitercik, E.: Learning to branch. ICML (2018)
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). https://doi.org/10.1023/A:1018952429396
9. Achterberg, T.: Constraint integer programming. Ph.D. thesis, TU Berlin (2009)
10. 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
11. 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
12. Gleixner, A.M.: Personal Communication (2022)
13. Pietracaprina, A., Pucci, G., Silvestri, F., Vandin, F.: Space-efficient parallel algorithms for combinatorial search problems. J. Parallel Distrib. Comput. **76**, 58–65 (2015). https://doi.org/10.1016/j.jpdc.2014.09.007
14. Alpern, S., Gal, S.: The theory of search games and rendezvous. International Series in Operations Research and Management Science, vol. 55. Kluwer Academic Publishers, Boston (2003). https://doi.org/10.1007/b100809
15. 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: Foundations of Software Technology and Theoretical Computer Science, pp. 22–36. Springer, Berlin, Heidelberg (1995). https://doi.org/10.1007/3-540-60692-0_38
16. Berman, P.: On-line searching and navigation, pp. 232–241. Springer, Berlin, Heidelberg (1998).https://doi.org/10.1007/BFb0029571
17. Thomas Kamphans: Models and algorithms for online exploration and search. PhD thesis, Rheinische Friedrich-Wilhelms-Universität Bonn (2006). https://hdl.handle.net/20.500.11811/2622
18. Baligács, J., Disser, Y., Heinrich, I., Schweitzer, P.: Exploration of graphs with excluded minors. In: 31st Annual European symposium on algorithms (ESA 2023). Schloss Dagstuhl, Saarbrücken/Wadern (2023). https://doi.org/10.4230/LIPIcs.ESA.2023.11
19. Megow, N., Mehlhorn, K., Schweitzer, P.: Online graph exploration: new results on old and new algorithms. Theoret. Comput. Sci. **463**, 62–72 (2012). https://doi.org/10.1016/j.tcs.2012.06.034
20. Kalyanasundaram, B., Pruhs, K.R.: Constructing competitive tours from local information. Theoret. Comput. Sci. **130**(1), 125–138 (1994). https://doi.org/10.1016/0304-3975(94)90155-4
21. 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
22. Munro, J.I., Paterson, M.S.: Selection and sorting with limited storage. Theoret. Comput. Sci. **12**(3), 315–323 (1980). https://doi.org/10.1016/0304-3975(80)90061-4
23. Mitzenmacher, M., Upfal, E.: Probability and Computing: An Introduction to Randomized Algorithms and Probabilistic Analysis. Cambridge University Press, New York (2005)