Optimal Selection and Sorting via Dynamic Programming

MICHA HOFRI, Worcester Polytechnic Institute

We show how to find optimal algorithms for the selection of one or more order statistics over a small set of numbers, and as an extreme case, complete sorting. The criterion is using the smallest number of comparisons; separate derivations are performed for minimization on the average (over all permutations) or in the worst case. When the computational process establishes the optimal values, it also generates C-language functions that implement policies which achieve those optimal values. The search for the algorithms is driven by a Markov decision process, and the program provides the optimality proof as well.

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Sorting and Searching; F.2.3 [**Analysis of Algorithms and Problem Complexity**]: Tradeoffs between complexity measures; G.3 [**Probability and Statistics**]: Markov Decision Process; I.2.8 [**Artificial Intelligence**]: Dynamic Programming

General Terms: Algorithms, Experimentation

Additional Key Words and Phrases: Graph isomorphism, Markov decision process, optimal selection, optimal sorting, small set

ACM Reference Format:
Hofri, M. 2013. Optimal selection and sorting via dynamic programming. ACM J. Exp. Algor. 18, 2, Article 2.3 (July 2013), 14 pages.
DOI: http://dx.doi.org/10.1145/2444016.2493373

1. INTRODUCTION

How would you find the median of five distinct numbers, using the smallest number of comparisons, when averaged on their 120 possible permutations? Or sort a similar set of eight numbers?

The need for efficient small-scale selection algorithms occurs in numerous situations, and Chern and Hwang [2001] illustrate a classic one. Our specific need arose during work described in Battiato et al. [2000], Cantone and Hofri [2013] on the remedian, a fast, approximate median selection algorithm which is based on recursively finding the exact median of small sets of elements. The performance of the remedian depends on the number of comparisons used during those exact median searches, and the relevant sizes of the sets (n), are small odd integers. Initially $n = 3$ was used, but later work led to the desirability of using higher values, with 5 or 7, possibly 9, being the most promising (7 ended up the winner).

What is the optimal way to find the median of such a small set? Both of the Quicksort and remedian algorithms make the small-bore selection many times, getting all possible permutations of n elements repeatedly, hence "optimal" there means "optimal

Initial work on this research was done when the author enjoyed the hospitality of project ALGO at INRIA in the spring and summer of 2006.
Authors' address: M. Hofri, Department of Computer Science, Worcester Polytechnic Institute, 100 Institute Road, Worcester, MA 01609-2280; email: hofri@wpi.edu.

DOI: http://dx.doi.org/10.1145/2444016.2493373

on the average", requiring the fewest number of comparisons when averaged over all $n!$ permutations. These are the *mean-optimal algorithms*. To our surprise we could find very little information about them in the literature, and had to create them. This article tells the tale.

Standard computing consists of using algorithms well-suited for their specific tasks, over a range of problem sizes. Indeed, the questions that we ask when analyzing algorithms often concern the change of performance with the problem size. Only when we focus on *optimal* algorithms we come to realize that performing some task optimally (in our case, finding the median) may require a different algorithm for each problem size. This is the kind of algorithms we seek here.

The only substantial source we found is Knuth, in his volume on sorting and searching [Knuth 1999, Section 5.3.3]. He reports on considerable activity with the objective of *worst-case* minimization, but very little for the average case; for the latter criterion Knuth provides values for the mean-optimal cost of selecting any order statistic, for n up to 7 elements, but no algorithms are given and no references for proofs. The source of the numbers is "an exhaustive computer search."

Knuth, in the preceding reference, and several other sources provide information on such *worst-case* optimal values for selection; additional sources are Noshita [1974] and Gasarch et al. [1996], and for sorting Peczarski [2002]; none (excepting Noshita [1974], displaying a worst-case optimal algorithm for the median of 9 elements) provides information how to perform selection at the claimed cost. No algorithms for any mean-optimal work were found.

Since our need was to use the functions which achieve these values, the search for such algorithms was the motivation of this work.

It turns out that the same method we used to find the algorithms we needed can be easily modified to support additional objectives, such as looking for more than a single order statistic, or even, in the extreme case, all of them, which implies sorting. In addition, the objective can be very simply changed from average-case minimization to worst-case optimization.

The method we have used can, however, only be employed for such "small sets" as described before. It is not easy to handle sets larger than 9 or 10, and even there some attention is required. It is prey to what Richard Bellman memorably called "the curse of dimensionality" in Bellman [1957, page ix]. The underlying cause is that the approach to follow uses a natural representation of a comparison-based sorting or selection activity, over n values, as adding arcs to a directed graph over n vertices. Bellman was mainly referring to cost of solving problems which increased exponentially, with an exponent that is linear in the number of stages, points, or dimensions. However, in a graph as given earlier, each pair of vertices gives rise to three possible states (unrelated, when the items have not yet been compared, and two possible directions of an arc, representing the outcome of a comparison). Hence the number of distinct directed labeled graphs is $3^{n(n-1)/2}$: the exponent is quadratic in the number of entries. This expression, for $n = 8$, reaches 2.28768×10^{13}. Happily, we did not need to consider such numbers, as we explain shortly, but the required number is still substantial, and grows super-exponentially with the number of elements. This is a serious limitation. Fortunately, it satisfied our initial needs. The reader may wonder why such a limited method was selected—the reason is simple enough: it is the only one we know which comes with a certificate of optimality.

Following Knuth we denote by $\overline{F}(n)$ the mean-minimal cost, measured in number of comparisons, needed to sort n distinct numbers, and by $\overline{V}_k(n)$ the minimum required to find the kth order statistic, $K_{n,k}$, among n such entries, with $1 \le k \le n$, on the average. The same symbols, without the bars, stand for optimal bounds: the minimal costs in the

worst-case scenarios. Because of the initial motivation, most of the discussion is about selection of the median, where $k = \lfloor (n+1)/2 \rfloor$. Much of the treatment of the selection and sorting problems is unified, since they can be viewed as extreme ends of a spectrum of problems of locating more and more order statistics. This common approach leads us to denote the optimal cost to learn a desired fact about the set (its median, or extreme quintiles, or sorted order...) by U; we use $U(S)$ to denote the optimal cost when starting from a state-of-knowledge S about the entries. The state information is encapsulated in a *state graph*, a directed (acyclic) graph over the n entries (the number n does not change, and is normally suppressed in what follows); we found it convenient to denote nodes by the first lower-case letters of the alphabet, $a, b, c, \ldots$. The terms "nodes" (of the graph) and "entries" (or elements of the poset) correspond to two views of the same objects. The arcs in the graph stand for comparisons which have been made, and by convention an arc is from the larger entry to the smaller one.

As successive pairs of elements are chosen for comparison, arcs are added to the state description, and their directions determine the current state of knowledge. The information needed to determine which comparisons can still be made (= which arcs may be added) is the *transitive closure* of all past comparisons. New states thus arise as long as they are not resolved. Such states carry enough information to determine the desired fact about the set. We denote by $\mathcal{S}_r$ the set of *resolved states*.

Note. Many states display the following phenomenon: a comparison is made and one outcome produces a resolved state, while the opposite outcome does not yet provide this happy end.

Hence the process evolves as a first-order Markov chain, allowing us to control it (choose the most informative, or effective comparisons) by solving the underlying Markov decision process. The same approach is also called *stochastic dynamic programming* [Ross 1983]. The transition probabilities of the chain are due to the number of permutations that are compatible with the information contained in the state descriptions; we assume throughout that the input is equally likely to be in any of the $n!$ possible permutations.

Note that there are two types of search operations reported here: one is the search involved in selection or sorting, consisting in comparing entries; let us call it generically selection. The other is the main thrust of the reported work: a search for the optimal policy to perform the selection. Selection can be seen as done on a binary decision tree. The second search type views this tree as embedded in a vastly larger decision tree in which each state can be followed by several, sometimes many, different actions, and the substance of the second search is to delineate the optimal policy among all alternatives. It may remind the reader of tracing out a skeleton in a very rich environment.

Here is the main relation we use, written as a recurrence, expressing the development of the process, one comparison at a time, for states which are not yet resolved

$$U(S) = 1 + \min_{\varepsilon \in C(S)} E[U(S \mid \varepsilon)], \qquad S \notin \mathcal{S}_r, \tag{1}$$

where "1" is our cost unit, one comparison. The action ε is the choice of a pair of items to compare. $C(S)$ consists of all the node pairs which have no arc between them in (the transitive closure of) S, and are then possible choices of action. The symbol $S \mid \varepsilon$ represents any of the two possible outcomes of using the action ε in the state S. The "expectation" is taken over the two possible results of the comparison. The double quotes represent the history of the work: initially we were only interested in the mean-optimal value, and this operation was a true expectation over the probabilities of the two outcomes. Later the same recurrence was used for upper and lower bounds, as discussed shortly, and then the operation needs to be replaced by taking the larger or smaller of the two values $U(S \mid \varepsilon)$. For a resolved state S_r we can write $U(S_r) = 0$.

The recurrence does not show what is the objective we denote by U: that is implicit in the determination whether a state is resolved. Different objectives require the use of different resolution functions.

This recurrence differs from a typical formulation of a Markov decision process in two aspects. One is that the "time" dimension (here, the number of comparisons to follow), normally called the horizon in the literature on Markov decision processes, is not specified. It has the natural, weak upper bound $\binom{n}{2}$. Each possible thread of the search is seen as continuing until the termination condition, a state in $\mathcal{S}_r$, is reached.

The other distinction is that the state space is not explicitly defined. The issue is not its size; large as it is, there is no difficulty in referring to all possible acyclic digraphs on n nodes, but runs deeper, for technical reasons that we explain in the next section.

The recurrence needs, of course, to be completed with suitable initial values, which we discuss soon, but for the moment, the reader may think of a start with S_0, a "blank state," a graph with no arcs. The recurrence has now the form often called "Bellman Equation" or his "Optimality Equation". It satisfies the criteria needed for Theorems 1.1 and 1.2 [Ross 1983, page 50] to hold, which provides us with the following.

THEOREM. *The solution of recurrence* (1) *is the optimal cost of reaching a resolved state from* S_0*; any sequence of possible actions which satisfies the solution is an optimal policy (algorithm).*

Note. Consistently with our main interest, in deriving optimal results (and algorithms) that minimize the expected number of comparisons, we retained the expectation operator in the relation (1), and most of the discussion to follow reflects this preference. We discuss later replacing it with two alternatives. The operator MIN selects the smaller of the two values, corresponding to the two results of the comparison; we then get the rarely interesting result of the minimum possible number of comparisons needed, under the most obliging circumstances, naturally always $n-1$ (for any of the objectives of selection or sorting). Using the corresponding MAX operator provides the value of the *worst-case bound* $V_k(n)$ (or $F(n)$); both cases also generate the algorithms that realize them. If so desired, the calculations can be instrumented to provide the probability of the set of input which leads to these extreme cases (assuming all permutations are equally likely).

2. THE COMPUTATIONAL PROCESS

We describe the metaprogram we prepared to perform the search outlined earlier. We list the ingredients of the process, show a minimal example, and then discuss the general flow of the computations.

As is evident from Eq. (1) the computation has a two-step character: in a given state, create a list of the possible actions, and the states they can lead to (possibly compute the transition probabilities, though this can be deferred); in the next step the minimizing action is selected. This requires in turn evaluating $U(\ )$ on all possible progeny. In practice we use a different approach, which is also a two-phase calculation, but proceeds "globally": In the first we build a network of all useful states, using a Breadth-First-Search-like (BFS) process and only then use a Depth-First-Search-like (DFS) process to do the actual evaluation, according to the operator used in the recurrence—E, MIN, or MAX, and select the optimal policy.

At this point it behooves us to observe that nearly all of the details that follow—not all, but most—are due to the sad fact that we solve our problems with computers that are not lightning fast, and come with storage that is not unboundedly vast.

The network is not a tree, as one expects such a procedure to induce, since we find it computationally essential to classify all states into equivalence classes, and retain only one state representative of each class. Equivalent states carry the same state

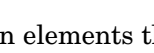
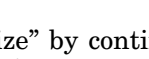
information (in the form of a permuted transitive closure of the comparisons that led to each), and they have the same cost of completion, the value $U(S)$. However, equivalent states may be reached via different comparisons sequences, and their roles in the optimal policy will vary, unlike the unique paths from a tree root to its nodes. A useful view of the network sees it as consisting of alternating layers of states and actions, starting at the top from the initial state. Also, two types of edges exist: one type connects the layers: a state to its possible action blocks and from each such block to the two possible resulting states; a second type connects a state to its equivalence class (more precisely, its representative, which is simply the first state of that class that was created.) An example is given when graph isomorphism is discussed shortly.

While the network may be viewed simply as a DAG, the second type of arcs mentioned makes the second phase of the derivation, which handles the calculation of the function $U(\,)$ in each state, and selects the optimal action, only DFS-like; the need to monitor the genealogy of the states during this process was the main computational challenge and our solution is outlined later.

Usually a dynamic program is solved by a bottom-up calculation; starting from end states. This is infeasible here, due to our keeping the state space unspecified as long as possible. Instead, this is the place for a top-to-bottom development, and only when we have created the entire network, of all relevant states' representatives, the optimization can be carried through.

Note. This mode of calculation is also called "backward induction".

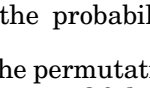
Transition probabilities. In a given state, when an action calls for making a comparison between two entries, say c and d, the probability $\Pr[c>d]$ depends on the current information state S, and equals, naturally, the fraction of the $n!$ permutations which are consistent with S, that are also consistent with $[c>d]$; in our calculations we determined it by explicitly counting such permutations.

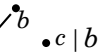
Minimal example. Here is the way to proceed in the minimal case, that of $n = 3$ entries, where the notions of selection for $k = 2$ (the median) and sorting coincide. The states are specified by exhibiting the known relations. We use the convention that all arcs are right-to-left, hence among elements which are connected, those on the right are larger than elements that are on the left. The initial state is simply  , except that since the first comparison can be taken arbitrarily we take it between a and b, and "symmetrize" by continuing with the assumption that $a < b$, that is  . The recurrence is now

$$U\left(\begin{array}{c}\bullet\,b\\ a\bullet\quad\bullet c\end{array}\right) = 1 + U\left(\begin{array}{c}\bullet\,b\\ a\bullet\quad\bullet c\end{array}\right) = 2 + \min_{\varepsilon\in(a:c,\ b:c)} E_\varepsilon U\left(\begin{array}{c}\bullet\,b\\ a\bullet\quad\bullet c\end{array}\,\middle|\,\varepsilon\right).$$

Each of the two possible choices on the right needs to be evaluated.

$$E_{a:c}U\left(\begin{array}{c}\bullet\,b\\ a\bullet\quad\bullet c\end{array}\,\middle|\,a:c\right) = \Pr[a>c]U\left(\begin{array}{c}\bullet\,b\\ a\bullet\\ \bullet c\end{array}\right) + \Pr[a<c]U\left(\begin{array}{c}\bullet\,b\\ a\bullet\!\!-\!\!\bullet c\end{array}\right)$$

Consider the probabilities. Assume the values of a, b, c are some permutation of $(1, 2, 3)$. The permutations of $(1, 2, 3)$ which are viable in the state  are 1,2,3; 1,3,2; and 2,3,1. Of these only the last survives when we require $a > c$, hence the two probabilities we obtain are 1/3 and 2/3, from left to right. Then observe that the first state on the right-hand side is resolved (whether we are looking for a complete sort

or only the median is of interest). The cost of this state is zero. The other state is not resolved yet, but the only possible additional comparison will resolve it, at a cost of 1, hence the value of the right-hand side given before is 2/3. When we turn to evaluate $E_{b:c}U($  $| b : c)$ we see it is entirely symmetric with the preceding, has the same value, and hence the choice is moot; we have found that $U($  $) = 8/3$.

This example is "too minimal" to exhibit the process of extracting the optimal policy here.

Resolution. This is the computational step where a state is tested for having sufficient information to determine the final outcome. Such a state is deemed *resolved*. It needs no further comparisons. Our method does not create a state node then, and all the information is in the action block that led to it.

Resolution criteria. To find the median of the set we mark and count the entries which are larger than $\lfloor n/2 \rfloor$ elements and those smaller than $\lceil n/2 \rceil$ elements. This is part of the state descriptor. (Similarly if looking for $K_{n,k}$ we mark and count those known to be larger than k elements, and those smaller than $n-k+1$.) Such entries will not be selected for additional comparisons. Once their number reaches $n-1$, the state is deemed resolved (the single unmarked entry is the sought element). More complex computing objectives, such as finding multiple order statistics, are possible, requiring more counters. Even complete sorting can be checked for this way, and in this case the criterion is that the sum of entries in the transitive-closure array reaches $\binom{n}{2}$.

Equivalent states. The need to decide which states may be considered equivalent appears early in the design process. Our first approach was natural and proved naïve: the graphs of such states must carry identical information about each pair of entries, as given by the transitive closure of the comparison graphs. As seen shortly, where we discuss state counts, this creates way too many equivalence classes. We had to consider equivalent states which have *isomorphic* graphs, not just identical.

Graphs are isomorphic when a bijective mapping T on their sets of nodes exists, such that one has an arc (c, d) if and only if the other has the arc $(T(c), T(d))$, for all pairs c and d. Two equivalent states will not only require exactly the same amount of work to resolve optimally, but their optimal policies are the same, modulo the mapping T.

Identifying isomorphic variations and retaining just one of each such class leads to a significant reduction in the number of states that need to be maintained for the final evaluation, and to material cost savings in solving the recurrence, both in space and in time. We discuss further the mechanics of handling graph isomorphism following an outline of the entire computation.

Outline of calculations. The solution we implemented proceeds in three phases. In the first, the recurrence (1) is iterated, starting with a suitable initial state (usually, for the first action we made a single, arbitrary decision, as in the minimal example given before). Putting this state in a queue is the final initialization step. The entire phase is a loop over dequeuing a state, selecting all the possible actions according to its graph transitive closure; then, if needed, calculating their probabilities, and enqueueing the states which result and are not: (i) yet resolved nor (ii) equivalent to a state already processed. When a new such state is enqueued it is the first in its equivalence class, and it is entered into a Binary Search Tree (BST). The compacted transitive-closure array is used as the key in the tree. The choice between proceeding in a depth-first or breadth-first manner in constructing our state network was determined in favor of the second. There are several advantages and disadvantages to each, and BFS seemed the more natural.

This calculation creates the basic network. It can be seen as having alternating layers of nonequivalent states and of actions. The set of arcs leaving a state is the action set $C(S)$ defined earlier (it does not depend on the notion of equivalent states used). Each arc terminates at an *action block* which carries information about this action: possible outcomes, the probabilities of these outcomes (if needed), the states they result in, and some additional information we describe later. It also has a field for the cost of this choice of action, denoted by $U(S \mid \varepsilon)$ in (1) which is computed in the second phase. Arcs go from action blocks to the states they create, if they are yet unresolved (actually, to the representatives of the equivalence classes of the outcome states). When the queue is depleted, the network is completed.

The second phase uses the network to perform the optimization seen in the recurrence (1). This is a recursive, DFS-like process which generates the optimal comparison count. During this phase the optimal choice of action is recorded for each state in the network, as well its (optimal) value; the last to be filled is the block of the initial state, with the cost of the optimal algorithm.

The final phase visits a skeleton in the network, beginning at the initial state, only using in each state the arc to the optimal action, and producing the final result: a C-language function that performs the optimal selection or sorting algorithm. It is a function only a compiler would love, as seen in the Appendix.

It is possible to view this calculation as evaluating a coöperating (nonadversarial) game tree; alternating layers are controlled by two players who select operations. One tries to confound the other through abundance of choice, and one helpfully culls down all nonoptimal choices. Here alternating layers hold the states and the action blocks; the states deploy in the next layer all nonredundant actions, nonselectively (since we know of no meaningful a priori selection criterion that maintains the guarantee of optimality). The action layer is the helpful one here; it is "controlled by nature," which determines the probabilities of the two outcomes of each action and tests for possible resolution.

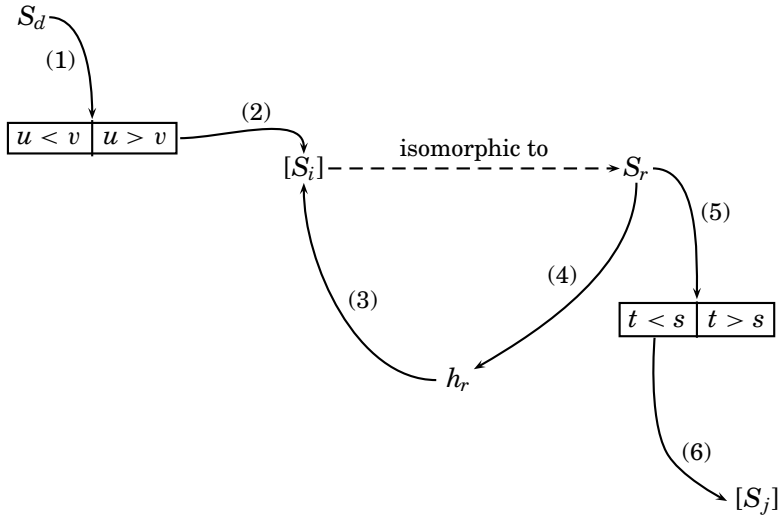
Graph isomorphism. Deciding whether two graphs are isomorphic is considered a hard problem (its complexity class is not yet quite known). A survey of much that is known about it is in Conte et al. [2004]. The usual approach is to define a set of criteria for graphs to be in canonical form,[1] and call two graphs isomorphic if their canonical forms are identical. The canonization of a graph requires, in principle, to find a permutation of the graph labels which converts it to a canonical form. The text Kreher and Stinson [1998] calls this calculating a certificate for the graph. For our experiments we used Brendan McKay's package *nauty* to "canonize" the transitively closed state graphs.[2] A detailed exposition of the algorithm is given in Hartke and Radcliffe [2009]. It is generally considered one of the finest (and most efficient) available. In addition to producing a canonical isomorph h for a state graph g, it returns a certificate, an array L with the permutation carrying the "canonical" labels h_i to those of the state graph g: $g_i = L[h_i]$, $1 \leq i \leq n$. As states evolve and are assigned to different class representatives, we need to keep track of their "permutational genealogy." These relationships are illustrated in the diagram that follow. The numbers on the arcs key the description.

As canonical graphs (= representative states) arise, they are inserted into a binary search tree. As keys we use the diagonal-less $n(n-1)$-bit adjacency matrices of their transitive closures compacted and split into computer-word long segments. These are used successively as keys in the tree insertion and search functions. Only one representative is maintained for each equivalence class, and other states in the network which

[1] A variety of terms are used; canonical labeling and canonical isomorph are common.

[2] The original package is available through its Web site, in *http://cs.anu.edu.au/~bdm/nauty/*. An update is at *http://pallini.di.uniroma1.it/*.

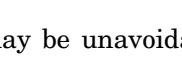
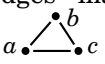
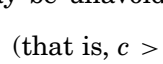
are isomorphic to it survive only as data fields in the action blocks that lead to them: (i) a pointer to the class representative state, and (ii) the permutation L that carries the canonical graph labels to those of the "eliminated" state, shown as arc (3) in the diagram. In addition each class representative carries the inverse of the permutation L described before on arc (4); it is a field in the state descriptor block. The following diagram shows how we keep track of state and action permutational relations.



Here is a possible sequence of operations when the network was constructed, resulting in the diagrammed configuration, shown as viewed during the search for the optimal policy. The state S_d spawned a number of actions; one of them, linked to the state with the arc (1) compares the array entries u and v. If $u > v$, the configuration denoted by S_i is obtained; we used *nauty* to find the canonical labeling corresponding to this configuration, h_r, and then search for it in the binary search tree of canonical representations. It was found there, with a pointer to the state S_r, which was the first in this equivalence class to be created (there could be others; for $n = 8$ the average number of states per equivalence class, in the network created to find $K_{8,4}$, was over 10, with high variance).[3] When the state S_r was created, its canonical representation h_r was calculated and inserted in the binary search tree. As said, *nauty* returns also the permutation L_r that would carry the labels of S_r to h_r. We invert it, and store the inverse permutation L_r^{-1} in the state descriptor of S_r; arc (4) represents this relation. No state structure is created for S_i, and the action block given before points directly at S_r (through the dashed arc). Each such action block also holds a permutation array (for each of the two comparison outcomes); when the action leads to a "real" state (which is either a class representative, such as S_r or a resolved state), it contains the identity permutation, but here, for S_i, it holds instead the permutation obtained by multiplying the permutations L_i, of arc (3) by the permutation L_r^{-1}. The result is of state S_d "seeing" the outcome of the action $u : v$, when $u > v$, leading to S_r with the labels properly scrambled. It is possible that an ancestor of the state S_r is not selected by the optimal policy, but S_d is selected, and therefore we need to know about the optimal continuation from S_r and use it. If that continuation is to proceed along arc (5), comparing t and s, the

[3]The advantage of using equivalence is not limited to the order-of-magnitude saving in the amount of storage and processing that this ratio would suggest; it is due to the exponential increase in additional storage and processing time that would have been required for the progeny of the now-eliminated states.

entries that get compared following $u > v$ in S_d are $L_i L_r^{-1}(t)$ and $L_i L_r^{-1}(s)$. In this way we need not propagate the permutations any further.

State count. Creating the network is the main computational task; this is the time needed to assemble the data structures, to determine, for each state, the possible continuing actions, and for the expected performance criterion, to compute their transition probabilities. Estimating the number of states that take part in the process was important in designing storage allocation, the critical resource here. No formula is available. Clearly, we only need to use a fraction of the humongous number of graph states: many potential states include arcs made superfluous by transitivity (as shown shortly), or contain cycles, or have more arcs than needed to resolve the final state. The *optimal* algorithms, generated as output, use modest numbers of states; for example, such an algorithm to locate the median of seven entries uses just under 400 states, out of the ten billion or so possible ones ($3^{\binom{7}{2}} = 10{,}460{,}353{,}203$, to be precise). The search for it needs more than 400, many more. The main factor in the number of states we need to consider is the definition of equivalent states. If states need to be identical to be considered equivalent, the relevant sequence is denoted by d_n in Comtet [1974, page 60], a sequence defined as the number of posets with n labeled elements (from our point of view this are labeled acyclic transitive digraphs), and given as sequence A001035 in Sloane's encyclopedia of integer sequences (http://oeis.org). However, as can even be seen in the minimal example given earlier, states that have "transitively superfluous edges" may be unavoidable: whichever way we end resolving $S =$  , either  (that is, $c > b$), or  , which says $b > c$, one of the arcs already in S becomes an unnecessary "transitive arc". However, when S is all we have, we do not know which one. As remarked before, if you pick them right, with hindsight, no more than $n - 1$ comparisons are needed for any order-related objective to be reached.

As n grows, such occurrences multiply and we end up looking at significantly more than d_n states. Thus, while $d_7 = 6{,}129{,}859$, our initial calculation constructed a network with 110,802,315 states, and in addition, the algorithm created 268,224,165 states which were found identical to previously logged states. This is still under 4% of the total, but exceeds d_7 by a factor of 60.

Such were the numbers that justified adopting isomorphism as the equivalence criterion. The relevant sequence of nonisomorphic labeled poset counts is denoted by Comtet [1974, page 60], and prefixes of both sequences are shown in the table that follows. The third line shows u_n, the number of nonisomorphic states used in our calculation for optimal median policies; this is a tallied value, not a calculated one.

n	1	2	3	4	5	6	7	8	9
d_n		3	19	219	4231	130,023	6,129,859	431,723,379	44,511,042,511
d_n^*	1	2	5	16	63	318	2045	16,999	183,231
u_n			2		54	291	1971	16,618	181,773

d_n – the number of transitive labeled DAGS; d_n^* – the number of their nonisomorphic classes.

We notice that while $d_n^* \ll d_n$, this sequence also grows superexponentially fast; $d_{11}^* = 46{,}749{,}427$ is the largest we needed to know (actual number used: $u_{11} = 46{,}687{,}305$). The reason why we need an increasing fraction of the possible nonisomorphic states as n increases, while untutored intuition points the other way (expecting relatively more overresolved states), has not been answered as yet.

3. OBSERVATIONS

Selection versus sorting. While the initial impetus for this work was the need for optimal *selection* functions, specifically for the median, it was soon evident that the same approach lends itself to producing optimal algorithms for other selection problems as well, such as the simultaneous selection of several order statistics, and even for complete sorting. It was fascinating to observe how the optimal algorithms for selection and sorting differ. Animation might be more illuminating here; limited to verbal description, we saw that selection "prefers" to crowd edges together; for example, the optimal policies for median selection among 5 or 7 elements start by sorting three elements (we took advantage of this in the example shown in the Appendix for the median of five, where the initial sorting is done with exchanges, since it keeps the function much shorter than the assignment-free version). For larger n it creates less-orderly clusters. The optimal sorting algorithms, on the other hand, start by "running around," and ordering independent pairs, and only when these are exhausted, or nearly so, they start connecting the pairs[4]. A close look reveals that the sorting algorithm for small sets performs, in effect, an in-place (assignment free), bottom-up merge-sort.

Sorting and selection displayed another difference that was initially puzzling: the minimal number of comparisons needed to complete either of these tasks is $n-1$. Mean-minimal *selection* functions, for a single order statistic, show this outcome for certain permutations. Otherwise, over the set of all permutations, they generate sequences of comparisons whose lengths have a range of values, from this minimum to nearly double (4 to 7, to select the median of 5, 8 to 16 for $n = 9$). But optimal algorithms for selecting simultaneously several order statistics have a more restricted range of possible costs. At the extreme, the mean-minimal sorting policy does not go there at all; when the optimum cost is some number x, all permutations are sorted by the optimal policy in either $\lfloor x \rfloor$ or $\lceil x \rceil$ comparisons. The mystery was resolved in part by Knuth [1999, Exercise 5.3.1.20], which shows that this is a necessary condition for a mean-minimal comparison sort, due to the combinatorics of binary trees. No other choices need be seen. The contribution of our tool is to create the comparison decision tree needed to affect this efficient sort.

What the proof (in the cited exercise) actually shows is that the cost of sorting all permutations is equal to the External Path Length (EPL) of the comparison decision tree (seen as an extended tree), and that the EPL of such a tree with a specified node count is minimized when all the external nodes (leaves) occupy the bottom one or two levels. For selection, however, the mean-min is not closely related to the EPL, because more than one, typically many, permutations can share the same path in the optimal decision tree; the optimality is obtained by concentrating permutations and "routing" them to early external leaves, rather than to such with long paths.

Worst-case performance. We have mentioned previously the possibility (suggested by Stanley Selkow) of replacing the expectation operator in Eq. (1) by a maximizing operator over the two possible outcomes of any action ε chosen. This converts the recurrence to one for the optimal cost under a worst-case scenario, and just as for the stochastic case, the optimal policies can be read-off from the network of states that survive the final, depth-first-search-like calculation step, when actions are selected that minimize the cost function

$$W(S) = 1 + \min_{\varepsilon \in C(S)} \max_{\{\varepsilon\}} W(S \mid \varepsilon), \qquad S \notin \mathcal{S}_r, \tag{2}$$

[4]We were reminded of the similarly contrasting appearances of two standard greedy algorithms for computing a minimum spanning-tree, the Kruskal and Prim algorithms.

where the symbol $\{\varepsilon\}$ stands for the set of two outcomes of the action ε, and the same initial state and termination conditions are used. The actual calculation is very similar and avoids the need to manage the lists of conforming permutations used to compute the branch probabilities. The bulk of the computation, which is constructing the network of information states, is identical for both recurrences. While we did not choose to do so, it is possible to evaluate both recurrences on the same network, once it is set up.

The search for constrained optimality. It is possible to define several mixed optimization criteria. We experimented with minimizing the expected number of comparisons, while not exceeding a certain number of comparisons. We selected a limit not smaller than the bound produced by the recurrence in Eq. (2). Smaller choices, which forfeit completion over some permutations, could be of interest as well, as is the search for an algorithm maximizing the number of permutations which can be resolved in one or two fewer comparisons than the bound that resolves all of them. To adapt the computational method to the constrained objective we add to the classification of states by isomorphism (of the transitive closure of the comparisons) also the requirement that they are reached at the same cost. The change increases significantly the number of state classes, but the cost is offset by discarding all states that exceed the bound.

For example, we found that the cost of the mean-optimal algorithm to locate the median of 9 is 12.789594 comparisons (see the second table in the next section), using 9 to 18 comparisons. The worst-case bound is 14, as shown by Noshita [1974], and confirmed by the solution of Eq. (2). The average number of comparisons his algorithm makes is 13.187302, which is not optimal, but remarkably close: the optimal algorithm which makes at most 14 comparisons has been found to need 13.136816 on the average (making 10 to 14 comparisons).

Note. The algorithm we generated when verifying Noshita's bound (solving Eq. (2) with no attempt to maximize the information content of the selected actions) is less efficient, making 13.534392 comparisons on the average.

4. NUMERICAL RESULTS

While the various figures of merit associated with the optimal algorithms were of marginal interest in this work, we used the known ones to direct our initial searches, and believe it is right that we complemment them as much as we can. Our caclculations have verified all the values for $V_k(n)$ and $\overline{V}_k(n)$ that are provided in Tables 1 and 2 in Knuth [1999, Section 5.3.3] (as well as generated the corresponding algorithms). The following are some additional values. First, we give the worst-case optimal bounds for selection of a single order statistic among 11 entries. We omitted the trivial cases $V_1(n) = V_n(n) = n - 1$,

$V_2(11)$	$V_3(11)$	$V_4(11)$	$V_5(11)$	$V_6(11)$	$V_7(11)$	$V_8(11)$	$V_9(11)$	$V_{10}(11)$
13	15	17	18	18	18	17	15	13

While both $K_{11,5}$ and $K_{11,6}$ require up to 18 comparisons, the particular functions generated for them used 17.219791 and 17.430487 comparisons on the average, respectively; the median *is* more expensive.

We could add more entries to Table 2 of Knuth, for the optimum in expectation, again omitting the trivial end values (which are identical to the preceding).

n	$\overline{V}_2(n)$	$\overline{V}_3(n)$	$\overline{V}_4(n)$	$\overline{V}_5(n)$	$\overline{V}_6(n)$	$\overline{V}_7(n)$	$\overline{V}_8(n)$	$\overline{V}_9(n)$
8	$8\frac{9}{10}$	$10\frac{16}{105}$	$10\frac{283}{315}$	$10\frac{283}{315}$	$10\frac{16}{105}$	$8\frac{9}{10}$		
9	$10\frac{1}{90}$	$11\frac{51}{112}$	$12\frac{8869}{22680}$	$12\frac{4477}{5670}$	$12\frac{8869}{22680}$	$11\frac{51}{112}$	$10\frac{1}{90}$	
10	$11\frac{5}{42}$	$12\frac{18593}{25200}$	$13\frac{24817}{30240}$	$14\frac{201533}{453600}$	$14\frac{201533}{453600}$	$13\frac{24817}{30240}$	$12\frac{18593}{25200}$	$11\frac{5}{42}$

The functions which were created when the aforesaid numbers were computed are available in *http://www.cs.wpi.edu/~hofri/Selection*. The mean-optimal function for $K_{n,k}$ is called *mn_k* (e.g., *m6_3*), takes a single argument, the name of an array of n integers, and is in the file *mn.k.c* in the given directory. For the worst-case optimal functions replace *m* by *w*.

Limited experimentation with more complex selections produced little new information. We found it of interest to compare the costs of obtaining single and multiple order statistics: while $K_{6,2}$ and $K_{6,3}$ require, optimally, $6\frac{1}{2}$ and $7\frac{7}{18}$ separately, doing them at once produced an optimal policy that needed only $7\frac{8}{9}$ comparisons on the average (and came closer to the optimal sorting algorithm in having a narrower range of comparison counts until completion, requiring 7, 8, or 9 comparisons, for all permutations).

5. CONCLUSION

The preceding sketch of the computational process hides a relatively complex programming task. Little is routine when dealing with structures that strain the capabilities of a conventional computer, and surprising trade-offs hide at every turn. That is the reason why the method was presented in the Introduction as valid for small problems only. Partial relief is possible by using parallelization, but until we find a reasonable way to deal with the vast number of information states needed for even slightly larger problems ($n = 10$ was a challenge on a desktop computer of the year 2011;[5] searching for the optimal policy for $\overline{V}_6(11)$ required 136GB of main storage) the limitation is acute. There are some coding techniques that could save considerable space (paid for by computing time), but the order-of-magnitude saving is not enough to accommodate $n = 12$, where the standard computer architecture would add a few more bounds that need be overcome, such as using arrays with more than 2^{32} entries.

So much for direct numerical solution. This is related to another unresolved issue. Here is more from Bellman, on the same page mentioned earlier: "Assume ... that we have circumvented all ... difficulties and have attained a certain computational nirvana. Withal,[6] the mathematician has not discharged his responsibilities. *The problem is not to be considered solved in the mathematical sense until the structure of the optimal policy is understood*" (the italics are in the original). In this problem, if we look at the various optimal costs, it is easy to discern trends, but no exact numerical pattern suggests itself, nor is one likely. Also, even when we know our optimal policies, they do not imply readily any properties or understanding of their structure. Not surprising when we observe that the functions generated by the calculation, while optimal, and indeed very efficient, doing little beyond branching on a few comparisons, span large trees, and run to amazing lengths, often many thousands of lines (the function that locates $K_{10,5}$ is 16,555 lines long; to sort 10 elements we need to construct a tree with 10! leaves, more than 3.6 million). Understanding their structure might enable us to design compact optimal functions and generate such algorithms for higher values of n, without

[5]The computers used were conventional, with Intel X86 architecture processors, mostly AMD Opteron 280 and similar, with the SuSE-11 distribution of Linux, and used the gcc, 4.6.2 compiler. One had the distinction of being fitted with a large main storage of 136GB; the two others had 4 to 12GB.

[6]"At the same time; in spite of all; notwithstanding, nevertheless." (the OED)

solving Eq. (1), or at least suggest useful pruning of its network; this we cannot yet do: scrutinizing the structure of smaller selection decision trees did not provide the light.

A weaker objective would be discovering properties of the network generated by the metaprogram that can be used to expedite the search for the optimal algorithm. One such observation we used was that when looking for a selection policy, in a state where the test for resolution identified some of the entries as too extreme (too large or too small; we call them inadmissible, or taboo entries), we generate no actions involving the taboo entries, based on the following.

Conjecture 1: No optimal policy includes comparisons which involve entries known then to be inadmissible.

The conjecture appears reasonable, but we have not found in our framework a proof for it. Several tests looking for a counterexample (searches for optimal policies without using this conjecture) upheld it: we never found a more efficient algorithm this way.

It is often a tenet of belief that optimal policies for similar states have commonalities, providing for a structure of the entire policy, but none has been proven, and so far—nor observed. Several hypotheses came up when we looked at early calculations, for five or seven elements (such as: to find the median, first sort three of the elements). Further calculations, for larger sets, disabused us of them, proving to be yet another manifestation of the principle that "there are not enough small integers" [Guy 1990].

Several possible improvements on the computations we performed can be outlined. For example, the generated functions are longer than they need be; lines where both results of a comparison of the same pair resolve the state in the same way can be found and folded, recursively (without changing the performance, naturally). This would only affect the size of the algorithms, not their functionality. Since the possible reduction appears marginal, we made no such effort.

And yet, the beginning of this work was largely under the shadow of a lament made by Knuth, in the same section cited earlier, that average-case selection is much harder than solving for the worst-case criterion. In a sense, this does not seem now to be the case: we have now a tool, an automaton: give it a powerful device to run on and it produces results at the turn of the handle, and only when nearly all the work is done do we need to tell it which criterion to use in the optimization. Two considerations intervenem, however, and justify the lament: Expectation *is indeed more* computationally intensive than maximization, because of the need to handle the probabilities, which require either much more storage or time. The other consideration is that if one is only after the numbers, the various V and F and their flavors, and is not interested in the policies that achieve them, then the worst-case results can be obtained by a far more efficient search, as in Gasarch et al. [1996], but we do not know of such a short-cut for the mean-min values.

APPENDIX

The function *med5*(Q) given next was generated by our program; it selects the median of five elements in the array Q. For the sake of brevity we use the observed fact that when the metaprogram starts with the standard initial state (one comparison, between the first two items), the optimal algorithm proceeds to sort the first three entries, and we do it here directly, symmetrizing over the first three terms, and achieve the expected mean-minimum of $704/120 = 5\frac{13}{15}$ comparisons. The number of comparisons for each permutation is in the range 4 to 7. Giving up on the symmetrization provides an assignment-free function, with slightly over six times the number of labels (58).

```
int med5(int *Q) { int t;
if (Q[0]>Q[1]) swap(&Q[0], &Q[1]);
if (Q[2]<Q[1]) {if(Q[2] > Q[0]) swap(&Q[1], &Q[2]);
           else  {t=Q[2]; Q[2]=Q[1]; Q[1]=Q[0]; Q[0]=t;}}
```

```
L0: if(Q[1]>Q[3]) goto L1; else goto L2;
L1: if(Q[1]>Q[4]) goto L3; else return Q[1];
L2: if(Q[1]>Q[4]) return Q[1]; else goto L4;
L3: if(Q[0]>Q[3]) goto L6;   else goto L5;
L4: if(Q[2]>Q[3]) goto L7;   else goto L8;
L5: if(Q[3]>Q[4]) return Q[3]; else return Q[4];
L6: if(Q[0]>Q[4]) return Q[0]; else return Q[4];
L7: if(Q[3]>Q[4]) return Q[4]; else return Q[3];
L8: if(Q[2]>Q[4]) return Q[4]; else return Q[2];
}
```

ACKNOWLEDGMENTS

Discussions with Stanley Selkow, Dan Dougherty, and William Martin, all at WPI, shed much light on the question. Several attentive referees provided useful references and posed questions which helped to improve the presentation.

This article is dedicated to the memory of the author's host at INRIA, Philippe Flajolet, 1948–2011.

REFERENCES

Battiato, S., Cantone, D., Catalano, D., Cincotti, G., and Hofri, M. 2000. An efficient algorithm for the approximate median selection problem. In *Proceedings of the 4th Italian Conference on Algorithms and Complexity*. Lecture Notes in Computer Science, vol. 1767, Springer, 226–238.

Bellman, R. 1957. *Dynamic Programming*. Princeton University Press. (The 1972 6th printing of the book was reissued in 2003 by Dover Publications).

Cantone, D. and Hofri, M. 2013. Further analysis of the remedian algorithm. *Theor. Comput. Sci.* (to appear).

Chern, H.-H. and Hwang, H.-K. 2001. Transitional behaviors of the average cost of quicksort with median-of-(2t+1). *Algorithmica 29*, 44–69.

Omtet, L. 1974. *Advanced Combinatorics*. D. Reidel Publishing.

Conte, D., Foggia, P., Sansone, C., and Vento, M. 2004. Thirty years of graph matching in pattern recognition. *Int. J. Pattern Recogn. Intell. 18,* 3, 265–298.

Gasarch, W., Kelly, W., and Pugh, W. 1996. Finding the ith largest of $n$for small i, n. *ACM SIGACT News 27,* 2, 88–96.

Guy, R. K. 1990. The second strong law of small numbers. *Math. Mag. 63,* 1, 3–20.

Hartke, S. G. and Radcliffe, A. J. 2009. Mckay's canonical graph labeling algorithm. In *Communicating Mathematics: A Conference in Honor of Joseph A. Gallian's 65th Birthday*, T. Y. Chow and D. C. Isaksen, Eds., Contemporary Mathematics, vol. 479, American Mathematical Society, 99–111.

Knuth, D. 1999. *The Art of Computer Programming.* Vol III: Sorting and Searching 2nd Ed. Addison-Wesley.

Kreher, D. and Stinson, D. 1998. *Combinatorial Algorithms: Generation, Enumeration, and Search*. CRC Press, Boca Raton, FL.

Noshita, K. 1974. Median selection of 9 elements in 14 comparisons. *Inf. Process. Lett. 3,* 1, 8–12.

Peczarski, M. 2002. Sorting 13 elements requires 34 comparisons. In *Proceedings of the 10th Annual European Symposium on Algorithms (ESA'02)*. R. Mohring and R. Raman, Eds., Lecture Notes in Computer Science, vol. 2461, Springer, 785–794.

Ross, S. 1983. *Stochastic Dynamic Programming*. Academic Press.

Received February 2013; accepted May 2013