CS/CE 412/471 - Project Technical Summary

Musab Kasbati, Meesum Abbas

1 Problem and Contribution

The paper "A nearly optimal randomized algorithm for explorable heap selection" [1] as its name implies addresses the issue of *explorable heap selection*, which involves selecting the nth smallest value in an infinitely large binary heap where the key values can only be accessed by traversing the tree from the root. The complexity of an algorithm for this problem is measured by the total distance traveled in the tree, with each edge having a unit cost. This problem was originally proposed by Karp, Saks and Wigderson as a model to study search strategies for the branch-and-bound algorithm with storage restrictions.

The paper's main algorithmic contribution is a **new randomised algorithm** for explorable heap selection that achieves an expected running time of $O(n \log(n)^3)$ using $O(\log n)$ space against an oblivious adversary. This result **substantially improves the previous best randomised running time**[2], though with a slight increase in the memory usage for this problem, which was $n \cdot \exp(O(\sqrt{\log n}))$ using $O(\sqrt{\log n})$ space. The authors also present a lower bound of $(n \log(n)/\log(\log(n)))$ for any algorithm solving this problem with the same space complexity, indicating that their algorithm is **nearly optimal**.

In essence, the paper tackles the challenge of efficiently finding the nth smallest element in a heap under the constraint that access to the heap's values is limited to tree traversal, i.e. the values of a node are not known until the node is visited. It introduces a new randomised approach that significantly lowers the expected running time compared to prior randomised methods while maintaining a relatively small space complexity. The paper argues for the practical relevance of this problem to node selection in branch-and-bound algorithms, even in settings with ample memory, by highlighting the benefits of locality in search strategies, which the explorable heap selection problem captures by measuring running time in terms of tree travel distance.

2 Algorithmic Description

The algorithm takes as input an infinite binary (min-)heap along with a rank n and outputs the n-th smallest element in the heap.

The algorithm in the paper achieves this by finding all of the n smallest elements. It does so by starting with a subtree of the original heap which originally only contains the root (the smallest element overall), and then doubles the size of this subtree by expanding it through the nodes immediately below the selected subtree, until the number of nodes contained are n. The invariant here is that the nodes in the selected subtree are always *good*.

Good values/nodes are those that are less than the n-th largest element. While we don't know what the n-th largest element is, we can determine if a value is good or not through a depth-first search from the root of the heap. The way this works is that you provide the DFS the search value, and it goes down the heap while the nodes it is coming across have value less than the search value. Once you find a node value larger than the search value, you know that all nodes further down are going to be larger through the min-heap property, and you stop going down this branch. Either the DFS terminates by itself with at most n nodes explored, meaning the search value was *good*, or if it finds $n+1$ nodes less than it, then we know the search value is *not good* = *bad* and we stop.

Another key idea in the functioning of this algorithm is that, whenever it doubles the size of the selected subtree to k, it maintains a range $(\mathcal{L},\mathcal{U})$ in which the k-th largest value is guaranteed to fall in. Initially $\mathcal{L} = \text{root}(val)$, $\mathcal{U} = +\infty$. This range is refined as the heap is explored. Particularly,

when the subtree is to be expanded, a random child that has a value that falls in the range $(\mathcal{L},\mathcal{U})$ is selected. Then, the number of extra *good* nodes that need to be found are searched entirely within this subtree. Here the algorithm is recursively used, i.e. if we need to find j further *good* nodes then we find the j smallest nodes in this subtree by using the algorithm recursively. Note that since the heap is infinite, the subtree will have at least j nodes. Now, we note that the following, the j-th largest value of this subtree is not necessarily the k-th largest value of the original heap. However, we have that it will at least be as large as the k-th largest value of the original heap. So we randomly pick values from this subtree to effectively do a binary search on the largest *good* value ($\tilde{\mathcal{L}}$) in the subtree and the smallest *bad* value ($\tilde{\mathcal{U}}$) in this subtree. We can use these values to update our range where $\mathcal{L} = \max(\mathcal{L}, \tilde{\mathcal{L}}), \mathcal{U} = \min(\mathcal{U}, \tilde{\mathcal{U}})$. This limits the next child we can explore, with the idea being that if the children are selected randomly, roughly half of them will be discarded each turn. This allows the algorithm to quickly converge on a singular value $\mathcal{V}$ which is equal to the k-th largest value. As we keep doubling k each time, the algorithm also quickly converges on the n-th largest value of the original heap, which is the answer of the overall problem.

3 Comparison

The proposed randomized algorithm significantly departs from previous approaches in both structure and performance. Unlike Karp, Saks, and Wigderson's[2] earlier deterministic and randomized algorithms, which rely on a combination of recursive search and heavy memory usage, the new technique adopts a *lightweight, randomized binary search framework* over the set of potential node values. This search strategy avoids the full construction of the subtree of good nodes and instead incrementally extends a frontier by intelligently sampling and pruning the search space.

A limitation of the new method is, however, that it operates under the assumption of an **oblivious adversary**, meaning the heap is fixed before the algorithm begins. This contrasts with previous algorithms that account for *adaptive adversaries*, which can dynamically adjust the heap as the algorithm progresses. While this assumption limits the algorithm's adaptability, it simplifies the design and provides strong performance guarantees in practice, especially in settings like branch-and-bound, where the authors argue that node values are typically determined in advance.

The paper also compares its algorithm with the **best-first search strategy**, which, although optimal in the number of nodes visited $\Theta(n)$, suffers from a poor worst-case travel cost of $\Theta(n^2)$ and a high space complexity $\Omega(n)$, as at least n items must be stored at all times. In contrast, the new approach achieves much better travel efficiency $O(n \log(n)^3)$ while using only $O(\log n)$ space.

4 Data Structures and Techniques

The algorithm makes use of the following data structures and mathematical techniques to efficiently solve the problem:

- **Min-Heap as a Binary Tree**: The problem is modeled using a rooted binary tree where the nodes represent values, and the structure enforces the min-heap property (i.e., values increase along every path from the root).
- **Randomized Algorithms**: The approach heavily relies on randomization, particularly in choosing subtrees to explore, and in binary search over nodes.
- **Depth-First Search (DFS)**: DFS is used to traverse the tree, ensuring that the algorithm explores subtrees efficiently and travels as little distance as possible.
- **Binary Search**: Binary search is used on the list of roots of subtrees, ordered by their corresponding node values, to optimize the search for good nodes.
- **Recursion**: The algorithm employs a recursive strategy to explore the tree and progressively identify good nodes.

5 Implementation Outlook

In theory, the algorithm requires an infinitely large heap, when testing the algorithm, we will have to ensure that our heaps are large enough such that for the execution of the algorithm it is virtually infinite (i.e. the leaves of the heap are never reached). Alternatively, we could modify the algorithm to work on finite heaps once we have developed a stronger comfort with the functioning of the algorithm. Analysis will be a bit challenging due to the random nature of execution, however, averaging over many turns should resolve this concern. Moreover, the algorithm lacks pseudocode for random selection and the GoodValues procedure. We will have to be more particular with interpreting their function through the mathematical/technical definitions.

References

[1] Sander Borst, Daniel Dadush, Sophie Huiberts, and Danish Kashaev. A nearly optimal randomized algorithm for explorable heap selection. *Mathematical Programming*, 210(1):75–96, March 2025.

[2] Richard M Karp, Michael E Saks, and Avi Wigderson. On a search problem related to branch-and-bound procedures. In *FOCS*, pages 19–28, 1986.