VISOR: vast independence system optimization routine

Citation for published version (APA):

Jager, de, A. G., & Banens, J. P. A. (2001). VISOR: vast independence system optimization routine. *Algorithmica*, 30(4), 630-644. <https://doi.org/10.1007/s00453-001-0030-x>

DOI:

[10.1007/s00453-001-0030-x](https://doi.org/10.1007/s00453-001-0030-x)

Document status and date:

Published: 01/01/2001

Document Version:

Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

- A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
- The final author version and the galley proof are versions of the publication after peer review.
- The final published version features the final layout of the paper including the volume, issue and page numbers.

[Link to publication](#)

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy

If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

VISOR: Vast Independence System Optimization Routine

B. de Jager¹ and J. Banens¹

Abstract. An algorithm is sketched that generates all K maximal independent sets and all M minimal dependent sets of an arbitrary independence system, based on a set of cardinality n having at most 2^n subsets, with access to an oracle that decides if a set is independent or not. Because the algorithm generates all those sets, it solves the problems of finding all maximum independent and minimum dependent sets. Those problems are known to be impossible to solve in general in time polynomial in n , K , and M , and they are $\mathcal{NP}$ hard. The algorithm proposed and used is efficient in the sense that it requires only $O(nK + M)$ or $O(K + nM)$ visits to the oracle, the nonpolynomial part is only related to bitstring comparisons and the like, which can be performed rather quickly and, to some degree, in parallel on a sequential machine. This complexity compares favorably with another algorithm that is $O(n^2K^2)$. The design of a computer routine that implements the algorithm in a highly optimized way is discussed. The routine behaves as expected, as is shown by numerical experiments on a range of randomly generated independence systems with n up to $n = 34$. Application on an engineering design problem with $n = 28$ shows the routine requires almost 10^6 times less visits to the oracle than an exhaustive search, while the time spent in visiting the oracle is still significantly larger than that spent for all other computations together.

Key Words. Combinatorial optimization, Independence system, Maximal independent set, Input/output selection

1. Introduction. The problem of establishing the structure of an independence system is of interest for many practical problems. This structure is characterized by the maximal independent and/or minimal dependent sets. In general this characterization is not possible in time polynomial in the problem size and in the cardinality of the family of sets that completely characterizes the solution [1]. Unfortunately, only a few of the problems have enough additional structure to allow an efficient solution. For instance, problems with matroids, having one additional property, are relatively easy to solve with greedy algorithms [2]. A few other problems with additional structure are discussed in [1].

In this paper no other structure is assumed besides the independence system structure. So the results are applicable to arbitrary independence systems. It therefore tackles the problem that is hard to solve. To be able to analyze and improve existing algorithms we make one assumption: that it is expensive to evaluate the independency of a set and to determine if it belongs to the family of independent sets. This assumption is motivated by practical problems arising in engineering, where costly computations, e.g., based on finite-element methods, are used to evaluate designs.

¹ Department of Mechanical Engineering, Eindhoven University of Technology, P. O. Box 513, 5600 MB Eindhoven, The Netherlands. A.G.de.Jager@wfw.wtb.tue.nl.

Now, when the solution algorithm uses a relatively costly device, here called an oracle, to test if a set is in the family of independent sets, and when a complete solution can be found in a number of visits to the oracle that is polynomial in the size of the problem and in the cardinality of the family of sets characterizing the complete solution, the solution is also generated in a time with this complexity if n is not too large, because oracle visits will dominate the computation.

The algorithm proposed does obtain a low degree polynomial complexity of the type sketched above and compares favorable with other algorithms, particularly the ones analyzed in [1].

Section 2 supplies theoretical background of the proposed problem and introduces some notation. Next, Section 3 proposes the algorithm, while Section 4 discusses the routine that implements the algorithm. A demonstration of the proposed method on synthesized problems and application to a large-scale selection problem for a tractor-semitrailer active suspension design in Section 5 illustrate the use of the routine that was developed. Section 6 concludes the paper.

2. Preliminaries. Let E be a finite set with cardinality $|E| = n$, and let $\mathcal{I}$ be a nonempty family of subsets of E that satisfies the following rule: if $I \in \mathcal{I}$ and $I' \subseteq I$, then $I' \in \mathcal{I}$. Now $(E, \mathcal{I})$ is called an independence system and $\mathcal{I}$ is its family of independent sets. An independent set I is called maximal if there is no $I' \in \mathcal{I}$ such that $I' \supset I$.

Subsets of E that are not in $\mathcal{I}$ are dependent sets. A dependent set J is minimal if $J' \in \mathcal{I}$ for all $J' \subset J$. All dependent sets form the family $\mathcal{J}$.

Now the problem is to establish the structure of the independence system, that is, to find $\mathcal{I}$ and/or $\mathcal{J}$. To do this an oracle is available that decides if a subset of E belongs to $\mathcal{I}$ or $\mathcal{J}$. Because the oracle is costly in terms of computing time it should not be visited too often, so an efficient method is needed. In general it suffices to find the K maximal independent sets of $\mathcal{I}$ or the M minimal dependent sets of $\mathcal{J}$, because with these sets one can generate the families $\mathcal{I}$ and/or $\mathcal{J}$ without visiting the oracle. Given those sets it is also easy to establish the maximum independent and minimum dependent sets, i.e., the largest sets among the maximal independent sets, respectively the smallest among the minimal dependent sets.

It is possible to find $\mathcal{I}$ by $O(n|\mathcal{I}|)$ visits to the oracle [1], but $|\mathcal{I}|$ is bounded by 2^n and may be very large. One could conceive a more efficient method by only trying to find the K maximal independent or M minimal dependent sets, but the number of these sets may grow combinatorially with n because both K and M are bounded by $\binom{n}{n/2}$. This implies that one cannot guarantee obtaining a solution in time polynomial in n .

One could ask if a solution in time polynomial in n and K or M is possible. However, Lawler et al. [1] state for an arbitrary independence system that if $\mathcal{P} = \mathcal{NP}$, then an algorithm for finding all K maximal independent sets and all M minimal dependent sets in time polynomial in n and $K + M$ does exist. They also provide an “only if” version of this statement. Because it is conjectured that $\mathcal{P} \neq \mathcal{NP}$, it is unlikely that the problem is solvable in time polynomial in n and K or M . For more information about the complexity classes $\mathcal{P}$ and $\mathcal{NP}$ see, e.g., [3] and [4].

If we assume that only the visits to the oracle have to be paid, and all other operations are free, a reasonable assumption for high tech engineering problems if n is not too large, the situation is brighter. As we will sketch, it is possible to establish all K maximal independent sets and all M minimal dependent sets visiting the oracle only $O(nK + M)$ or $O(K + nM)$ times. This means that a solution with visits polynomial in n , K , and M , and, if n is not too large, also time polynomial, is realizable.

In [1] an analysis is given of the complexity of a generalized Paull–Unger procedure with improvements by Tsukiyama et al. [5]. This recursive procedure has a bound on the number of visits that is at least $O(n^2 K^2)$, so the complexity of the algorithm proposed here is significantly better with respect to the number of visits.

3. Algorithm. The algorithm is described as if we want to find minimal dependent sets. It could also be described as if finding maximal independent sets is the goal. Because the algorithm establishes both sets and can easily be tricked into swapping maximal independent sets and minimal dependent sets, this does not matter.

All (sub)sets of E are elements of the power set $P(E)$. To represent these elements we use their characteristic vectors, i.e., bitstrings of length n . Let X be the set of strings of binary digits of length n , so $X = \{x \mid x_i = \{0, 1\}, i = 1, \dots, n\}$. Let $s(x)$ be a function that counts the number of ones in x , which is called the weight of the bitstring. Define the ordering relations $\geq$ and $\leq$ so that $y \geq (\leq) x$ if and only if $y_i \geq (\leq) x_i, i = 1, \dots, n$, $x, y \in X$. The ordering is partial, because there are $x, y \in X$ for which neither $x \leq y$ nor $y \leq x$ hold, e.g., 101 and 010 are not $\leq$ nor $\geq$ with respect to each other. The subset inclusion relation corresponds with the order relation defined for the bitstrings. The power set and the partial order combine to a poset.

The oracle is a mapping $t(x)$ from X to $\{0, 1\}$ with 0 as outcome when it says “independent,” and 1 when it says “dependent.” The mapping t has the following properties, that correspond with the structure of an independence system:

1. If $t(x) = 1$, then also $t(y) = 1$ for all $y \geq x$.
2. If $t(x) = 0$, then also $t(y) = 0$ for all $y \leq x$.

The goal is to determine the pre-image X_j of 1 under the mapping t , so $X_j = \{x \mid t(x) = 1, x \in X\}$, or the pre-image X_i of 0. It is often sufficient to establish the “boundary” between X_j and $X_i = X/X_j$, i.e., the maximal independent or minimal dependent sets.

For visualization the problem can be cast in a graph setting: associate each $x \in X$ with a node, draw all nodes x with $s(x) = l$ at the same level l and connect them with those nodes at level $l + 1$ and $l - 1$ that are $\geq$ and $\leq$. This creates a Hasse diagram. An example for $n = 4$ is displayed in Figure 1. The set X_j is a collection of nodes at the “upper” part of the diagram, while X_i is the remaining “lower” part.

The basic algorithm employed to find the M minimal dependent sets traverses the levels in the Hasse diagram sequentially and from the top to one level above the empty set, so $l = n, \dots, 1$. There is no need to go to level $l = 0$. It works as follows:

init Create a list L_j of bitstrings representing minimal dependent sets and a list L_i representing maximal independent sets. Initially these lists are empty. Set $l = n$.

Fig. 1. Hasse diagram for $n = 4$.

level Generate a new bitstring y with $s(y) = l$. Reject the bitstrings y for which the next tests hold.

- $\exists x \in L_j$ for which $y \geq x$ or $y \leq x$.
- $\exists x \in L_i$ for which $y \leq x$.

Each y that is not rejected by the bitstring ordering tests pays a visit to the oracle.

- If $t(y) = 1$: search depth-first for a minimal dependent set representation $y' \leq y$, so $s(y') \leq l$, and add y' to L_j .
- If $t(y) = 0$: add y to L_i .

Repeat for all bitstrings of weight l .

crit If $l = 1$ or no y with $s(y) = l$ has paid a visit to the oracle then stop, else repeat **level** with $l \leftarrow l - 1$.

The depth-first search is performed as follows. The 1-bits in the bitstring y are turned off one-by-one in a certain order (see Section 4.3) and each time the oracle is visited. If $t(y') = 1$ the bit is kept at 0, else the bit is put back to 1. In both cases the search proceeds by turning off the next 1-bit until all 1-bits are processed.

An analysis shows that the depth-first search needs exactly $l + 1$ visits to produce a minimal dependent set at any level l' , starting from level l . For n , the largest value of l , this is equal to $n + 1$. More details of this analysis are given in [6].

Because we update L_j after finding a minimal dependent set, each visit at level l with $t(y) = 1$ is guaranteed to yield a new minimal dependent set, of which there are M . So, the depth-first part of the algorithm needs at most $O(nM)$ visits.

Because L_i is updated each time a new maximal independent set is found, a visit with $t(y) = 0$ yields a new maximal independent set, of which there are K . This generates exactly K visits. In total the algorithm requires $O(K + nM)$ visits.

A straightforward variation of this algorithm, starting the search from $l = 0$ toward $l = n - 1$, generates all K maximal independent sets and all M minimal dependent sets in $O(nK + M)$ visits. One can choose this variant or the previous one, depending on the expected values of K and M .

If the bitstrings are not stored in compressed form, the two lists need $O(nK + nM)$ bit storage. Normally, a complete bitstring is stored in a word of sufficient size, so $O(K + M)$ words are needed. If each bit is stored in a word, the storage requirement is n times as large.

This analysis does not include all operations, that would imply $\mathcal{P} = \mathcal{NP}$, but the other

operations only require multiset permutation generation, bitstring comparisons, and the like, that can be performed much faster than a visit to the oracle. A rough upper bound for the number of bitstring comparisons is $O(2^n(K + M))$ and a bitstring comparison takes constant time unless n exceeds the word size. In the practical cases studied, the ratio of time spent in an oracle visit and in a single bitstring comparison is in the range $[10^7, \dots, 10^{10}]$.

Combining the results of the analysis, the polynomial part of the algorithm, related to oracle visits, scales with n and the exponential part, related to multiset permutations and bitstring comparisons, with 2^n . Both parts have components that scale linearly with K and/or M .

4. Implementation. The algorithm has been implemented in a routine called VISOR and yields results that agree with the analysis. Because it appeared that the implementation can be coded naively or highly optimized, with potential speed differences close to one order of magnitude for the bitstring comparison part, details of the implementation are highlighted.

To avoid tuning for a specific processor architecture, compiler, or problem, four different architectures, x86, SPARC, MIPS, MIPS64, two different compilers, gcc-2.8.0 and a native C compiler, and three different test problems, one with K and M large, one with K and M small, and a randomly generated one, were used. Only those measures that were beneficial for at least some of the configurations and did not hamper other ones are discussed.

The discussion is split into four parts:

1. general outline,
2. bitstring generation,
3. measures that reduce the number of tests of independency, and
4. measures that reduce the number of bitstring comparisons and improve their speed.

4.1. General Outline. The routine is structured like the algorithm presented in the previous section. Its main loop is a “level” iteration loop that starts at level n , proceeds to level 1, and guards the end-of-iteration conditions. The main loop calls a bitstring generator that generates all bitstrings of size n and weight l in a specific order. For each bitstring a function is called, that checks the bitstring by calling bitstring comparison functions and, if necessary, pays a visit to the oracle.

If the outcome of the visit is positive, $t(y) = 1$, the depth-first search function is called. The depth-first search is implemented recursively and returns with a bitstring representing a minimal dependent set. This set is stored in a dynamically allocated sequential list L_j .

If the outcome of the visit is not positive, $t(y) = 0$, a bitstring representing a maximal independent set is found which is also stored in a dynamically allocated sequential list L_i .

Furthermore, test facilities, some oracles, and functions to interface with MATLAB [7] are incorporated.

4.2. Bitstring Generator. Here the generation of all permutations of a binary string of length n and weight l is discussed. Two algorithms were implemented. The first is the one proposed in [8]. That implementation uses a double linked list as the data structure,

where each bit is stored in a single word. In our implementation all n bits are stored in a single word, so using the algorithm of [8] would require an interface routine which takes time.

Another approach was investigated, namely one based on a recursive divide and conquer type algorithm. Besides being easier and much more compact to program, it also appeared to be twice as fast for some of the cases tested, with n up to 32, so this approach was selected. The algorithm works as follows.

The initial bitstring has all 1-bits squashed to one end, assume this to be the right end of the string. For the next permutations move the leftmost 1-bit to the left, until the string boundary is reached. Then the next 1-bit from the left is moved up one place, while the 1-bits to its left are squashed against it. This is repeated until all 1-bits are in their leftmost positions.

The implementation of this algorithm is recursive. It is compactly coded in the function `perm` as follows:

```
void    perm(unsigned long bstr, int l, int n) {
    unsigned long bit=1;
    if ( l==n ) check(bstr);
    else if ( l==0 ) check(bstr-(bit<<n)+1);
    else { perm(bstr-(bit<<(n-1)),l,n-1); perm(bstr,l-1,n-1); }
}
```

The function `perm` is called with a bitstring with all 1-bits, n of them in total, in `bstr`, and the desired weight of the bitstrings, in l . The function `check` performs the bitstring comparisons and depth-first search and handles the management of the lists L_i , L_j (and L_p , for this list see later). The function `perm` generates bitstrings of size n and weight l in the standard ordering for nonnegative integers, from small to large.

Note that the degenerate case with $l = 0$ and n the size of a word, when the term $(\text{bit} \ll n)$ will shift outside the word, will not occur in our application, because the **level** iteration stops with $l = 1$, so there is no need to check for this case. The bottom-up version of the algorithm, starting from $l = 0$, is handled by the implementation by reversing the bits in the bitstrings and the answer of the oracle, so in this case also the bitstring generation is internally handled by starting with $l = n$ and proceeding to $l = 1$ at most. Basically the same algorithm could be used to start building up the bitstrings from an initial `bstr` that was zero. Then the degenerate case occurs for $l = n$.

For values of n that are rather small, e.g., $n = 20$, the generation of all bitstrings of all weights is rather fast. For larger values, e.g., $n = 34$, it tends to take considerable time, due to the sheer number of those bitstrings. Nevertheless, for larger n the bitstring generation is negligible compared with the bitstring comparisons that are performed for each bitstring, because the lists L_i and L_j tend to be larger for larger n . This is confirmed by profiling the routine for the randomly generated independence systems with increasing values of n .

4.3. Independence Test. Three measures were taken to reduce the number of visits to the oracle:

1. During the depth-first search a bitstring corresponding to a set that is known to be independent, its maximal element is already in L_i , may be encountered. To avoid

Fig. 2. Search directions.

visiting the oracle for such a bitstring, during the depth-first search bitstring comparisons are performed against elements of the list L_i .

2. During the depth-first search, sets that were found to be independent during another depth-first search may be encountered again. To avoid duplicating tests, those sets are stored in a secondary list L_p and incorporated in bitstring comparisons during a depth-first search, so the same set is never tested twice. The list L_p is cleaned up whenever a new element is added to L_i or L_p . This measure has the disadvantage that the bound on the number of bitstring comparisons is raised to $O(2^n(nK + M))$ or $O(2^n(K + nM))$. For the type of problems we consider it is more advantageous to reduce oracle visits than to reduce the number of bitstring comparisons. The bound on the memory needed is also raised to $O(nK + M)$ or $O(K + nM)$ words.
3. From the analysis of the algorithm it appears to be advantageous to find a minimal dependent set starting from as low a level l as possible. This can be influenced by trying to find first those minimal dependent sets that “screen off” a large part of the Hasse diagram. This is the case when the “inner product” of two sets is as small as possible. Choosing sequentially one of the six search directions in Figure 2, i.e., the order in which the ones in the bitstring are turned off in the depth-first search, gives better results than using a single search direction. The search directions are cyclically reused.

Using these measures the number of tests can be reduced by a factor of three to five in some cases.

4.4. *Bit Comparisons.* To reduce the number and increase the speed of bitstring comparisons the following measures, some of them go back to [9], are taken:

1. To compare all bits of a given bitstring y in parallel, the bits are stored in a single word, padded with zeros.
2. To access the sets in lists L_i and L_j readily they are stored in sequential order. The disadvantage is that sorting, etc., cannot be performed very efficiently due to the need to move the words, instead of changing a few references in a double linked list.
3. The elements of the list L_p that bubble up to level $l - 1$, so have weight $l - 1$, are sorted by Quicksort in the same collating sequence as used by the bitstring generator and moved to L_i at the end of the **level** step of the algorithm.

4. During the procedure the list L_i of maximal independent sets is naturally in weight-sorted order, so starting the bitstring comparisons with the set with the largest weight is most efficient. The comparison proceeds to sets with weight one larger than l . Because the elements of L_i with weight l are in sorted order, a single comparison suffices for this part of L_i .
5. The minimal dependent sets found are in no particular order. To use the sets more efficiently, the list L_j is sorted in buckets with sets of equal weight in the same bucket. Insertion sorting is used, while only the front of some buckets need to be moved to the tail.
6. The part of the list L_j that has weight l , the bucket at the start of the **level** step, is sorted by Quicksort in the same collating sequence as used by the bitstring generator.
7. The $\leq$ comparison related to L_j starts with the bucket with the largest weight, and proceeds to the bucket with the weight of the set y tested plus one. The $\geq$ comparison starts with the bucket with smallest weight and proceeds to the bucket with weight one smaller than the set y tested. For the equality test at level l only a single bitstring comparison for each y is needed, due to sorting the bucket with weight l .
8. The most recently found element of L_i and L_j that was compared successfully is compared first with the next y , to create an approximation to a list with the most recently found elements at the front.
9. The comparisons are optimized by avoiding loop index testing using sentinel records at the end of the (part of) the list to be searched.
10. The comparisons are optimized by avoiding all index computations and using a pointer that “slides” along the lists L_i , L_j , or L_p , so only address increments are needed. The key loops of the routine are now of the form

```
while (y & ~*++Px);
```

where the condition is true if $y \not\leq x$ and

```
while (~y & *++Px);
```

where it is true for $y \not\geq x$. Repeated reversal of the bitstring y can be avoided by storing its negated value in a register variable, although some compilers already do this for the code as given.

Both the number of bitstring comparisons and the speed of a single bitstring comparison are improved by these measures. Using measures 3–10 simultaneously yields a performance up to a factor of five better than using none of them.

A remaining improvement, that has still to be implemented, is to pack more bitstrings (or parts of) in a single word so bitstring comparisons are performed in parallel. For instance, with $n = 20$, a 64 bit word size, and byte addressable memory, eight bitstrings can be packed in three words, so a speedup by a factor $8/3$ is possible, neglecting overhead. More efficient packing of the bitstrings is possible, by putting them head to tail, but requires additional processing (shifting, masking, etc.). Some versions of packing may be inconvenient for sorting, e.g., when the bitstrings are not aligned at boundaries of words of minimal size in which the bitstring fits. For the example, a more convenient

packing would be two bitstrings of size $n = 20$ in a 64 bit word, causing negligible overhead.

Using this packing strategy could improve the speed of the implementation by an additional factor of two. Combined with the measures already taken a total speedup of an order of magnitude would be achieved, except for some specific cases.

Further improvements can be realized by an implementation for parallel computation. Although in our applications an oracle that is implemented for parallel computation would be most beneficial, in other applications it may be necessary to improve bitstring comparison speed further. This is easy to do by splitting up the lists L_i , L_j , and L_p and assigning parts of them to different processors. This can give a speedup almost linear in the number of processors.

Another approach is to split the problem in two parts, each for an independence system based on an E' of cardinality $n - 1$, when two processors are available, or in more parts if more processors are present. This will increase computational speed, but not nearly linear in the number of processors because

- the number of visits to the oracle will increase, and
- the separate solutions need to be reconciled to a solution for the original problem.

These effects are easy to see when a split-up in 2^n parts is considered. This is just exhaustive search with each element of the power set assigned to a separate processor: all elements of $P(E)$ pay a visit to the oracle and from the answers of the oracle the maximal independent and minimal dependent sets have still to be found.

5. Examples

5.1. Random Independence Systems. To test the routine and to evaluate how close the bounds on oracle visits and bitstring comparisons are as functions of n , randomly generated independence systems were used for n up to $n = 34$. For the oracle a randomized function is used with the following code:

```
return (1*(random()%n+1) >= n);
```

The time needed to visit the oracle is negligible, so this oracle is not realistic. Evaluating the oracle with Monte Carlo simulations delivered the probabilities in Figure 3 for each of the levels l between 0 and n for $n = 32$. The random oracle generates independence systems that show agreement with those of practical problems encountered, although the values of K and M tend to be larger, making the synthesized problems tougher. To average the results, for n up to $n = 32$ eight different independence systems were created for each value of n . For $n = 34$ only two, because the computation takes a lot of time.

To show the effectiveness of the measures to reduce the number of oracle visits, Figure 4 gives the actual number of visits scaled by, respectively, the bound $K + (n+1)M$, and the more accurate and smaller $K + \sum_1^M (l+1)$, with l indicating the level where the depth-first search did take off. For larger n the measures seem to be more effective.

The improvement by using the most recently found bitstring of L_i or L_j in the bitstring comparisons is displayed in Figure 5. This shows the fraction of all tested bitstrings that

Fig. 3. Probability distribution of random oracle.

Fig. 4. Oracle visits scaled by two bounds.

Fig. 5. Efficiency of utilizing the most recently used bitstring.

Fig. 6. Relative number of bitstring comparisons.

are rejected in a single shot. The variation in the results is due to differences in the generated independence systems, especially for $n = 34$ where an average over only two Monte Carlo simulations is used.

The previous result also influences the number of bitstring comparisons scaled with the factor $2^n(K + M)$ as presented in Figure 6. It can be seen that this factor, although not even a bound, is very conservative for large n .

The computational speed, expressed in bitstring comparisons per second, depends on the processor and the sizes of L_i , L_j , and L_p . The results for an SGI R10000 (200 MHz) 64-bit processor are displayed in Figure 7. The speed increases for increasing n . This is caused by the diminishing overhead for bookkeeping and multiset permutation generation, whose time is not accounted for separately.

An overall impression of the results of all performance improving measures is shown in Figure 8. This shows the computation time divided by the bound $2^n(K + M)$ as a function of n and scaled to 1 for $n = 34$. The decreasing nature is caused by

- a reduced number of bitstring comparisons (Figure 6), and
- more speedy computational speed due to less overhead (Figure 7).

From the last three figures the actual time spent can be deduced when one time point is known. For $n = 34$ the average time was $\approx 72,500$ seconds or ≈ 20 hours.

Fig. 7. Computational speed.

Fig. 8. Relative total computation time.

The amount of storage needed can be bounded by a number of 8-byte words equal to the number of oracle visits, and for these examples came to 2 MB for one of the independence systems at $n = 34$. This should be compared with the memory needed to store the complete Hasse diagram for this n : $8 \cdot 2^{34} = 137.4$ GB. The last size merits the use of the word “vast” in the name of the routine.

5.2. Engineering Application. The algorithm outlined above, and its corresponding implementation, has also withstood the test of a practical, large-scale, engineering design problem. The problem is the proper selection of sensors and actuators that enables the design of a control system that has to meet certain design specifications.

The selection of actuators and sensors (or inputs and outputs) has been studied before. In its basic formulation it is a combinatorial feasibility problem, where those combinations of inputs and outputs have to be selected that guarantee a specified level of performance. In general, combinations with a small number of devices are preferred. Exhaustive search can be used to generate all solutions. In practice, this is too costly for larger size problems due to the large number of combinations (2^n) and the cost of testing them. At least two approaches are available, which can be combined, to beat the curse of dimensionality. First, a simplified criterion can be used that allows (semi-) direct selection. This reduces the number of combinations to be checked. The criterion is often relatively easy to compute. A disadvantage is that this criterion is only loosely connected to the real design specifications. This approach therefore does not provide any guarantee that the results meet the specifications. Second, an optimization method can be used by adding an optimization criterion and using one of the approaches for local search in combinatorial, nonconvex, optimization problems like tabu search or branch and bound. A disadvantage is that only a single local optimum is found. Guarantees of how close this solution is to the global optimum are difficult to give. Also, a single solution is generated while it may be convenient to know other solutions that are almost as good. For more information about input/output selection methods see [10].

The problem we consider is a selection problem with $n_u = 7$ actuators and $n_y = 21$ sensors to choose from, so $n = 28$, for an active suspension control system of a tractor-semitrailer vehicle. The vehicle is modeled as a multibody system, with seven main

Fig. 9. Tractor-semitrailer with actuators $u_i, i = 1, \dots, 7$, and sensors (only indicated) $y_i, i = 1, \dots, 21$.

bodies: three axles, semitrailer, tractor chassis, engine, and cabin, having ten degrees-of-freedom in total. More information on the model, the model parameters, and the control problem formulation can be found in [11].

Force generators act between the bodies. Accelerations and relative positions of some of the bodies are sensed. The most fruitful combinations of actuators and sensors, using not too many of them and achieving a specified control objective, are to be selected. See Figure 9 for an overview of the system and the potential actuators and sensors. More details of this example can be found in [12].

The total number of combinations is $2^{28} \approx 268 \cdot 10^6$, so an exhaustive search is out of the question. With some assumptions on the behavior of the oracle this problem fits the setup of an independence system. The family of subsets $\mathcal{I}$ just gathers all actuator/sensor combinations that are not acceptable and $\mathcal{J}$ characterizes all acceptable ones. The power set $P(E)$ contains all possible combinations of actuators and sensors, $P = \mathcal{I} \cup \mathcal{J}$.

Results for this problem were obtained with VISOR. The oracle has been implemented with the help of the MATLAB program [7] and some of its toolboxes. Although the algorithm appeared to be rather efficient, the visit to the oracle is extremely involved. Only when K and M are not too large is the problem manageable, i.e., solvable in acceptable time.

A search that started with $l = n$ yields $K = 9$, $M = 58$, and 658 visits to the oracle. This represents a fraction $2.5 \cdot 10^{-6}$ of all elements of the power set. The number of bitstring comparisons was $\approx 733 \cdot 10^6$. The selection procedure was completed in exactly 15 hours computing time on an SGI R10000 processor, where the visits to the oracle took almost all the time and all other computations only 188 seconds. From this it is clear that the main assumption in the development and analysis of the algorithm, namely that the oracle is relatively expensive, is completely justified.

Table 1. Dependent sets with a single actuator and sensor.

Actuator	Sensor
Rear trailer suspension force	Suspension deflection trailer
Rear trailer suspension force	Acceleration trailer axle
Rear trailer suspension force	Acceleration rear trailer

The results of the selection are tabulated in Table 1. As can be seen from Table 1 there were three minimum dependent sets with one actuator and one sensor. Furthermore, there were five minimal dependent sets with one actuator and two sensors, and fifty with one actuator and three sensors.

From the values of K and M it may appear that a search starting from $l = 0$ would be more efficient. Indeed, the number of visits for this case is reduced to 274, representing only a fraction 10^{-6} of all combinations. Nevertheless, the time needed for the computation was almost twice as large, namely ≈ 28 hours, while other computations needed only 3.5 seconds, so the oracle took virtually all the time. The reason is that the fraction of tested sets that are termed independent is much larger now and those sets need significantly more time in the independency test than dependent sets. The sum paid to the oracle depends on its verdict, so only looking at the values of K and M is not sufficient, in general, to determine the fastest search direction.

6. Conclusions. A combinatorial optimization algorithm was proposed for arbitrary independence system problems. A routine was developed that enables a rigorous assessment of the structure of the independence system, without performing an exhaustive search and more efficiently than possible with previously published algorithms. The approach may still require an amount of work that is combinatorial in terms of the cardinality n of the set E , but it is polynomial in terms of this number and the cardinality of the sets that completely characterize the solution for practical problems with an expensive oracle and n not too large.

Application to a large-scale selection problem, for a medium-sized plant with 28 input/output devices to choose from, so $n = 28$, shows the algorithm and the corresponding implementation to be able to tackle this problem. The method has enough potential to make applying it to even larger problems a realistic option, especially when a cheaper oracle is available, because in this application the oracle was the limiting factor.

References

- [1] E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan, Generating all maximal independent sets: NP-hardness and polynomial-time algorithms, *SIAM J. Comput.*, vol. 9, pp. 558–565, Aug. 1980.
- [2] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. The MIT Electrical and Computer Science Series. Cambridge, MA: MIT Press, 1994. Eleventh printing.
- [3] D. P. Bovet and P. Crescenzi, *Introduction to the Theory of Complexity*. Prentice-Hall International Series in Computer Science. New York: Prentice-Hall, 1994.
- [4] C. H. Papadimitriou, *Computational Complexity*. Reading, MA: Addison-Wesley, 1994.

- [5] S. Tsukiyama, M. Ide, H. Ariyoshi, and I. Shirakawa, A new algorithm for generating all the maximal independent sets, *SIAM J. Comput.*, vol. 6, pp. 505–517, Sept. 1977.
- [6] B. de Jager and O. Toker, Complexity of input output selection, in *Proc. Internat. Symp. on the Mathematical Theory of Networks and Systems (MTNS 98)* (A. Beghi, L. Finesso, and G. Picci, eds.) (Padova), pp. 597–600. Padova: Il Poligrafo, July 1998.
- [7] MATLAB, *Using MATLAB, Version 5*. Natick, MA: The MathWorks, Dec. 1996.
- [8] J. F. Korsh and S. Lipschutz, Generating multiset permutations in constant time, *J. Algorithms*, vol. 25, pp. 321–335, Nov. 1997.
- [9] D. E. Knuth, *The Art of Computer Programming*, vol. 3. Reading, MA: Addison-Wesley, 1973.
- [10] M. van de Wal and B. de Jager, A review of methods for input/output selection, *Automatica*, vol. 37, pp. 487–510, Apr. 2001.
- [11] B. de Jager, Multiobjective suspension control problem, in *Proc. 34th IEEE Conf. on Decision and Control* (New Orleans, LA), vol. 4, pp. 3652–3657. Piscataway, NJ: IEEE, Dec. 1995.
- [12] B. de Jager, M. van de Wal, and R. Kamidi, Large-scale rigorous actuator and sensor selection, in *Proc. 37th IEEE Conf. on Decision and Control* (Tampa, FL), pp. 4314–4319. Piscataway, NJ: IEEE, Dec. 1998.