Exact Lower Bounds for the Number of Comparisons in Selection

Josua Dörrer
University of Stuttgart, Germany

Konrad Gendle
University of Stuttgart, Germany

Johanna Hofmann
University of Stuttgart, Germany

Julius von Smercek
University of Stuttgart, Germany

Andreas Steding
University of Stuttgart, Germany

Florian Stober
University of Stuttgart, Germany

Abstract

Selection is the problem of finding the i-th smallest element among n elements. We apply computer search to find optimal algorithms for small instances of the selection problem. Using new algorithmic ideas, we establish tighter lower bounds for the number of comparisons required, denoted as $V_i(n)$. Our results include optimal algorithms for n up to 15 and arbitrary i, and for $n = 16$ when $i \leq 6$. We determine the precise values $V_7(14) = 25$, $V_6(15) = V_7(15) = 26$, and $V_8(15) = 27$, where previously, only a range was known.

2012 ACM Subject Classification Theory of computation → Sorting and searching; Mathematics of computing → Combinatorial algorithms

Keywords and phrases selection, lower bounds, exhaustive computer search

Digital Object Identifier 10.4230/LIPIcs.SEA.2025.3

Supplementary Material *Software (Source Code)*: `https://github.com/JGDoerrer/selection_generator`

1 Motivation

The problem of selecting the i-th smallest element in a list of n elements is a well-known problem in computer science called *selection*. Explicitly, we concern ourselves with the optimal worst-case selection of a single element from a set of initially unordered unique elements, measuring the cost by the number of comparisons. We denote this cost as $V_i(n)$.

For selecting the smallest element, optimal algorithms are known with $V_1(n) = n - 1$. For the second smallest element, it is known that $V_2(n) = n - 2 + \lceil \log n \rceil$ [7] (all logarithms are to base 2). In general, the selection problem is solvable in linear time using the median of medians algorithm [2]. Looking at the special case of selecting the median $i = n/2$, the best-known algorithm requires $2.95n$ comparisons [4]. For other values of i, the algorithm in [4] requires fewer comparisons, thus providing a general upper bound. This presents a significant gap compared to the best-known lower bound, which is $(1 + H(i/n)) \cdot n + \Omega(\sqrt{n})$, where $H(x) = x \cdot \log \frac{1}{x} + (1 - x) \log \frac{1}{1-x}$ [1]. For the median, this lower bound is $2 \cdot n - \mathcal{O}(n)$. Paterson conjectured that the lower bound for selecting the median is $n \log_{4/3} 2 \approx 2.41n$ [11]. To improve these bounds toward tightness, it is essential to have known optimal reference points that general approaches can be compared against.

Gasarch, Kelly, and Pugh [5] were the first to use computer search to find optimal selection algorithms for fixed n and i. Oksanen continued this line of work, improving upon the

previously known lower bounds [10]. His results and the computer program he used to obtain them are available on his website [9], but have not been published in a scientific journal.

We will also tackle the selection problem using computer search. We reimplement the minimax algorithm used in [5, 9, 10] and improve upon it by introducing suitable heuristics. Then we explore a different search strategy, the backward search, which is another significant improvement over the minimax algorithm. A quote from Miguel de Cervantes from Don Quijote will hold true for this article: "the journey is better than the inn" [3]. So buckle up.

1.1 Contribution.

In this work, we apply two different search algorithms to finding optimal algorithms for selection. The first, which we will call forward search in the remainder of this article, is based on the minimax algorithm also used by Gasarch et. al. [5] and Oksanen [9, 10]. We introduce a novel pruning criterion based on the notion of compatible solutions, and demonstrate that this leads to a significant improvement compared to the benchmark implementation from [9].

The second algorithm, the backward search, is based on an entirely different idea. Here, the start and endpoint of the search switch places. This type of search has not been applied to the selection problem before, and its efficient application poses several challenges, the solution of which is the main technical contribution of this work.

One such challenge is the use of reduced posets. A poset is reduced by removing all elements that cannot be the i-th smallest. In the forward search this is a straightforward optimization to reduce the search space. To apply this optimization in the backward search, we show that reduced posets are well-behaved, in a way that allows us to efficiently reverse the reduction.

Another challenge, more of an engineering nature, is the need for an efficiently computable normal form. The goal of the normal form is to identify isomorphic posets and thus reduce the size of the search space. A normal form can be computed using the well-known nauty tool [8]. However, a call to `nauty` is expensive compared to the other steps in the backward search. Thus, we spend a lot of effort on being able to avoid a call to `nauty` in many cases when computing the normal form.

Using our two algorithms, we obtained the following results:

1. We confirm most of the values $V_i(n)$ computed by Oksanen and correct an error in his work which states that $V_5(15)$ would be 25 [9]. We show that the optimal algorithm requires one comparison fewer, that is $V_5(15) = 24$.
2. We determined the precise values $V_7(14) = 25$, $V_6(15) = V_7(15) = 26$, and $V_8(15) = 27$. Previously, only a range of values was known for these instances.
3. We computed $V_i(16)$ for $i \leq 6$ and determined a better lower bound for $V_7(16)$ and $V_8(16)$.

2 Fundamentals

2.1 Posets.

A partial order is a reflexive, transitive, and antisymmetric relation. A partially ordered set, short *poset*, is a set Ω with a partial order $P \subseteq \Omega \times \Omega$. Instead of $(a, b) \in P$ we often write $a \leq_P b$. By a slight abuse of notation, we denote the poset by P as well. When necessary, we write Ω_P to refer to the underlying set. Throughout this paper, Ω is finite. By E_n we denote the unordered poset on n elements, where each element is related only to itself. Two posets P and Q are *isomorphic* if there is a bijective mapping $\varphi : \Omega_P \to \Omega_Q$ such that $a \leq_P b \iff \varphi(a) \leq_Q \varphi(b)$ for all $a, b \in \Omega_P$. The *dual* of a poset P is obtained by reversing

the direction of all edges, i.e., $P^\delta = \{ (b, a) \mid (a, b) \in P \}$. Given a poset P, its *Hasse diagram* H is given by the smallest subset $H \subseteq \Omega \times \Omega$ such that P is the reflexive, transitive closure of H. We denote by $P{+}ab$ the transitive closure of $P \cup \{(a, b)\}$. By $P|_{\Omega'}$ we denote the restriction of P to Ω'. The downset of an element a is $D_P(a) = \{ b \in \Omega_P \mid b \leq_P a \}$, and the upset is $U_P(a) = \{ b \in \Omega_P \mid b \geq_P a \}$.

2.2 The Selection Problem.

The selection problem is, given a poset P and an integer i, to determine the i-th smallest of the n elements in Ω_P where we already know the relation P. Note that this means we assume there is a total order on the elements of Ω_P. We do not know this total order but only the suborder P. The suborder P corresponds to the relations known at a certain step of the algorithm. We denote an instance of the *selection problem*, or problem for short, by (P, i). The notion of isomorphism naturally extends to selection problems. For the dual, we define $(P, i)^\delta = (P^\delta, n - i + 1)$. The problem (P, i) is *reduced* if for each element there are at most $i - 1$ smaller elements and at most $n - i$ larger elements. Hence, each element in a reduced problem could be the i-th smallest. There is a unique reduced problem corresponding to (P, i) which is obtained by removing all elements that cannot be the i-th smallest and if necessary, adjusting i. We denote the reduced problem corresponding to (P, i) by $\mathrm{red}(P, i)$.

2.3 Selection Algorithms.

A selection algorithm is a binary decision tree. Each node is labeled with a selection problem. The root node is labeled with (E_n, i). The leaf nodes are labeled with *solved* problems (P, i) that have a unique element $a \in \Omega_P$, such that $|D_P(a)| = i$ and $|U_P(a)| = n - i + 1$. Thus, a is the i-th smallest element.

The selection algorithm associates each inner node (P, i) with a comparison $\{a, b\}$, meaning that the algorithm compares a with b as its next step. The two children, $(P{+}ab, i)$ and $(P{+}ba, i)$, correspond to the two possible outcomes of the comparison $a < b$ and $a > b$. The number of comparisons required by the algorithm (in the worst case) is the maximum length of a path from the root to any leaf.

2.4 Minimum Number of Comparisons.

Let $V_i(n)$ denote the minimum number of comparisons required to select the i-th smallest out of n elements in the worst case. We prove the following transfer lemma for lower bounds, showing that if k is a lower bound for selecting the i-th smallest from n elements, then selecting the i-th smallest from $n + 1$ elements requires at least $k + 1$ comparisons.

▶ **Lemma 1.** $V_i(n + 1) \geq V_i(n) + 1$.

Proof. Let $k = V_i(n + 1)$. There exists an algorithm that selects the i-th smallest from $n + 1$ elements using k comparisons. We now construct an algorithm that selects the i-th smallest from n elements using at most $k - 1$ comparisons. Let a and b be the two elements compared first by the algorithm for $n + 1$ elements. Replace a with a new element ω that is larger than any other element in the input of the algorithm. The algorithm still returns the i-th smallest of the remaining n elements. Any comparison involving $a = \omega$ can be skipped, as ω is always larger. In particular, the first comparison is skipped, reducing the number of comparisons by at least 1. Thus, we obtain an algorithm for selecting the i-th smallest from n elements using $k - 1$ comparisons. ◀

▶ Remark 2. It appears that Oksanen was unaware of Lemma 1, as the ranges provided in his table could be improved using this Lemma [9].

Note that the bound in Lemma 1 is tight for some instances, as can be seen from $V_1(n) = n - 1$. An easy corollary to the lemma is $V_{i+1}(n+1) \geq V_i(n) + 1$, which follows immediately from the next lemma.

Let $V_i(P)$ denote the minimum number of comparisons required to select the i-th smallest element of the poset P in the worst-case. $V_i(n)$ is the special case $V_i(E_n)$. We observe that the cost of selection does not change when considering the reduced problem.

▶ **Observation 3.** *Let* $(P', i') = \mathrm{red}(P, i)$. *Then* $V_i(P) = V_{i'}(P')$.

We prove the following lemma showing that the cost of selection remains unchanged when considering the dual problem.

▶ **Lemma 4.** $V_i(P) = V_{n-i+1}(P^\delta)$

Proof. Given $V_i(P)$, we know there exists an algorithm that determines the i-th smallest element in P using exactly that many comparisons. By viewing this algorithm as a binary decision tree and swapping all the children, we obtain an algorithm for selecting the i-th largest element in P^δ, which is also the $(n - i + 1)$-th smallest element of P^δ. ◀

3 Methods and Tools

In this section, we describe our two main approaches to determining $V_i(n)$: the forward search and the backward search. The forward search follows the approach used by Oksanen [10]. We enhance it by using a pruning technique based on compatible solutions. The backward search is a novel approach that has not been previously applied to the problem of selection. It allows us to further improve the computation of optimal selection algorithms.

3.1 Data Structures and Isomorphism Testing.

The key to reducing the search space is to consider only reduced problems and detect isomorphic problems. Isomorphism testing is performed by computing a *canonical* representative. For the backward search, we use a custom algorithm to compute a canonical representative. For complicated cases, our custom algorithm uses `nauty` [8] as a fallback. For the forward search, we use a best-effort approximation to reduce the cost of computing the representative, at the expense of a slightly larger search space.

We store posets as adjacency matrix. We choose a canonical representative in each isomorphism class such that we have a lower triangular matrix that can be stored using $\frac{n^2-n}{2}$ bits.

Forward Search The approximated canonical representative is computed as follows. First, a hash value is computed for each vertex in the poset. We begin with the in- and out-degree for each vertex and assign a hash value based on these degrees. Then, we iteratively update the hash value of each vertex, considering the hash value of the adjacent vertices. Only three iterations are sufficient to obtain good-enough hash values. Then, the vertices are ordered according to their hash values. Our tests showed that the performance gained by computing only an approximated normal form, where some isomorphic problems may have different representatives, outweighs the cost induced by the larger search space.

Table 1 Percentage of normal form computations using `nauty` for variable n and i, where lower values are preferable.

n \ i	1	2	3	4	5	6	7	8
13	0	30.205	6.808	1.526	0.467	0.185	0.114	
14	0	33.667	7.552	1.651	0.425	0.151	0.073	
15	0	36.390	8.184	1.678	0.459	0.132	0.065	0.041
16	0	39.407	8.805	1.796	0.467	0.144	-	-

Backward Search The backward search requires a unique normal form. To that end, we extend the approximation method applied in the forward search in the following ways, which we will explain in detail below. By Lemma 4, we do not need to distinguish a problem from its dual, thus we select one deterministically. We apply some special consideration for elements with the same hash value. Should the algorithm fail to produce a unique representation, then it will fall back to calling `nauty`.

Whether or not to use the dual problem is decided such that $i < \frac{n}{2}$. A potential issue arises if $i = \frac{n}{2}$. In this case, it is impossible to decide whether (P, i) or $(P, i)^\delta$ corresponds to the normal form based solely on the value of i. To resolve this, the subsequent steps will be computed for the problem and its dual. At the end, one is deterministically selected by comparing the binary representations of the posets.

The next step is computing a hash value for each vertex in the poset, using the same algorithm as for the approximated normal form in the forward search, and ordering the vertices accordingly. If there are vertices with the same hash, then we apply a special treatment to the following common situation. Let there be l non-overlapping pairs of vertices with identical hash values (and all other pairs have different hash values). Then, there are 2^l possible permutations that could correspond to the normal form, since each of the l pairs may or may not be swapped. Given the realistic assumption that l is small, all 2^l permutations can be efficiently iterated. With the aim of obtaining values for $n = 16$, it follows that there are at most $\frac{n}{2}$ pairs, hence $l \leq 8$ always holds. All 2^l posets are then calculated, and one permutation is deterministically selected based on its binary representation. It should be noted that this special treatment only applies to pairs of two elements and no longer applies if there are three vertices with the same hash value. All remaining ambiguous cases, not covered by the special treatment above, are handled using `nauty`.

In practise, our algorithm for computing the normal form rarely falls back to using `nauty`, as illustrated in Table 1. It is particularly noteworthy that as i increases, the percentage of `nauty` calls decreases. For small values of i, the high percentage of `nauty` calls is not critical, as computations for small i are generally quick.

3.2 Forward Search.

The forward search algorithm is based on the work of Oksanen [10] and Gasarch et. al. [5]. We first describe the basic algorithm and then discuss the optimizations and pruning techniques we applied. Our main contribution to the forward search is the introduction of a novel pruning criterion based on compatible solutions.

The forward search starts with the problem (E_n, i) and recursively determines the cost of selecting the i-th smallest element of a poset P. Between the two possible outcomes of a comparison, we assume the worst. However, since the algorithm is free to choose which elements to compare, we seek the comparison with the lowest cost in the worst-case outcome.

Thus, the cost $V_i(P)$ can be expressed as follows:

$$V_i(P) = 1 + \min_{a,b \in \Omega_P} \max\{\, V_i(P{+}ab),\, V_i(P{+}ba) \,\}\ . \tag{1}$$

The algorithms generated by the search program are built by saving, for each problem, the comparison that led to the cheapest result.

To save memory and allow further pruning, we traverse the search tree using a depth-first search approach. This reduces the maximum number of comparisons assigned to child problems to one less than the best result currently found. This principle is implemented using a minimax algorithm.

3.2.1 Compatible Solutions.

Suppose we have a solved problem (P, i) with a unique i-th smallest element e. We then know precisely the set of elements that are smaller than e as well as the set of elements that are larger than e. This observation leads us to the notion of compatible solutions, which is such a partition compatible with the current relation.

▶ **Definition 5.** *The solved problem (S, i) is a **compatible solution** of the problem (R, i) if $a \leq_S b \implies b \not\leq_R a$ and S has no relations other than the $n-1$ relations involving the i-th smallest element along with their transitive relations.*

Clearly, a solved problem has exactly one compatible solution. Let

$$\mathcal{C}(P, i) = \{(S, i) \mid (S, i) \text{ is compatible with } (P, i)\}$$

be the set of all solutions compatible with (P, i). Observe that, given two elements a, b that are unrelated in P, every solution compatible with (P, i) is compatible with at least one of $(P{+}ab, i)$ and $(P{+}ba, i)$ and thus

$$\mathcal{C}(P, i) = \mathcal{C}(P{+}ab, i) \cup \mathcal{C}(P{+}ba, i)\,. \tag{2}$$

We use the concept of compatible solutions to derive the following lower bound on the number of comparisons required to select the i-th smallest element of a poset P, which we use as a pruning criterion in the forward search.

▶ **Theorem 6.** *Selecting the i-th smallest element of a poset P requires at least $\lceil \log(|\mathcal{C}(P, i)|) \rceil$ comparisons in the worst case.*

Proof. Assume we have an optimal algorithm for selecting the i-th smallest element of a poset P. From Equation (2), it follows that for every $(S, i) \in \mathcal{C}(P, i)$, there is at least one leaf in the decision tree labeled with $\{(S, i)\}$. Hence, there are at least $|\mathcal{C}(P, i)|$ leaves, implying that the height of the tree is at least $\lceil \log(|\mathcal{C}(P, i)|) \rceil$. ◀

3.2.2 Optimizations.

Caching. We can significantly speed up the exploration by caching previous results, even with a simple usage-based ejection policy. Since the search always imposes an upper bound on the number of comparisons, this also includes unsolved problems, for which we record the currently known minimum.

Maximum Depth. We use the minimax search algorithm to cut off unpromising branches. While searching the possible comparisons of a poset, we keep track of the current best result. The remaining comparisons are searched with a limited depth, ensuring that only solutions improving the current best result are found. At the start of a search, possible comparisons are sorted using a heuristic so that the most promising comparisons are searched for first.

Free Comparison. Another pruning criterion, which has already been used in [9], aims to reduce the size of the searched subtree by adding a 'useful' comparison to eliminate elements faster. Explicitly, it searches for unordered elements a and b such that a has at least two elements less than it and no elements greater than it and b has at least two elements greater than it and no elements smaller than it, and adds $a < b$ to the poset. The new problem is then searched for a solution using the forward search described above without reducing the number of allowed comparisons. If the new problem is not solvable, then the original problem cannot be solvable either. This is valid because adding a comparison 'for free' does not make the problem harder to solve. Connecting a small element to a large element can result in many elements being larger or smaller than i elements. This allows for more elements to be discarded when reducing the poset.

3.3 Backward Search.

The backward search is a different kind of search algorithm, which has not been applied to the selection problem before. The idea is to start with the set of solved problems, and iteratively remove comparisons until the unordered poset is found. Recently, this approach has been applied with great success to the related problem of determining the exact lower bound for sorting [13].

3.3.1 Overview.

We briefly describe the core principles of the backward search algorithm. The input parameters for the backward search are denoted by n and i, similar to the forward search. A first challenge when starting the search with the set of solved selection problems is that, even if we restrict ourselves to a fixed cardinality n and rank i, the set of solved problems remains large. We overcome this by only enumerating reduced problems. With this restriction, the starting point of the backward search is $(E_1, 1)$. By A_k we denote the set of all reduced selection problems solvable using k comparisons. For all n and i, we have $A_0 = \{(E_1, 1)\}$. The backward search begins with A_0 and iteratively computes, for each problem in A_k, the corresponding predecessors, which form the set A_{k+1}. If $(E_n, i) \in A_\ell$, then $V_i(n) = \ell$. Figure 1 illustrates the backward search for $n = 4$ and $i = 2$.

3.3.2 Predecessor calculation.

We begin with a formal definition of a predecessor.

▶ **Definition 7** (Predecessor). *The problem* (Q, j) *is a* predecessor *of* (P, i) *if there is a comparison* (a, b) *such that:*

1. $(P, i) = \mathrm{red}(Q{+}ab, j)$, *and*
2. $V_j(Q{+}ba) \leq V_i(P)$.

Note that the first condition implies $V_i(P) = V_j(Q{+}ab)$.

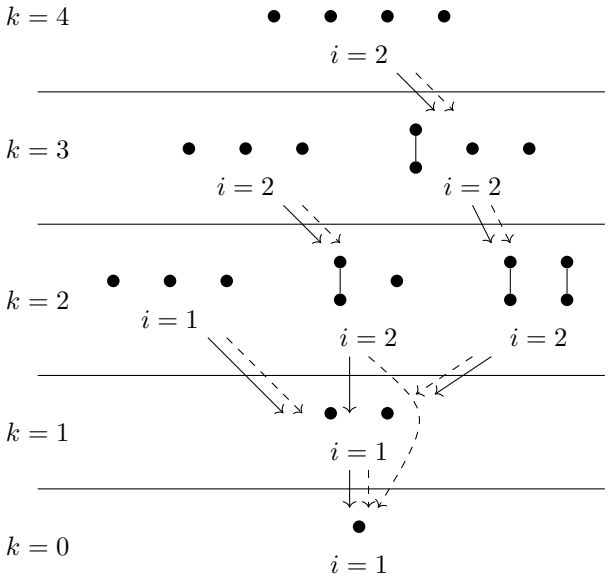


Figure 1 Search tree for $n = 4$ and $i = 2$. Level k contains all posets that can be solved in k comparisons and contribute to the solution for the given parameters n and i. Solid arrows indicate predecessors, while dashed arrows represent the resulting problem when the reversed comparison is inserted.

Any problem (Q, j) satisfying the first condition of the above definition is called a *potential predecessor*. In fact, the first step in enumerating the predecessors is to enumerate the potential predecessors. The second step is to check the second condition.

▶ **Lemma 8.** *Let (P, i) and (Q, j) be reduced problems, where (Q, j) is a predecessor of (P, i). Then, $V_j(Q) \leq V_i(P) + 1$.*

Proof. Since (Q, j) is a predecessor of (P, i), (a, b) exists with $\mathrm{red}(Q{+}ab, j) = (P, i)$ and $V_j(Q{+}ba) \leq V_i(P)$. Therefore, $V_j(Q) \leq \max\{V_j(Q{+}ab), V_j(Q{+}ba)\} + 1 = V_i(P) + 1$. ◀

Storing only reduced problems presents a significant challenge to predecessor enumeration. To illustrate this, consider a problem (P, i) and its predecessor (Q, j). We know there exists a comparison (a, b) such that $(P, i) = \mathrm{red}(Q{+}ab, j)$. However, it is possible that the edge (a, b) is not present in P because either a or b could have been removed during the reduction process. The question arises: How can we undo a comparison that is not visible? Furthermore, even if a and b are not removed during the reduction, there may be other elements that are removed. The challenge is to determine how many elements are removed and what their relationships are.

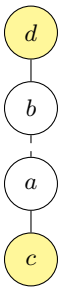
To address these challenges, we will prove two lemmas. The first lemma shows that after adding a comparison (a, b), at most one of $\{a, b\}$ will be removed by the reduction.

▶ **Lemma 9.** *If (P, i) is a reduced problem and $(Q, j) = \mathrm{red}(P{+}ab, i)$, then, $\Omega_Q \cap \{a, b\} \neq \emptyset$.*

Proof. Since (P, i) is reduced, we have $|D_P(c)| \leq i$ and $|U_P(c)| \leq n - i + 1$, where $n = |\Omega_P|$, for every $c \in P$. In particular, this holds for both a and b. Let $P' = P{+}ab$ and assume $Q \cap \{a, b\} = \emptyset$. Observe that $D_{P'}(a) = D_P(a) \leq i$ and $U_{P'}(b) = U_P(b) \leq n - i + 1$. Thus, for a and b to be removed, we must have $U_{P'}(a) \geq n - i + 2$ and $D_{P'}(b) \geq i + 1$. Note that there are no elements between a and b, as they are incomparable in P and there is a

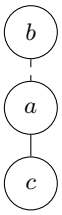
Initial Problem (P, i)

Here $(P, 2)$ *is reduced.*
$R = \{c, d\}$, $C^- = \{a, d\}$.



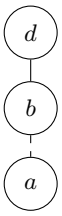
Option 1: Remove d

Result: (P', i') *where*
$P' = P|_{\Omega_P \setminus \{d\}}$, $i' = 2$



Option 2: Remove c

Result: (P'', i'') *where*
$P'' = P|_{\Omega_P \setminus \{c\}}$, $i'' = 1$



The problem (Q, j) is obtained from (P, i) by adding $a < b$ (the dashed edge) and reducing. We have $(Q, j) = \text{red}(P{+}ab, i) = \text{red}(P'{+}ab, i') = \text{red}(P''{+}ab, i'')$ and $R = \Omega_P \setminus (\Omega_Q \cup \{a, b\}) = \{c, d\}$. The problem (P', i') is reduced, but (P'', i'') is not.

Figure 2 Example for Lemma 10.

Hasse arc between them in P'. Hence, $U_{P'}(a) \cap D_{P'}(b) = \{a, b\}$, leading to the contradiction $n = |\Omega_{P'}| \geq |U_{P'}(a) \cup D_{P'}(b)| = |U_{P'}(a)| + |D_{P'}(b)| - |U_{P'}(a) \cap D_{P'}(b)| \geq n + 1$. ◀

The next lemma shows that the elements removed by the reduction, which are not a or b, can be added one after the other in such a way that all the intermediate problems are reduced. Figure 2 exemplifies that this is not trivial – a specific order is necessary for the intermediate problem to be reduced.

▶ **Lemma 10.** *Let* (P, i) *be a reduced problem, and let* $(Q, j) = \text{red}(P{+}ab, i)$. *If* (P, i) *is a predecessor of* (Q, j) *and* $\Omega_P \setminus (\Omega_Q \cup \{a, b\})$ *is not empty, then there exists an element* $c \in \Omega_P \setminus (\Omega_Q \cup \{a, b\})$ *such that* $(P|_{\Omega_P \setminus \{c\}}, i')$, *where* $i' = i - 1$ *if* $|U_{P+ab}(c)| \geq n - i + 2$ *and* $i' = i$ *otherwise, is a reduced predecessor of* (Q, j).

Proof. Let $R = \Omega_P \setminus (\Omega_Q \cup \{a, b\})$ be the set of elements, other than a or b, removed by the reduction. It is easy to see that for every $c \in R$, the problem (P', i') where $P' = P|_{\Omega_P \setminus \{c\}}$ and $i' = i - 1$ if $|U_{P+ab}(c)| \geq n - i + 2$ and $i' = i$ otherwise is a predecessor of (Q, j):

- It is obvious that $(Q, j) = \text{red}(P'{+}ab, i')$.
- The problem $(P'{+}ba, i')$ cannot be harder than $(P{+}ba, i)$, hence $V_{i'}(P'{+}ba) \leq V_i(P{+}ba)$.

The challenge is to find an element c such that P' is reduced. We define the following sets:

$$C^+ = \{e \in \Omega_P \mid |U_P(e)| = n - i + 1\}$$
$$C^- = \{e \in \Omega_P \mid |D_P(e)| = i\}$$
$$R^+ = \{c \in R \mid |U_{P+ab}(c)| \geq n - i + 2\}$$
$$R^- = \{c \in R \mid |D_{P+ab}(c)| \geq i + 1\}$$

Note that $R = R^- \cup R^+$. The elements in C^- and C^+ are critical: If P' is not reduced, it is because one of these elements can no longer be the i'-th smallest. To avoid this, we need an element $c \in R^+$ that is smaller (in P) than all elements in C^-. By symmetry, any element $c \in R^-$ larger than all elements in C^+ works as well. We first show that if we have an element in $R^- \cap C^-$ or $R^+ \cap C^+$, then this is the case as

$$\forall c \in C^+, e \in C^- \colon c \leq_P e\,. \tag{3}$$

Assume we have $c \in C^+$ and $e \in C^-$, but $c \not\leq_P e$. Then the sets $U_P(c)$ and $D_P(e)$ are disjoint, leading to the contradiction $|U_P(c)| + |D_P(e)| = n + 1 > |\Omega_P|$.

The second step is to show that if $R^- \cap C^- = \emptyset$ and $R^+ \cap C^+ = \emptyset$, we can pick any $c \in R$. We show:

$$\forall c \in R^-, e \in C^+ \colon e \leq_P c \text{ or } e \in R^+ \,. \tag{4}$$

Assume we have $c \in R^-$ and $e \in C^+$. By another counting argument we observe that the sets $D_{P+ab}(c)$ and $U_P(e)$ cannot be disjoint: Assuming $D_{P+ab}(c) \cap U_P(e) = \emptyset$ leads to the contradiction $|D_{P+ab}(c)| + |U_P(e)| \geq n + 2$. Thus, $(e, c) \in P{+}ab$. Hence, we have either $e \leq_P c$ or $e \leq_P a$. If $e \leq_P a$ but $e \not\leq_P c$, we have to show $a \neq e$ to conclude $e \in R^+$. Assume $a = e$. Then $U_P(a) \setminus \{a\}$ and $D_{P+ab}(c)$ must be disjoint, as $e \not\leq_P c$. We obtain the contradiction $|U_P(a) \setminus \{a\}| + |D_{P+ab}(c)| \geq n + 1$. By symmetry, we also obtain:

$$\forall c \in R^+, e \in C^- \colon c \leq_P e \text{ or } e \in R^- \,, \tag{5}$$

which concludes the proof. ◀

The set of predecessors for a given problem (P, i) can be split into the following three subsets, which we compute in that order.

1. Predecessors on the same set of elements.
2. Predecessors with exactly one additional element that is involved in the comparison.
3. Predecessors with additional elements of which at least one is not part of the comparison.

Of the predecessors to the problems in A_k, those that actually require $k + 1$ comparisons (those not in A_k) make up the set A_{k+1}. We describe the steps to compute the three sets below.

Predecessors on the same set of elements. Our goal is to find all posets with n elements that result in poset P after inserting a comparison $a < b$. Each edge in the Hasse diagram of P potentially represents a comparison by which (P, i) can be obtained from a predecessor. A challenge arises from transitive relations, as the insertion of a single comparison can lead to the insertion of multiple transitive relations. This is illustrated in Figure 3. Removing a comparison from (1) can result in either (2) or (3). Therefore, both (2) and (3) are potential predecessors, even though the same comparison is removed each time.

After enumerating the potential predecessors, we check for each one whether it is a predecessor of (P, i). For a potential predecessor (Q, j) where $(P, i) = \mathrm{red}(Q{+}ab, j)$, we check whether $(Q{+}ba, j)$ can be solved using at most $V_i(P)$ comparisons. This is done by checking whether $\mathrm{red}(Q{+}ba, j)$ is contained in one of the sets A_k for $k \leq V_i(P)$, which have already been computed.

Predecessors with exactly one additional element that is involved in the comparison. We construct the potential predecessor for this case as follows. We add a new element to the poset P and enumerate all possibilities for the relation between the new element and

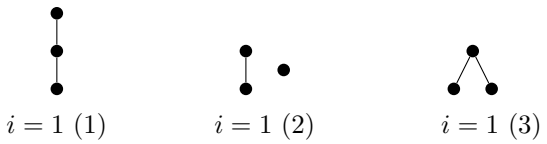


Figure 3 Case where further comparisons can be removed transitively by removing a comparison.

the existing elements. We want the predecessor to be reduced, thus it is crucial to ensure that the new element cannot be immediately reduced and that no existing elements can be reduced either. Since the new element is either smaller or larger than the i-th smallest element being searched for, either the $(i+1)$-th smallest or the i-th smallest element is searched for in these predecessors. Furthermore, for each (Q, j) obtained this way, there must exist elements a and b such that $\mathrm{red}(Q{+}ab, j) = (P, i)$. This is required for (Q, j) to be a potential predecessor, and since we additionally want the new element to be part of the comparison, we mandate that either a or b is the new element. Checking whether the potential predecessors constructed this way are predecessors is done in the same way as in the preceding step.

For the correctness of our approach, note that if we have an arbitrary reduced predecessor, then by removing all elements that are not present in P, with the exception of a and b, we obtain another reduced predecessor. We get this by induction on the number of elements removed using Lemma 10. If both $a, b \in \Omega_P$ then this predecessor is enumerated in the first step. Otherwise, as proven in Lemma 9, if one of a or b is in Ω_P, the predecessor is enumerated in this second step. In the next step, we will iteratively enumerate all predecessors by adding additional elements to the ones already found. This way, we will discover the arbitrary predecessor we started with.

Predecessors with additional elements of which at least one is not part of the comparison. Starting with the predecessors computed in the first two steps, new elements are iteratively inserted. We alternate between generating new potential predecessors and checking which of those are actually predecessors. We stop when no new predecessors are found or when we reach an upper limit on the number of elements. New potential predecessors are generated by adding a new element to each predecessor and enumerating all possible relations with the existing elements. When inserting a new element, it is important to note that it may no longer be the i-th smallest but rather the $(i+1)$-th smallest element being searched, similar to the second step. We only consider potential predecessors that are reduced and of cause by the addition of the comparison $a < b$ and subsequent reduction, the resulting problem (P, i) should be obtained anew.

3.3.3 Optimizations.

Limit search space. Each time an element is removed from a poset during reduction of a problem, a possible consequence is, that now we are looking for the $(i-1)$-th instead of the i-th element. Therefore, if (P, j) is a problem appearing in an algorithm selecting the i-th smallest of n elements, then $i - j \leq n - |\Omega_P|$. An optimization we apply to the backward search is to ignore any problems that violate this inequality. They cannot contribute to the solution. E.g. no problem with $n = 6$ and $i = 2$ can lead to a problem with $n = 7$, $i = 4$.

Iterative deepening. As the theoretical upper bounds are too high in practice, the program uses an iterative deepening approach. It starts with an upper bound that corresponds to the theoretical lower bound, derived by Lemma 1 from the smaller values for n and increments this bound until a solution is finally found. As it is not possible to save which posets are ignored due to the guessed upper bound without considerable effort, the backward search is restarted several times. Although results from previous rounds are not used, the search space can be considerably reduced, making the program more efficient.

Table 2 Efficiency of parallelism for $n = 13, i = 7$

cores	1	2	3	6	12	24
time (m)	165	84	64	32	17	9
efficiency	1.00	0.99	0.90	0.87	0.81	0.75

Table 3 Minimum number of comparisons needed to select the i-th smallest of n elements. Values resulting from our work are printed in bold.

n	i							
	1	2	3	4	5	6	7	8
1	0							
2	1							
3	2	3						
4	3	4						
5	4	6	6					
6	5	7	8					
7	6	8	10	10				
8	7	9	11	12				
9	8	11	12	14	14			
10	9	12	14	15	16			
11	10	13	15	17	18	18		
12	11	14	17	18	19	20		
13	12	15	18	20	21	22	23	
14	13	16	19	21	23	24	**25**	
15	14	17	20	23	**24**	**26**	**26**	**27**
16	15	18	**21**	**24**	**26**	**27**	**28** - 33	**28** - 36

Remaining comparisons. This number of edges in the Hasse diagram of a poset is a lower bound to the number of comparisons that must be removed until the unordered poset is reached. Since at most one comparison can be removed in each step, all posets containing too many Hasse arcs can be discarded, as they cannot lead to an unordered poset with the remaining comparisons.

Parallelization. The backward search can be ideally parallelized by performing the calculation of the predecessors in parallel. The only two bottlenecks are read access to the cache and the efficient merging of all partial results. As shown in Table 2 for $n = 13$ and $i = 7$, it can be seen that the backward search scales well with the number of cores. To set the different times in relation to the number of cores, the efficiency was determined, which represents a direct correlation between the two variables. This can be calculated as follows

$$\text{efficiency} = \frac{\text{single-core time}}{\text{number of cores} \cdot \text{multi-core time}}.$$

The higher the efficiency, the better the time scales with the number of cores.

4 Results

Running our computer search, we obtained the values $V_i(n)$ shown in Table 3. Our findings confirm most of the values computed by Oksanen [9]. Notably, $V_5(12) = 19$ contradicts a

Table 4 Execution times and number of posets stored in the cache for different search methods.

		Forward Search		Backward Search		Oksanen
n	i	time	posets	time	posets	time
12	6	1m 30s	$1.9 \cdot 10^6$	18.0s	$996 \cdot 10^3$	1.9s
13	7	59m 20s	$67.6 \cdot 10^6$	7m 16s	$14.5 \cdot 10^6$	16h 10m
14	7	14h 40m	$925.3 \cdot 10^6$	2h 17m	$263.3 \cdot 10^6$	>5d[a]
15	8	14d 21h 26m	$15.7 \cdot 10^9$	1d 3h 7m	$2.2 \cdot 10^9$	>5d[a]
16	6	6d 11h 21m	$3.6 \cdot 10^9$	1d 1h 26m	$2.6 \cdot 10^9$	-

[a] We aborted Oksanen's program after 5 days.

conjecture by Gasarch [5] that the optimum can be achieved using a "pair-forming algorithm", where the first comparison of any singleton is with another singleton (in this case, the best pair-forming algorithm requires 20 comparisons). The values printed in bold were unknown previously. For $V_7(14)$, $V_6(15)$, $V_7(15)$, and $V_8(15)$, only a range was known prior. Oksanen incorrectly lists $V_5(15)$ as 25 on his website [9], although his search algorithm does produce the correct value of 24. The values for $n = 16$ have not been computed before; we provide all values for $i \leq 6$ and ranges for $V_7(16)$ and $V_8(16)$. The upper bound for the ranges is $V_i(n) \leq n - i + (i-1)\lceil \log(n+2-i) \rceil$ [6].

To validate the upper bounds of the values we calculated, we checked the algorithm certifying each number on each of the $n!$ permutations. For the lower bound we computed each number twice using two different algorithms, the forward and the backward search. This reduces the likelihood of an incorrect result due to a coding error.

Table 4 compares the execution times of the different algorithms to find optimal selection algorithms. All experiments were conducted on a machine with two Intel Xeon CPUs, each equipped with 12 cores (24 threads), and a total of 768 GB of RAM. The forward search was started with 500 GB of RAM and restarted for each combination of n and i, ensuring no use of cached data from previous runs to provide comparability. The 'Oksanen' column presents execution times of Oksanen's program [9] on our hardware, started with a cache size of 25 GB RAM. It was originally designed to use 400 MB RAM and 25 GB is close to a natural limit due to the use of 32-bit indices. Additionally, we measured the number of posets stored in the cache after the calculation, which can be found in Table 4.

To evaluate the potential of compatible solutions as pruning criterion, we determined the maximum number of compatible solutions encountered for a given cost for $n \leq 14$ and used that as a boundary in a subsequent run. For $n = 14, i = 7$, this resulted in a time of 1h 21m with $147 \cdot 10^6$ posets in the cache – improvements by factors of 11 and 6.3, respectively.

5 Conclusion and Open Questions

As stated in the motivation before: the road is better than the inn. Along the way we improved the forward search using a pruning criterion based on compatible solutions and evaluated a new algorithmic approach – the backward search coming to a final conclusion that both are valid. Between the structural constraints of our algorithms and the available hardware, $n = 16$ is likely the limit of feasible calculation for the current state. We believe that higher n are realistic with new algorithmic ideas and conclude this work by talking about promising directions for further research. The latest version of our software is available at GitHub (`https://github.com/JGDoerrer/selection_generator`).

Bidirectional Search. The classical meet-in-the-middle approach for a bidirectional search will not work for the selection problem. Our experiments with the forward and backward search showed that, using the number of comparisons as metric, at any possible meeting point, at least one of the two searches has already covered over 99% of its search space.

An alternative approach to the bidirectional search would be to first run the backward search, but restricted to problems with specific properties such as having a large number of compatible solutions, e.g. problems (P, i) in A_k with $\mathcal{C}(P, i) \geq \alpha \cdot 2^k$ for some $\alpha \in [0, 1]$. These are presumably hard to solve, so the subsequent forward search only has to explore problems with a small number of compatible solutions, which should be easier to solve.

There might be better metrics than the compatible solutions, as there is a significant gap between the lower bound and the cost of a problem. We observed that the number of comparisons required to solve a selection problem is typically about twice the lower bound obtained from the number of compatible solutions, minus a constant. This observation is reasonable, as the lower bound for the median is $n + \mathcal{O}(n)$, which is far from the best known asymptotic lower bound $2n + \mathcal{O}(n)$.

Improved Weight Function. A better lower bound might be achieved by assigning a weight to each solution rather than merely counting compatible solutions. Mimicking the techniques used to obtain the $2n + \mathcal{O}(n)$ bound in [1] could be a fruitful approach.

Yao's conjecture. Yao conjectured that finding the i-th smallest of n elements is at least as hard as finding an n-element subset S of m elements, where $m > n$, and an element $s \in S$ such that s is the i-th smallest element in S [14]. If true, it would imply a $2.5n + \mathcal{O}(n)$ algorithm for computing the median [12]. By adapting our search algorithm, one could search for counter examples to the conjecture.

References

1 Samuel W. Bent and John W. John. Finding the Median Requires 2n Comparisons. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, pages 213–216, 1985. doi:10.1145/22145.22169.

2 Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, and Robert E. Tarjan. Time Bounds for Selection. *Journal of Computer and System Sciences*, 7(4):448–461, 1973. doi:10.1016/S0022-0000(73)80033-9.

3 Miguel de Cervantes Saavedra. *Don Quijote*. Madrid, 1605. Originally published as "El ingenioso hidalgo don Quixote de la Mancha".

4 Dorit Dor and Uri Zwick. Selecting the Median. *SIAM Journal on Computing*, 28(5):1722–1758, 1999. doi:10.1137/S0097539795288611.

5 William Gasarch, Wayne Kelly, and William Pugh. Finding the ith largest of n for small i, n. *SIGACT News*, 27(2):88–96, Jul 1996. doi:10.1145/235767.235772.

6 Abdollah Hadian and Milton Sobel. Selecting the t-th Largest Using Binary Errorless Comparisons. Technical report, University of Minnesota, 1969. doi:11299/199105.

7 Donald Knuth. *The Art of Computer Programming, Vol. 3: Sorting and Searching*. Addison-Wesley, 1973.

8 Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, II. *Journal of Symbolic Computation*, 60:94–112, 2014. doi:10.1016/j.jsc.2013.09.003.

9 Kenneth Oksanen. Selecting the 'i'th largest of 'n' elements, 2002. URL: https://www.cs.hut.fi/~cessu/selection/.

10 Kenneth Oksanen. Searching for Selection Algorithms. *Electronic Notes in Discrete Mathematics*, 27:77, 2006. ODSA 2006 - Conference on Optimal Discrete Structures and Algorithms. doi:10.1016/j.endm.2006.08.064.

11 Mike Paterson. Progress in Selection. In *Scandinavian Workshop on Algorithm Theory*, pages 368–379. Springer, 1996. doi:10.1007/3-540-61422-2_146.

12 A Schönhage, M Paterson, and N Pippenger. Finding the Median. *Journal of Computer and System Sciences*, 13(2):184–199, 1976. doi:10.1016/S0022-0000(76)80029-3.

13 Florian Stober and Armin Weiß. Lower Bounds for Sorting 16, 17, and 18 Elements. In *2023 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*, pages 201–213. SIAM, 2023. doi:10.1137/1.9781611977561.ch17.

14 Foong Frances Yao. On Lower Bounds for Selection Problems. Technical Report MIT-LCS-TR-121, Massachusetts Institute of Technology, Cambridge, MA, 1974. doi:1721.1/149426.