A Faster Branch-and-Bound Algorithm for the Test-Cover Problem Based on Set-Covering Techniques

TORSTEN FAHLE
INFORM GmbH
and
KARSTEN TIEMANN
University of Paderborn

The test-cover problem asks for the minimal number of tests needed to uniquely identify a disease, infection, etc. A collection of branch-and-bound algorithms was proposed by De Bontridder et al. [2002]. Based on their work, we introduce several improvements that are compatible with all techniques described in De Bontridder et al. [2002] and the more general setting of *weighted* test-cover problems. We present a faster data structure, cost-based variable fixing, and adapt well-known set-covering techniques, including Lagrangian relaxation and upper-bound heuristics. The resulting algorithm solves benchmark instances up to 10 times faster than the former approach and up to 100 times faster than a general MIP solver.

Categories and Subject Descriptors: G.2.1 [**Discrete Mathematics**]: Combinatorics—*Combinatorial algorithms*; G.2.3 [**Discrete Mathematics**]: Applications

General Terms: Algorithms, Experimentation, Performance

Additional Key Words and Phrases: Branch-and-bound, Lagrangian relaxation, set-cover problem, test-cover problem, variable fixing

This work was partly supported by the EU within the 6th Framework Programme under contract 001907 (DELIS) and by the DFG as part of the research cluster "Algorithms on Large and Complex Networks" (1126).
A previous version of this paper appeared as "An Improved Branch-and-Bound Algorithm for the Test Cover Problem" at the 4th International Workshop on Efficient and Experimental Algorithms (WEA 2005).
Authors' addresses: Torsten Fahle, INFORM GmbH, Pascalstraße 23, 52076 Aachen, Germany; torsten.fahle@inform-ac.com; Karsten Tiemann, International Graduate School of Dynamic Intelligent Systems, Faculty of Computer Science, Electrical Engineering and Mathematics, University of Paderborn, Fürstenallee 11, 33102 Paderborn, Germany; email: tiemann@uni-poderbom.dc.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2006 ACM 1084-6654/2006/0001-ART2.2 $5.00 DOI 10.1145/1187436.1216579 http://doi.acm.org 10.1145/1187436.1216579

1. INTRODUCTION

1.1 Problem Definition

We are given a set of m items and a set of tests $\mathbf{T} = \{T_1, \ldots, T_n\}$, $T_k \subseteq \{1, \ldots, m\}$, $k = 1, \ldots, n$. A test T_k is positive for an item i if $i \in T_k$, and negative if $i \notin T_k$. In general, we must use different tests to uniquely identify a given item because a single test may be positive for several items. We say that a test T_k *separates a pair of items* $\{i, j\}$, $1 \leq i < j \leq m$, if $|T_k \cap \{i, j\}| = 1$. Finally, a collection of tests $\mathcal{T} \subseteq \mathbf{T}$ is a *valid cover* if $\forall\ 1 \leq i < j \leq m : \exists\, T_k \in \mathcal{T} : |T_k \cap \{i, j\}| = 1$.

The *test-cover problem (TCP)* asks for a valid cover $\mathcal{T} \subseteq \mathbf{T}$ with minimum cardinality, i.e., for all valid covers $\mathcal{T}'$ it holds $|\mathcal{T}| \leq |\mathcal{T}'|$. The *weighted test-cover problem* is a canonical extension of the test-cover problem: Given $c : \mathbf{T} \to \mathbb{N}$, where $c(T)$ represents the cost for test T, we look for a valid cover $\mathcal{T}$ that is cheapest among all valid covers $\mathcal{T}'$: $\sum_{T \in \mathcal{T}} c(T) \leq \sum_{T \in \mathcal{T}'} c(T)$.

1.2 Related Work

Test-cover problems are of interest whenever the size or the cost of a set of tests should be minimized without losing any discrimination power. Early research on the test-cover problem was motivated by the identification of potato diseases [Lageweg et al. 1980]. In this application, a problem instance with 28 items and 63 tests was considered. Nowadays much bigger instances are of interest, e.g., for the protein identification by epitope recognition in computational biology [Halldórsson et al. 2001b]. There are many other real-life applications for TCPs in medical diagnostics, biology, pharmacy, fault detection, pattern recognition, or coding theory (see De Bontridder et al. [2002] and Moret and Shapiro [1985]).

The unweighted test-cover problem has been studied by several authors in recent years. Garey and Johnson [1979] showed its *NP*-hardness by a reduction from three-dimensional (3D) matching. More recently, De Bontridder et al. [2003] were able to show that the problem is *NP*-hard even if each test is positive for, at most, two items.

An approximation algorithm for the unweighted TCP (see e.g., Berman et al. [2004]) achieves an approximation ratio of $1+\ln m$. It is based on the well-known greedy algorithm for set-covering [Johnson 1974] and a natural reduction of test-cover to set-covering [Moret and Shapiro 1985].

The same reduction is used by Halldórsson et al. [2001a] in combination with approximation hardness results for set-covering to show that there is no $o(\log m)$ approximation algorithm for the test-cover problem unless $P = NP$.

Exact branch-and-bound algorithms for unweighted TCPs were early used and analyzed by Moret and Shapiro [1985]. Later, De Bontridder et al. [2002] introduced new branching rules and lower bounds for such algorithms. They compared the effectiveness of several pruning criteria, branching rules, and lower bounds.

1.3 Contribution

In this paper, we improve on the approach of De Bontridder et al. [2002] and generalize it to tackle *weighted* test-cover problems. We introduce and analyze several improvements that can be used to speed up a test-cover branch-and-bound algorithm:

1. We use a much faster data structure for the algorithm that is based on the idea to represent the given test-cover instance by a set-covering instance. All techniques discussed by De Bontridder et al. [2002] benefit from this efficient data structure and numerical results show a typical speed-up of 2 to 10 compared to solving the problem with the straightforward data structure. Using the new data structure, we compare the runtimes for different combinations of branching quality orderings and lower bounds on both unweighted and weighted benchmark sets.
2. For weighted-problem instances, we introduce techniques that are based on the cost values of the tests. We perform cost-based variable fixing during tree search that attempts to identify tests T_k that have to be (cannot be) part of any improving solution. Those tests are included (excluded) without branching. In so doing, runtime is reduced by a factor of 2 to 4 and the number of branch-and-bound nodes decreases by a factor of up to 6.
3. Lower bounds found by Lagrangian relaxation and upper bounds that are computed, based on such lower bounds, were successfully used for set-covering problems. We apply these techniques together with other popular approaches for solving set-covering problems in our test-cover branch-and-bound algorithm.

1.4 Organization of the Paper

We start our discussion by presenting the basic branch-and-bound approach of De Bontridder et al. [2002] in Section 2. After that, we show how to adapt it to get a new approach that solves TCPs, which are described by set-covering problems. In Section 3 we present variable fixing strategies. We discuss which techniques from set-covering algorithms might be helpful in the TCP context in Section 4. In Section 5, we present our numerical results. In particular, we compare the runtimes of the old, the new branch-and-bound approach, and thus of CPLEX, a commercial solver for linear programs. Finally, we conclude.

2. BASIC BRANCH-AND-BOUND APPROACHES

In this section, we describe the basic branch-and-bound approaches used to solve TCPs. We first briefly recall the approach used by De Bontridder et al. [2002], which works on the original TCP formulation. Thereafter, we describe how to adapt that approach to handle a set-covering formulation of the TCP. Interestingly, all methods used in the direct (TCP) approach can easily be adapted to work also in the indirect (set-covering) one. This allows us to also apply all structural methods known for the TCP in the indirect approach.

2.1 Solving the Test-Cover Problem Directly

In De Bontridder et al. [2002] various lower bounds and pruning criteria, as well as variable ordering heuristics for the TCP were described. Furthermore, a branch-and-bound framework was presented that allows to use any combination of a lower bound, a pruning criterion, and a variable-ordering heuristic.

Each branch-and-bound node is assigned a tuple $(\mathcal{T}, \mathcal{R})$. $\mathcal{T} \subseteq \mathbf{T}$ contains those tests that are included in the solution currently under construction. $\mathcal{R} \subseteq \mathbf{T}$ contains tests that have been discarded from the current solution. Thus, $\mathcal{T} \cap \mathcal{R} = \emptyset$. Initially, it holds $\mathcal{R} = \emptyset$, $\mathcal{T} = \emptyset$. Branch-and-bound decides on the remaining tests $T \in \mathbf{T} \setminus (\mathcal{T} \cup \mathcal{R})$.

A branch-and-bound node $(\mathcal{T}, \mathcal{R})$ potentially generates $l := |\mathbf{T} \setminus (\mathcal{T} \cup \mathcal{R})|$ child nodes. These nodes are ordered by some quality criterion, such that the most promising nodes (those with high quality) are processed first. After renumbering, we obtain the child nodes: $(\mathcal{T} \cup \{T_1\}, \mathcal{R})$, $(\mathcal{T} \cup \{T_2\}, \mathcal{R} \cup \{T_1\})$, $(\mathcal{T} \cup \{T_3\}, \mathcal{R} \cup \{T_1, T_2\}), \ldots, (\mathcal{T} \cup \{T_l\}, \mathcal{R} \cup \{T_1, \ldots, T_{l-1}\})$.

Nodes are considered in a depth-first search, i.e., node $(\mathcal{T} \cup \{T_1\}, \mathcal{R})$ and all its successors are explored prior to node $(\mathcal{T} \cup \{T_2\}, \mathcal{R} \cup \{T_1\})$.

2.1.1 *Quality Criteria.* In De Bontridder et al. [2002], four quality criteria are described for unweighted TCPs: Separation criterion D, information criterion ΔE, power criterion P, and least separated pair criterion S. We now give simple adaptions of these criteria to the case of weighted TCPs. Before that, define a partition of all m items into subsets $I_1^{\mathcal{T}}, I_2^{\mathcal{T}}, \ldots, I_t^{\mathcal{T}} \subset \{1, \ldots, m\}$ (also used in De Bontridder et al. [2002]). Given a partial cover $\mathcal{T} \subset \mathbf{T}$, a subset $I_j^{\mathcal{T}}$ is a largest subset containing items that are pairwise not separated by the current partial solution $\mathcal{T}$.

We begin our list of quality criteria with the separation criterion D that was introduced by Moret and Shapiro [1985]. It calculates the cost per additional separated pair of items that test T causes when it is added to $\mathcal{T}$ as

$$D(T|\mathcal{T}) := \frac{1}{c(T)} \sum_{h=1}^{t} \left|T \cap I_h^{\mathcal{T}}\right| \cdot \left|I_h^{\mathcal{T}} \setminus T\right|$$

The information criterion ΔE was originally used in Rescigno and Maccacaro [1961] and Chang et al. [1970] and can be calculated using the information value $E(\mathcal{T})$ of a test set $\mathcal{T}$:

$$\Delta E(T|\mathcal{T}) := \frac{E(\mathcal{T} \cup T) - E(\mathcal{T})}{c(T)}$$

where

$$E(\mathcal{T}) := \log_2(m) - \frac{1}{m} \sum_{h=1}^{t} \left|I_h^{\mathcal{T}}\right| \cdot \log_2 \left|I_h^{\mathcal{T}}\right|.$$

In De Bontridder et al. [2002], the power criterion P was introduced. Here, the power of a test T is given by:

$$P(T|\mathcal{T}) := \frac{1}{c(T)} \sum_{h=1}^{t} \min\left\{\left|T \cap I_h^{\mathcal{T}}\right|, \left|I_h^{\mathcal{T}} \setminus T\right|\right\}$$

Finally, we define the least separated pair criterion S used in, e.g., Moret and Shapiro [1985], Payne [1981], and Rypka et al. [1967, 1978]. First, we search for a pair of items $\{i, j\}$ that is separated by the least remaining tests. We then separate the remaining tests into those separating the item pair and those not separating it by:

$$S(T|\mathcal{T}) := \begin{cases} 1 & \text{if } |T \cap \{i, j\}| = 1 \\ 0 & \text{else} \end{cases}$$

Criterion S can be used alone and follows the idea of assigning most critical items early in the search. In our work, as well as in De Bontridder et al. [2002], S is only used in combination with some other criterion (denoted as $S(C)$, where $C \in \{D, \Delta E, P\}$). There, all $T \in \mathbf{T} \setminus (\mathcal{T} \cup \mathcal{R})$ are grouped according to S: $\{T \mid S(T|\mathcal{T}) = 1\}$ and $\{T \mid S(T|\mathcal{T}) = 0\}$. Within each of these groups, tests are sorted according to criterion C.

2.1.2 *Lower Bounds.* Several different lower bounds have been proposed in the literature. An overview in De Bontridder et al. [2002] presented four called $L_1, \ldots, L_4$. All four yield a lower bound on the additional number of tests needed to separate all pairs of items (in addition to the already selected tests in $\mathcal{T}$). Our work uses these bounds, except for L_3, as that one was shown to be too slow in De Bontridder et al. [2002].

The first bound L_1 is based on the property that at least $\lceil \log_2 k \rceil$ tests are necessary to separate k items that have not been separated so far. This results in the lower bound

$$L_1(\mathcal{T}) := \lceil \log_2 \big(\max_{h=1,\ldots,t} |I_h^{\mathcal{T}}| \big) \rceil$$

Lower-bound L_2 requires the use of the power criterion P and some instance-independent values $F(j, i)$ that can be found using dynamic programming. $F(j, i)$, $j = 1, \ldots, m$, $i = 1, \ldots, n$ denotes the minimal power sum needed to separate j items by i tests. Bound L_2 is the minimum number of tests, such that their power reaches or exceeds the required power of all unseparated pairs of items. If P^* denotes the unweighted version of the previously described power criterion P, i.e.,

$$P^*(T|\mathcal{T}) := \sum_{h=1}^{t} \min \big\{ |T \cap I_h^{\mathcal{T}}|, |I_h^{\mathcal{T}} \setminus T| \big\}$$

then L_2 reads as

$$L_2(\mathcal{T}, \mathcal{R}) := \min \Big\{ k \mid \sum_{j=1}^{k} P^*(T_{\sigma(j)}|\mathcal{T}) \geq \sum_{h=1}^{t} F\big(|I_h^{\mathcal{T}}|, k\big) \Big\}$$

where σ is a permutation, such that

$$P^*(T_{\sigma(1)}|\mathcal{T}) \geq P^*(T_{\sigma(2)}|\mathcal{T}) \geq \cdots \geq P^*(T_{\sigma(l)}|\mathcal{T})$$

and

$$\{T_{\sigma(1)}, \ldots, T_{\sigma(l)}\} = \mathbf{T} \setminus (\mathcal{T} \cup \mathcal{R})$$

Finally, L_4 is based on information values of test sets. Here, one is looking for the minimal number of tests to raise the information value from $E(\mathcal{T})$ to

$\log_2(m)$. Because this work mainly focuses on algorithmic improvements, we refer to De Bontridder et al. [2002] for more details on L_2 and L_4. An interesting lemma in that paper states that the lower bound L_2 is always at least as good as L_1: $L_2(\mathcal{T}, \mathcal{R}) \geq L_1(\mathcal{T})$. Unfortunately, the computation of L_2 is more time-consuming, because it requires the computation of test power values.

2.1.3 *Pruning.* Pruning, that is detecting and discarding useless parts of the search tree, is based on four different criteria. Let L be a lower bound on the additional number of tests needed to separate all pairs of items. We prune

- *PC*1, if according to the lower bound, more tests are needed than are available, i.e.,

$$\text{if } L > |\mathbf{T}| - |\mathcal{T}| - |\mathcal{R}|, \text{ then prune.}$$

- *PC*2, if the minimal number of tests needed to construct a solution is not smaller than the number of tests, U, in the best solution found so far, i.e.,

$$\text{if } |\mathcal{T}| + L \geq U, \text{ then prune.}$$

- *PC*2*w*. (PC2 for weighted case, U is objective value of the best solution found so far)

$$\text{if } \sum_{T \in \mathcal{T}} c(T) + \sum_{i=1}^{L} c(T_i) \geq U, \text{ then prune.}$$

  (PC2w requires the remaining tests $T_i \in \mathbf{T} \setminus (\mathcal{T} \cup \mathcal{R})$ to be sorted according to increasing costs. Such an ordering can be determined in a preprocessing step.)

- *PC*3, if there is an unseparated pair of items that cannot be separated by any of the remaining tests in $\mathbf{T} \setminus (\mathcal{T} \cup \mathcal{R})$.

2.2 Solving the Test-Cover Problem Indirectly

In this section, we propose to solve the test-cover problem indirectly. That is, instead of working on the original formulation, we transform the TCP into a set-covering problem. Afterward, a branch-and-bound algorithm works on this set-covering problem representing the original TCP.

2.2.1 *Relation between Test-Cover and Set-Covering Problems.* A *set-covering problem (SCP)* consists of n subsets $\{S_1, \ldots, S_n\} = \mathbf{S}$, $S_i \subseteq \{1, \ldots, \ell\}$. Furthermore, there is a cost function $c : \mathbf{S} \to \mathbb{N}$, where $c(S_i)$ represents the cost of the subset S_i. The SCP asks for a collection $\mathcal{S} \subseteq \mathbf{S}$, such that $\{1, \ldots, \ell\}$ is covered at minimal cost. To be more precise:

1. $\cup_{S \in \mathcal{S}} S = \{1, \ldots, \ell\}$ and
2. for all $\mathcal{S}'$ such that $\cup_{S \in \mathcal{S}'} S = \{1, \ldots, \ell\}$ we have $\sum_{S \in \mathcal{S}} c(S) \leq \sum_{S \in \mathcal{S}'} c(S)$.

SCPs are *NP*-hard [Garey and Johnson 1979]. Given their importance in flight scheduling, crew rostering, etc., they have been subject of intensive research during the last decades, (see the survey in [Caprara et al. 2001]).

A TCP instance can be interpreted as a SCP instance (see Moret and Shapiro [1985]) by considering all pairs of items and asking for a coverage of all pairs

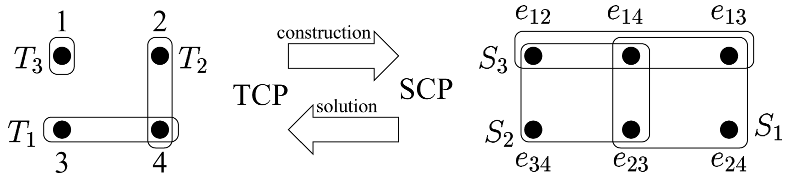

Fig. 1. Example: A test-cover instance (left) and the corresponding set-covering instance (right).

by the tests. Let $(m, \mathbf{T} = \{T_1, \dots, T_n\}, c)$ be a TCP instance. A SCP instance $(\ell, \mathbf{S} = \{S_1, \dots, S_n\}, c')$ is constructed by (see Figure 1, for example)

1. using all pairs of items: $\{1, \dots, \ell\} \leftarrow \{e_{ij} \mid 1 \leq i < j \leq m\}$,
2. setting S_k to contain all pairs of items separated by test T_k: $S_k \leftarrow \{e_{ij} \mid |T_k \cap \{i, j\}| = 1,\ 1 \leq i < j \leq m\}$, $k = 1, \dots, n$, and by
3. keeping the original costs by: $c'(S_k) := c(T_k),\ \forall\, k \in \{1, \dots, n\}$.

It is easy to see that a SCP solution $\mathcal{S} \subseteq \mathbf{S}$ defines a solution $\mathcal{T} \subseteq \mathbf{T}$ for the TCP. By construction, the objective value of both solutions is identical. Thus, we can solve a TCP with n tests and m items by solving the corresponding SCP having n subsets and $\Theta(m^2)$ items.

2.2.2 *Using the Transformation for a New Indirect Approach.* In our indirect approach, we work on the SCP interpretation of the given TCP. Because this SCP still represents the original TCP, all methods introduced in Section 2.1 for the direct approach can be easily adapted to handle the SCP formulation of the TCP.

The transformation to a SCP squares the number of "objects" in the problem, because rather than separating m items in the TCP, we have to cover $\Theta(m^2)$ pairs of items in the SCP. On the other hand, most techniques described before have to spend time $\Omega(m^2)$ anyway (e.g., they need to find out which pairs of items are not covered, etc.). That is, the direct formulation is more space efficient, but does not save computing time. Even worse, whereas in the SCP formulation, we can delete a pair of items from consideration as soon as we have covered it, we cannot do similarly in the original formulation.

In the original formulation, an item i can only be removed from the list of active items, when it is separated from all other items j. In the best case, this happens after $m - i$ steps (after separating $\{i, i+1\}, \dots, \{i, m\}$, we can discard item i). In the worst case, however, $\frac{1}{2}(m^2 - 2m) + 1$ steps are necessary before, in the direct formulation, a test can be discarded from consideration (see Figure 2).

If, in contrast, the new indirect approach is used, the number of set-covering items decreases each time a test is added to $\mathcal{T}$. In every node of the branch-and-bound tree, the SCP to consider contains only the set-covering items that are still uncovered. These items represent exactly the test-cover item pairs that are so far unseparated. Interpreting a TCP as a SCP, therefore, gives a natural way of finding a more efficient data structure for pairs of items.

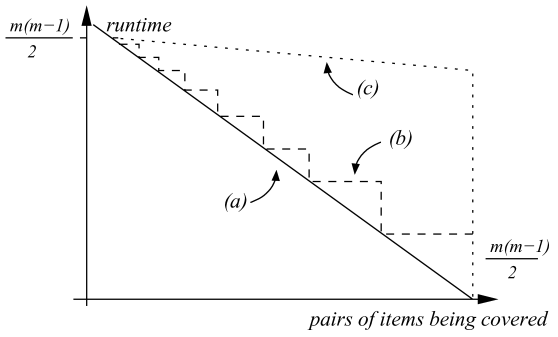

Fig. 2. Schematic view on the runtime (y axis) when working on a branch-and-bound node with a certain number of separated pairs of items (x axis). (a) is the runtime for the indirect approach—runtime is proportional to the number of unseparated pairs of items; (b) and (c) give examples for best-and worst-case runtimes when using the direct formulation of the problem.

3. VARIABLE FIXING

Variable fixing aims in identifying tests that have to be part of a feasible or optimal solution. It is one of the building blocks in constraint programming and it is frequently used in presolving for mixed-integer programming. We first explain in Section 3.1 the fixing of essential tests, a technique also used in De Bontridder et al. [2002]. Thereafter, we introduce a counterpart in Section 3.2: fixings that aim in removing redundant tests. In Section 3.3, we introduce another new variable-fixing method based on costs. The new fixings can reduce runtime by a factor of 2 to 4 (see Section 5.4).

3.1 Fixings Based on Feasibility

Before starting a branch-and-bound algorithm, it is helpful to identify *essential tests*. A test $T \in \mathbf{T}$ is essential if there is a pair $\{i, j\}$ of items that is only separated by T. In this case, T must be part of any feasible solution to the TCP and it can be added to $\mathcal{T}$. As described in De Bontridder et al. [2002], searching for essential tests can be performed in time $O(m^2n)$. In their approach, essential tests are only applied in the root node.

In our approach, we check for essential tests in all branch-and-bound nodes. During branching, some other test as well might become the only remaining test that separates a certain pair of items. It is thus justified to check for essential tests not only in the root node. Using an incremental algorithm, we can do so in time, proportional to the number of pairs not so far separated: Let $\mu_{\{i,j\}}$ denote the number of tests that separate item pair $\{i, j\}$. In an initial preprocessing step, we initialize $\mu_{\{i,j\}}$ in time $O(m^2n)$. For any branching decision or any fixing of tests, we decrement those entries of μ that correspond to pairs of items separated by the tests in question. Hence, in each branch-and-bound node, we can simply check whether only one test remains that separates a certain pair of items, and we can include that test into the current solution. The number of pairs of items not separated decreases with the depth of the search tree and is always $O(m^2)$. Thus, fixing essential tests in a branch-and-bound node requires time $O(m^2)$ per node, which is the time already needed by most other techniques described in Section 2.

3.2 Fixings Based on Redundancy Information

Similarly, we can introduce the notion of *useless tests*. A test $T \in \mathbf{T}$ is considered *useless* at some node $(\mathcal{T}, \mathcal{R})$ in the search tree if T does not separate any unseparated pair of items in that node, i.e.,

$$\forall h = 1, \ldots, t \ : \quad T \cap I_h^{\mathcal{T}} = \emptyset \quad \vee \quad I_h^{\mathcal{T}} \setminus T = \emptyset \tag{1}$$

Trying to include or exclude a useless test during branch-and-bound search is redundant work and, hence, we prefer to detect and eliminate useless tests. It is easy to see from (1) that criteria $\Delta E(T|\mathcal{T})$, $D(T|\mathcal{T})$, and $P(T|\mathcal{T})$ will return a quality value of zero if, and only if, a test T is useless. Therefore, detecting these tests can be done in time $O(1)$ per test after the evaluation with one of these pruning criteria in the current branch-and-bound node has been done. In the experiments, we will always remove useless tests during search. It should be noted that criterion $S(T|\mathcal{T})$, if used alone, might return a quality value of zero even if T is not useless (see the definition in Section 2.1.1).

Another source of redundancy are *duplicated tests*, two tests $T_1, T_2 \in \mathbf{T}$ that separate exactly the same items in a branch-and-bound node $(\mathcal{T}, \mathcal{R})$:

$$\forall h = 1, \ldots, t, \ \forall i, j \in I_h^{\mathcal{T}} \ : \quad |T_1 \cap \{i, j\}| = |T_2 \cap \{i, j\}| \tag{2}$$

Obviously, only one of the two tests is needed and, if costs of these differ, it suffices to try only the cheaper one during search. Detecting duplicated tests requires to check Eq. (2) for each pair of tests T_1, T_2 that received the same evaluation by the current quality criterion, if costs are neglected. Thus, the runtime is $O(n^2 m^2)$ per node. This runtime is rather high and experiments indicate that the few reductions in branch-and-bound nodes cannot overwhelm the higher runtime.

3.3 Fixings Based on Costs

In addition, we can include or exclude tests that have to be or cannot be part of an improving solution. Let L denote a lower bound on the number of tests required to form a valid test-cover in the current branch-and-bound node and let U denote the value of the incumbent upper bound. We order the remaining $k := |\mathbf{T} \setminus (\mathcal{T} \cup \mathcal{R})|$ tests in $\mathbf{T} \setminus (\mathcal{T} \cup \mathcal{R}) =: \{T_1, \ldots, T_k\}$ according to increasing costs.

If $k < L$ pruning criterion, PC1 already prunes the current search tree. We also prune, if the cost of all tests in $\mathcal{T}$, plus the costs of the L cheapest tests in $\mathbf{T} \setminus (\mathcal{T} \cup \mathcal{R})$, exceeds the current upper bound U (PC2w).

Otherwise, we can apply the following variable fixing: If replacing one of the L cheapest tests T_i by test T_{L+1} results in a value not smaller than the incumbent solution, test T_i is essential for any improving solution, and thus we can fix it:

$$\begin{array}{ll} \forall\, 1 \le i \le L : & \\ \quad \text{if} & \sum_{T \in \mathcal{T}} c(T) + \sum_{j=1}^{L} c(T_j) - c(T_i) + c(T_{L+1}) \ge U \\ \quad \text{then} & \mathcal{T} := \mathcal{T} \cup \{T_i\} \end{array} \tag{3}$$

Vice versa, if already replacing test T_L by some more expensive test results in a value not smaller than the incumbent solution, we can safely discard the latter test:

$$\begin{array}{ll} \forall\, L+1 \le i \le k : & \\ \quad \text{if} & (\sum_{T \in \mathcal{T}} c(T) + \sum_{j=1}^{L} c(T_j) - c(T_L) + c(T_i) \ge U \\ \quad \text{then} & \mathcal{R} := \mathcal{R} \cup \{T_i\} \end{array} \tag{4}$$

Notice that these checks are only useful for the weighted case as in the unweighted case no change in the cost structure occurs when exchanging tests.

4. APPLYING GENERAL SCP TECHNIQUES

In the previous sections, we have applied tailored techniques to the TCP. In this section, we discuss techniques that have been proposed to tackle general SCPs. After a transformation to SCP, these techniques can be applied to the TCP as well.

The most popular approaches for solving SCP are based on Lagrangian relaxation (see e.g., the survey by Caprara et al. [2001]). The key ingredients of these approaches are lower bounds found by Lagrangian relaxation, upper bounds that typically construct solutions from given lower bounds, and (cost-based) fixing techniques.

On the one hand, a transformation to SCP obviously allows to apply a much wider range of well-known techniques. On the other hand, these techniques have been developed for general SCPs and maybe are not as efficient as the specialized techniques discussed in previous sections. We will shortly recall these SCP techniques here and discuss their potential impact in a TCP scenario.

4.1 Lower Bounds for SCP

4.1.1 *Lower Bounds for SCP via Lagrangian Relaxation.* Lagrangian relaxation is a general technique that provides lower bounds on linear integer programs (IP) by removing (parts of) the constraints and introducing them as a penalty term into the objective. An optimal solution of the resulting IP then provides a lower bound on the original one. We refer to Fischer [1981] or Lemaréchal [2001] for further information on Lagrangian relaxation.

Beasley [1990] proposed the following Lagrangian relaxation for the SCP, which we also applied in our experiments. Given the IP formulation of the SCP:

$$\begin{array}{rl} \min & \sum_{i=1}^{n} c_i x_i \\ \text{s.t.} & Ax \ge 1 \\ & x \in \{0,1\}^n \end{array} \tag{5}$$

it removes all constraints $Ax \geq 1$ and introduce them into the objective as a penalty term.

$$lb(\lambda) = \min \sum_{i=1}^{n} c_i x_i + \lambda(1 - Ax), \qquad \text{s.t. } x \in \{0,1\}^n \tag{6}$$

$$= \min \sum_{i=1}^{n} \underbrace{\left(c_i - \sum_{j=1}^{q} \lambda_j A_{ij}\right)}_{=:\widetilde{C}_i} x_i + \sum_{j=1}^{q} \lambda_j, \qquad \text{s.t. } x \in \{0,1\}^n \tag{7}$$

A *Lagrangian multiplier* $\lambda = (\lambda_1, \ldots, \lambda_q)^T \in \mathbb{R}^q_{\geq 0}$, $q = \frac{m(m+1)}{2}$, is used to adjust the impact of the penalty term. Given some λ, Eq. (7) can be solved by simply checking the sign of x_i's coefficient in order to obtain a solution $\overline{x} = (\overline{x}_1, \ldots, \overline{x}_n)$:

$$\overline{x}_i := \begin{cases} 0, & \text{if } \widetilde{C}_i < 0 \\ 1, & \text{if } \widetilde{C}_i \geq 0 \end{cases}, \quad i = 1, \ldots, n \tag{8}$$

Any $\lambda \in \mathbb{R}^q_{\geq 0}$ gives a lower bound on the original problem. A tightest bound is found when solving the *Lagrangian dual problem*

$$\max_{\lambda \geq 0^q} lb(\lambda) \tag{9}$$

Since $lb(\lambda)$ is a convex function, *subgradient methods* can be used to solve Eq. (9). The idea of subgradient methods is to start with some λ and to move that λ into the direction of the steepest ascent. Subgradients are used to guide the search, since they point to the right direction. A subgradient $g = (g_1, \ldots, g_q)^T \in \mathbb{R}^q$ is defined as

$$g_i = 1 - \sum_{j=1}^{n} A_{ij} \overline{x}_j, \qquad i = 1, \ldots, q \tag{10}$$

The initial multiplier $\lambda^{(0)}$ is chosen as the largest relative cost per covered pair. Let $\mathcal{A}$ denote the set of all pairs of items covered in the current branch-and-bound node and let $\mathcal{E}$ denote the set of pairs of items not covered.

$$\lambda_e^{(0)} = \max_k \left\{ \frac{c(S_k)}{|S_k \setminus \mathcal{A}|} \; ; \; e \in S_k \right\}, \quad \forall e \in \mathcal{E} \tag{11}$$

Held and Karp [1971] proposed the following update step:

$$\lambda_i^{(k+1)} \leftarrow \max \left\{ 0, \; \lambda_i^{(k)} + \alpha \frac{(ub - lb) g_i}{\|g\|^2} \right\}, \quad i = 1, \ldots, q \tag{12}$$

where ub, lb denote incumbent upper and lower bound, respectively. Using Eq. (12), it is easy to calculate a sequence $\lambda^{(1)}, \lambda^{(2)}, \ldots$ of Lagrangian multipliers that converges toward an optimal solution to Eq. (9). The gap between upper and lower bound of the original problem adapts the update step to the progress of the optimization algorithm. The parameter α is halved if ub and lb have not changed for longer time. Whenever $\|g\| = 0$, or α is smaller than some predefined constant, the iterative generation of new multipliers (Eq. 12) stops and the best solution to Eq. (9) is returned as a lower bound.

The values determined above can also be used for some simple variable fixing technique: Assume subset S_k is not used in the lower bound (i.e., $x_k = 0$). If forcing S_k to be part of a solution will result in a high objective value, we can remove that subset from consideration in the current part of the search tree:

$$\forall\, k = 1, \ldots, n: \quad (x_k = 0 \text{ and } lb + \widetilde{C_k} > ub) \Rightarrow S_k \text{ can be neglected} \tag{13}$$

Lower bounds found by Lagrangian relaxation are typically rather tight. For the SCP, it can be shown that they are as tight as the LP relaxation of Eq. (5). This accuracy is paid for by rather high runtimes. It often requires between 200 and 1000 iterations before a good λ is found and a tight solution can be calculated.

4.1.2 *Lower Bounds via Partial Costs.* A less expensive lower bound can be obtained when calculating the minimal partial costs that are necessary to cover a certain pair of items. Again, let $\mathcal{A}$ denote the set of all pairs of items covered in the current branch-and-bound node and let $\mathcal{E}$ denote the set of pairs of items not covered. Let ρ_e denote the smallest cost that is required to cover element $e \in \mathcal{E}$, i.e.,

$$\rho_e = \min_k \left\{ \frac{c(S_k)}{|S_k \setminus \mathcal{A}|} \quad e \in S_k \text{ and } S_k \in \mathbf{S} \right\}, \quad \forall e \in \mathcal{E} \tag{14}$$

Summing over all ρ_e results in a lower bound. Notice, that the relaxation used in this bound is that we can pick an individual element e from its cheapest set S_k, rather than taking the entire set.

4.2 Variable Fixing for SCPs

Variable-fixing techniques for the SCP are often used in a preprocessing step and/or during search to tighten the SCP in the current branch-and-bound node. They can be derived from the more general case of LP presolving techniques (see Andersen and Andersen [1995]) and typically fall into one of the following four categories. Let $S_a, S_b \subseteq \{1, \ldots, k\}$ be two sets.

- $F1$. If S_a is the only set to cover a certain item, S_a must be part of every solution.
- $F2$. If S_a is empty, it can be removed.
- $F3$. If S_a and S_b are identical, the more expensive one can be removed.
- $F4$. If S_a is contained in S_b and S_a is at least as expensive as S_b than S_a can be removed.

Removal of empty or duplicated sets (techniques $F2$, $F3$) have already been discussed for the special case of TCPs in Section 3.2. If the SCP is stemming from a TCP via the transformation given in Section 2.2.1, then $F1$ corresponds to identifying essential tests and is thus already covered by the test in Section 3.1. Applying $F4$ to some SCP that originates from a TCP will have almost no effect, as shown in the next lemma:

Table I. Possible Combinations of Covered Pairs

	Included in Test	Not Included in Test	Included in Set	Not Included in Set	
	i, p, q	j	$e_{j,p}, e_{j,q}$	$e_{i,p}, e_{i,q}$	
	j, p, q	i	$e_{i,p}, e_{i,q}$	$e_{j,p}, e_{j,q}$	
T_1	i	j, p, q	$e_{i,p}, e_{i,q}$	$e_{j,p}, e_{j,q}$	S_1
	j	i, p, q	$e_{j,p}, e_{j,q}$	$e_{i,p}, e_{i,q}$	
	i, p	j, q	$e_{i,q}, e_{j,p}$	$e_{i,p}, e_{j,q}$	
	i, q	j, p	$e_{i,p}, e_{j,q}$	$e_{i,q}, e_{j,p}$	
T_2	j, p	i, q	$e_{j,q}, e_{i,p}$	$e_{j,p}, e_{i,q}$	S_2
	j, q	i, p	$e_{j,p}, e_{i,q}$	$e_{j,q}, e_{i,p}$	

LEMMA 4.1. *Let $T_1, T_2 \in \mathbf{T}$ be two tests and denote by $S_1, S_2 \in \mathbf{S}$ the two resulting sets after transforming TCP to SCP. Then it holds:*

$$S_1 \subseteq S_2 \iff (S_1 = \emptyset \lor \quad S_1 = S_2)$$

PROOF. The direction "$\Leftarrow$" is obvious. Consider now "$\Rightarrow$": Let $S_1 \subseteq S_2$. We prove by contradiction, i.e., we assume $S_1 \neq \emptyset$ and $S_1 \neq S_2$. Because we have $\emptyset \subset S_1 \subset S_2$, there is an element $e_{i,j} \in S_1$ that is also contained in S_2 and we have an element $e_{p,q} \in S_2$ that is not contained in S_1.

By construction (see Section 2.2.1) T_1 must either contain i or j. The same holds for T_2. Concerning p, q test T_1 must either contain both, or none of them and T_2 must contain exactly one of them. The left-hand side of Table I summarizes all possible combinations.

Now we consider the elements $e_{i,p}, e_{i,q}, e_{j,p}, e_{j,q}$. As can be seen in the right-hand side of Table I, S_1 either contains both $e_{i,\cdot}$ or both $e_{j,\cdot}$ elements. On the other hand, S_2 does always exclude one $e_{i,\cdot}$ and one $e_{j,\cdot}$ element. Thus, in all cases there is one element in S_1 that is not included in S_2, a contradiction to $S_1 \subseteq S_2$. □

Consequently, as $F4$ will only detect sets that will also be detected by $F2$ or $F3$ applying fixings based on testing subsets ($F4$) does not pay off, if the SCP results from a TCP.

4.3 Upper-Bound Computation

Equation (8) provides a vector x that corresponds to the optimal solution of the Lagrangian relaxation. x is a lower bound on the original SCP. However, x is usually not a feasible solution, since some pairs of items may not be covered.

4.3.1 *Upper Bounds from Lagrangian Bounds.* Beasley [1990] uses the two-phase approach that was also applied in our work (algorithm given in Figure 3). It starts by covering all missing pairs by a cheapest set S_k. If, after that phase, some pairs of items are overcovered it tries to remove those sets that are redundant. In the experiments with Lagrangian Relaxation, we call that algorithm whenever Eq. (7) improves the incumbent lower bound.

4.3.2 *Initial Upper Bound.* Obviously, the same algorithm can also be used for finding an initial upper bound prior to any branching and without applying Lagrangian techniques. By calling the algorithm for $x = 0^n$, it constructs a feasible solution x' of costs $ub = \sum_{i=1}^{n} c_i x'_i$.

```
$x' \leftarrow x$;   $\mathcal{S}' \leftarrow \{S_k \mid x_k = 1,\ k = 1, \ldots, n\}$
/* Phase 1: Cover all pairs of items by including sets */
while ($\exists$ a pair of items $\{i, j\}$ that is not covered by $\mathcal{S}'$)
    $l \leftarrow$ index(argmin$\{c(S_k) \mid S_k$ covers $\{i, j\},\ k = 1, \ldots, n\}$)
    $x'_l \leftarrow 1$;   $\mathcal{S}' \leftarrow \mathcal{S}' \cup \{S_l\}$
/* Phase 2: If pairs are over-covered: Remove redundant sets */
for all ($S_k \in \mathcal{S}'$ in order of decreasing costs)
    if ($\mathcal{S}' \setminus \{S_k\}$ covers all pairs $\{i, j\}$)
        then $x'_k \leftarrow 0$;   $\mathcal{S}' \leftarrow \mathcal{S}' \setminus \{S_k\}$
return $x'$
```

Fig. 3. Constructing an upper bound.

4.3.3 *Improving Solutions Found during Branch-and-Bound.*

Phase 2 of the algorithm given in Figure 3 can also be applied to solutions found during branch-and-bound search. Depending on the selected lower bound and pruning criteria, up to 20–30 solutions of decreasing costs are found by the branch-and-bound approach before an optimal one is detected. These intermediate solutions can be improved by removing sets containing only overcovered pairs of items.

5. NUMERICAL RESULTS

In De Bontridder et al. [2002], 140 benchmark sets were used to experimentally evaluate different branch-and-bound algorithms for the unweighted TCP. These benchmark sets were constructed randomly and they differ with respect to the number of items m, the number of tests n, and the probability p for a test to contain a certain item (E[$i \in T_k$] $= p$). There are ten different instances for each of the following (m, n, p)-combinations: (49, 25, {1/4, 1/2}), (24, 50, {1/10, 1/4, 1/2}), (49, 50, {1/10, 1/4, 1/2}), (99, 50, {1/10, 1/4, 1/2}), (49, 100, {1/10, 1/4, 1/2}). We use the same sets for the unweighted case and, thus, our results can be compared to those found in the earlier work on the TCP. For the weighted case, these instances were extended by assigning a cost value to each test uniformly at random from the interval $\{1, \ldots, n\}$.

In their paper, De Bontridder et al. note that they have not used *"clever data structures for storing and retrieving information"* but that a *"significant speedup"* could be expected from these. Therefore, in addition to the techniques used in De Bontridder et al. [2002] both approaches (direct and indirect) store information needed in every branch-and-bound node. We update this information incrementally rather than calculating it from scratch in each node. These data include the assignment of items to the sets $I_1^{\mathcal{T}}, I_2^{\mathcal{T}}, \ldots, I_t^{\mathcal{T}}$, which are needed for branching based on quality criterion D, ΔE, and P, as well as for lower bounds L_1, L_2, and L_4. Furthermore, for each pair of items $\{i, j\}$ not separated so far, we store the number of tests in $\mathbf{T} \setminus (\mathcal{T} \cup \mathcal{R})$ that separate $\{i, j\}$. The latter information is needed for PC3, for the least-separated pair criterion S, and for the fixing of essential tests that we do in every branch-and-bound node. Because the implementation in De Bontridder et al. [2002] always recalculates the just-mentioned values, it is justified to assume that our implementation of the direct approach is already faster than the original implementation used by the authors of De Bontridder et al. [2002].

In the following sections, we first compare in Sections 5.1–5.3 the three different approaches to solve test-cover problems, namely, (a) solving the TCP directly, (b) solving the TCP indirectly by transforming it to a SCP (see Section 2.2.1), and (c) solving its SCP formulation given in Eq. (5) via CPLEX. For all weighted benchmark sets, we use cost fixing (impact studied further in Section 5.4), but we do not use lower bounds based on Lagrangian relaxation or partial cost (considered in Section. 5.5). Furthermore, in Sections 5.1–5.3, we do not use upper-bound heuristics (explored in Section 5.6).

All figures given show average runtimes in seconds and average numbers of branch-and-bound nodes, respectively, for the ten instances in each (m, n, p) combination. Notice that some diagrams use a logarithmic scale. Lines connecting measure points are used for better readability and do not indicate intermediate results. A tuple (C, L_i) denotes a branch-and-bound algorithm using the quality criterion $C \in \{D, \Delta E, P, S(D), S(\Delta E), S(P)\}$ and the lower bound $L_i, i \in \{1, 2, 4\}$.

The experimental evaluation was performed on a Pentium 4 (1.7-GHz) running Linux (kernel version 2.4). The algorithms were coded in C++ and compiled by gcc 2.95.3 using full optimization. In the comparison, we used Ilog CPLEX 7.5 with default settings.

5.1 General Observations for all Benchmark Sets

We have studied different combinations of pruning criteria, branching quality orderings and lower bounds for both branch-and-bound approaches on unweighted and weighted benchmark sets. There are two general observations that hold for both approaches and all (unweighted and weighted) benchmark instances. The first observation is that an algorithm using a quality criterion $C \in \{D, \Delta E, P\}$ is by a factor of 1.5 to 300 slower than the same algorithm using $S(C)$ instead. (Where $S(C)$ denotes the least separated pair criterion S in combination with C, as introduced in Section 2.1.1.) The second general observation deals with the lower bounds. An algorithm using the lower bound L_4 is accelerated by a factor of 10 to 1000 (unweighted instances) or by a factor of 3 to 15 (weighted instances) by using L_1 or L_2 instead.

As an example, Figure 4 shows that algorithms using the S criterion and one of the lower bounds L_1, L_2 should be preferred for the indirect approach on weighted instances containing 49 items and 50 tests.

5.2 Unweighted Benchmark Sets

For the direct approach on unweighted instances, a combination of quality criterion $S(D)$ and lower bound L_1 or L_2 was found to be most efficient (this corresponds to the findings in De Bontridder et al. [2002]). For the indirect approach using L_1, rather than L_2, turned out to be most efficient (see Figure 5). Thus, we use the variant $(S(D), L_1)$ in most experiments. On the instances $(49, 100, 1/10)$, however, the algorithm $(S(D), L_2)$ is about six times faster than the $(S(D), L_1)$ algorithm. For a detailed study on the impact of different pruning criteria, branching rules, and quality criteria, we refer to the work in

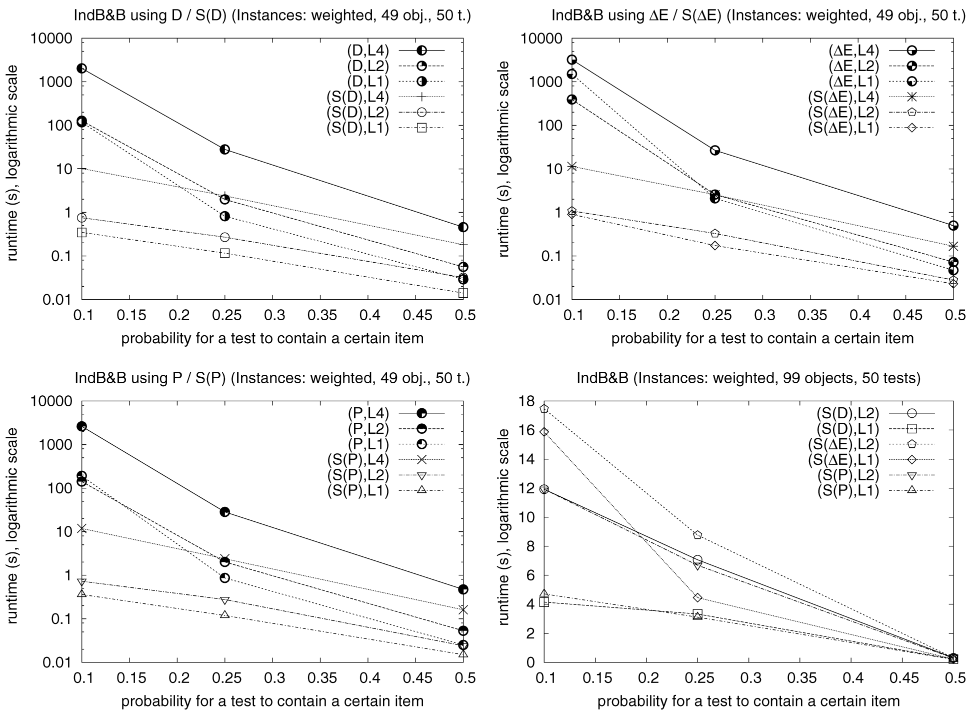

Fig. 4. Indirect approach using cost fixing on weighted benchmark sets.

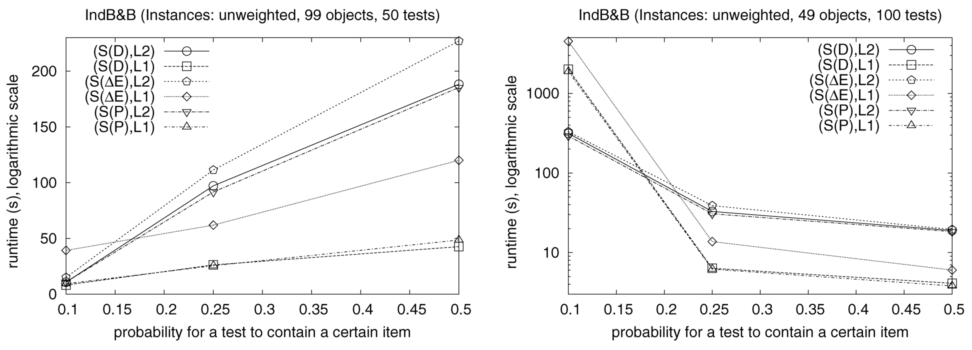

Fig. 5. Indirect approach on unweighted instances with 99 items, 50 tests (left) and 49 items, 100 tests (right).

De Bontridder et al. [2002] for the direct approach and to Tiemann [2002] for the indirect approach.

Figure 6 shows that both branch-and-bound approaches are between 10 to 100 times faster than Cplex. For instances having 49 items and 100 tests only one-third of all Cplex runs terminated within 24 hours.

In most cases, the indirect approach is about four times faster than the direct approach.

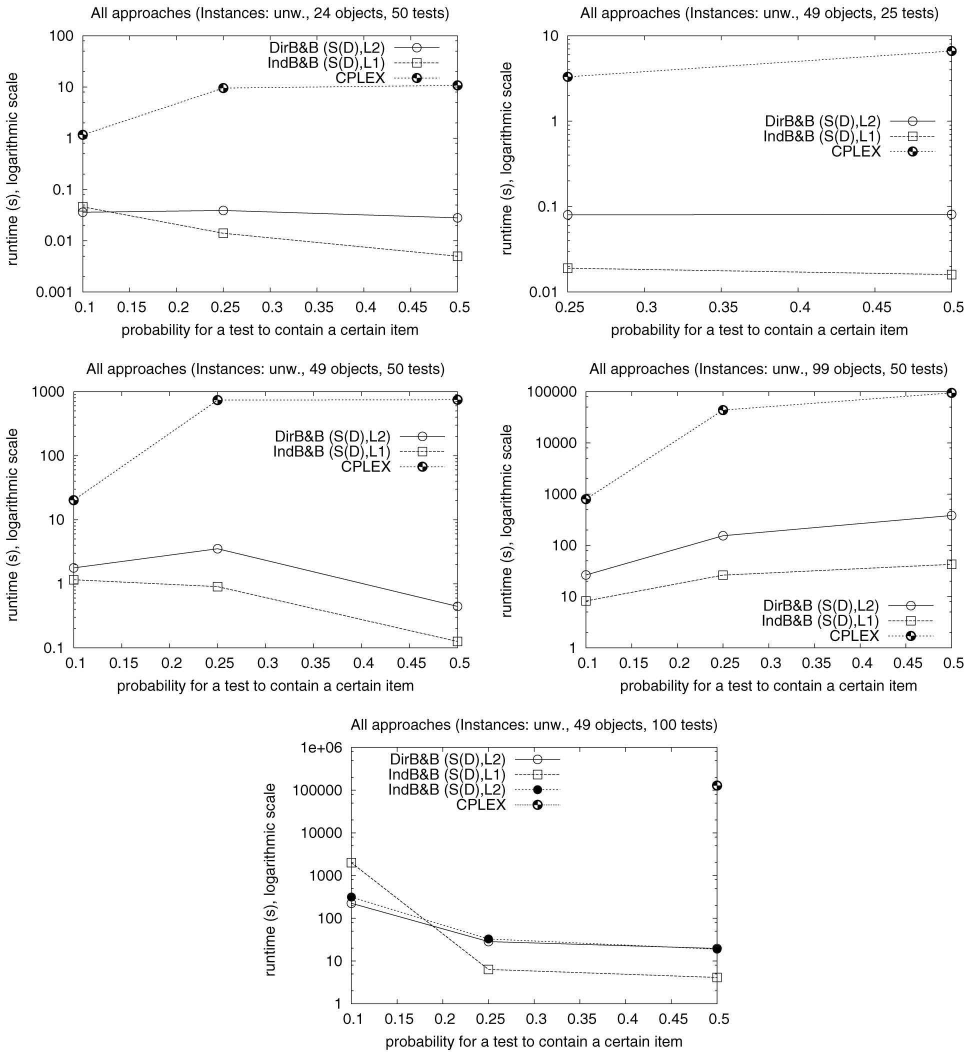

Fig. 6. Comparing direct approach, indirect approach, and CPLEX on all unweighted instances.

5.3 Weighted Benchmark Sets

As mentioned before, on weighted benchmark sets we do not compute an upper bound based on a Lagrangian lower bound, but we apply cost fixing in every branch-and-bound node (see Section 5.4). For the direct approaches, $(S(D), L_2)$ or $(S(P), L_2)$ are the most efficient approaches. Replacing L_2 by L_1 only slightly reduces runtime for the smaller instances. On larger instances (49, 100, 1/10), however, a factor of 2 is lost when using L_1 rather than L_2, as can be seen in Figure 7.

For the indirect approach, using $(S(D), L_1)$ or $(S(P), L_1)$ is the best choice (see Figure 4). In instances with 99 items and 50 tests, these two settings are about three times faster than all strategies using L_2. Interestingly, Figure 8

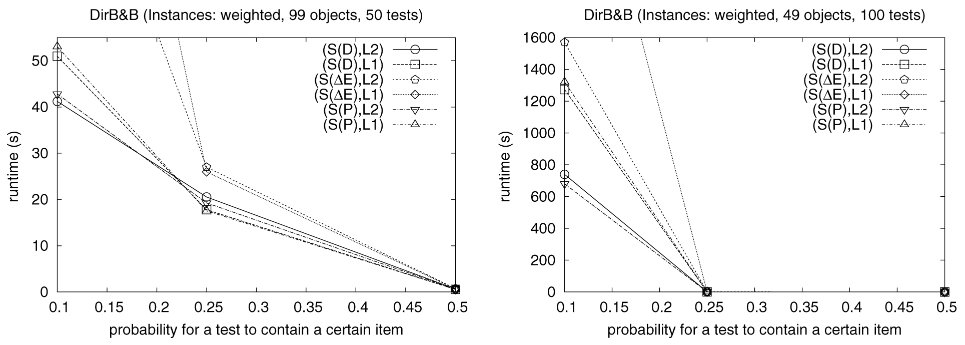

Fig. 7. Direct approach on weighted instances with 99 items, 50 tests (left) and 49 items, 100 tests (right).

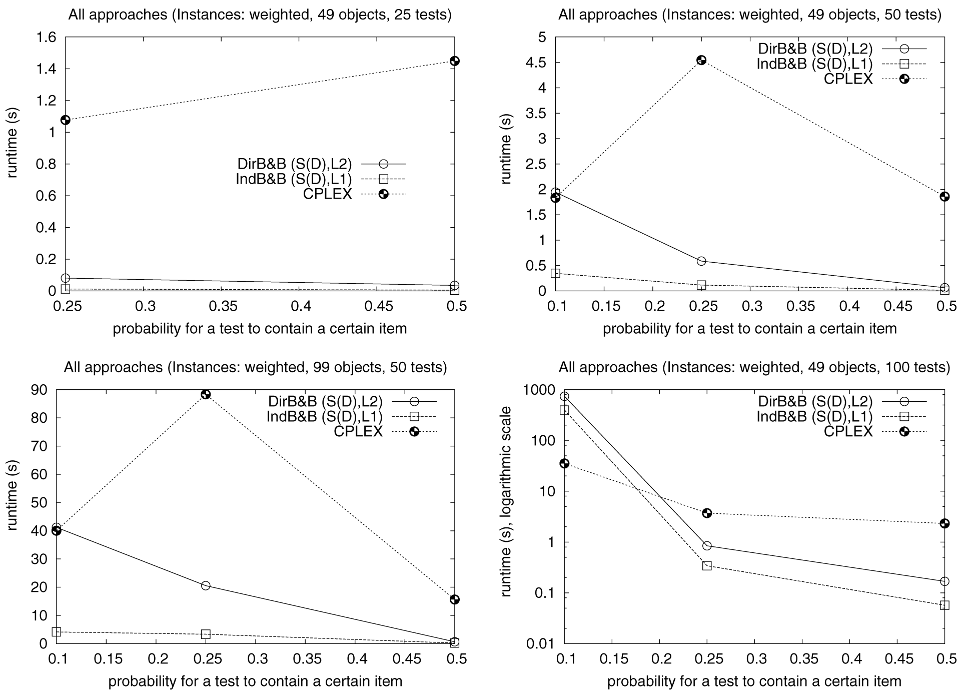

Fig. 8. Comparing the approaches on weighted benchmark sets.

shows that on the ten instances in class (49, 100, 1/10), CPLEX is able to outdo both the direct and the indirect approach by a factor of 10 or 20, respectively, whereas on all other classes (130 instances), the specialized algorithms prevail by up to a factor of 10.

The indirect approach is always faster than the direct one (factor 2–10).

5.4 Impact of Cost Fixing

In the direct approach, cost fixing, as described in Section 3.3, reduces runtime as well as number of branching decisions for almost all instances. As can be seen in Figure 9, cost fixing reduces runtime of instances (99, 50, {1/10, 1/4,

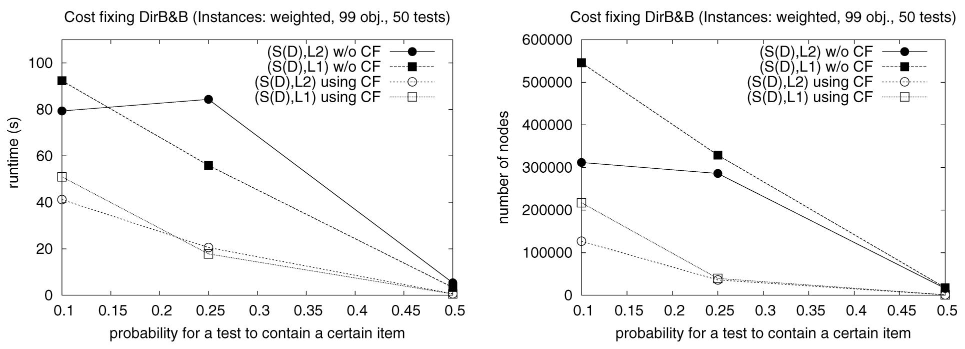

Fig. 9. Runtimes (left) and number of nodes (right) for the direct approach using/not using cost fixing (CF).

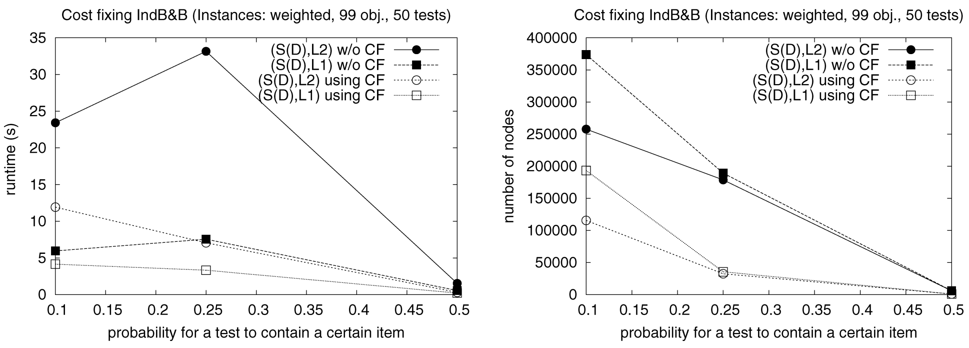

Fig. 10. Runtimes (left) and number of nodes (right) for the indirect approach using/not using cost fixing (CF).

1/2}) by a factor of 2 to 3, whereas the number of branch-and-bound nodes is reduced by a factor of 3 to 5.

Also, for the indirect approach, using cost fixing impacts positively on runtime as well as on number of branch-and-bound nodes. Only between 25–75% of the original runtime is needed when using cost fixing (see Figure 10). The number of branch-and-bound nodes is only 1/6–1/2 of the number of nodes when not using cost fixing. Impact of cost fixing diminishes, as more pairs of items are covered by a test. On the other hand, the number of branch-and-bound nodes is already very small, as can be seen in the figure. Experiments on larger instances that require more branching decisions are needed in order to conclude on the behavior of instances having many items per test.

5.5 Impact of Lower Bounds via Lagrangian Relaxation and Partial Cost

It is known that Lagrangian techniques are quite successful for general set-covering problems. We did some experiments to find out whether this also holds for the subclass of SCPs describing our TCP benchmarks. To do so, we compared on all weighted benchmark sets, our normal indirect approach with an indirect approach using in addition lower and upper bounds based on Lagrangian relaxation (see Sections 4.1.1 and 4.3.1).

If algorithms using quality criterion $S(D)$ are compared, the runtime increases, on average, by a factor of 21 (L_1) or 12 (L_2) by using the Lagrangian techniques, whereas the number of nodes reduces, on average, to 22% (L_1) or 20% (L_2). If, instead, only the ten most runtime-consuming benchmarks (49, 100, 1/10) are considered, the Lagrangian techniques can reduce the number of nodes, on average, to 1% and the runtime increases on average to 182 (L_1) or 136% (L_2). There is one instance on which the runtime is reduced to 81 (L_1) or 34% (L_2). This may indicate that lower bounds based on Lagrangian relaxations are only useful for very large problem instances where the expensive bound computation leads to a strong decrease in tree size.

In Section 4.1.2, we proposed the less expensive lower bound via partial cost. Unfortunately, the experiments with $S(D)$ algorithms show that this bound does not lead to a significant tree size reduction. On the weighted benchmark sets, the number of nodes decreases, on average, to 98 (L_1) or 99% (L_2). Therefore, it is not surprising that the runtime increases (to 230% with L_1 or 153% with L_2).

5.6 Impact of Upper-Bound Heuristics

Having good upper bounds in a branch-and-bound is helpful for fixing variables and pruning useless parts of the search tree. Our experiments show that additional upper-bound heuristics do improve runtime. However, the improvement is smaller than for the techniques discussed before. This is because of the good-quality criterion that guides the branch-and-bound search to promising parts of the search tree very early.

Additional upper bounds, thus, are the least important ingredients for a fast solver to the test-cover problem, although using them reduces runtime and number of branching nodes.

5.6.1 *Initial Upper-Bound Heuristic.* On weighted instances solution found by the initial upper-bound heuristic is typically between 0 to 39% away from the optimal solution, the average distance between these two is 9.98%. Computation time for the initial lower bound is, in all cases, smaller than 0.1 s. Depending on order strategy and lower bound, an initial upper bound might have more or less impact.

We started two runs using the indirect approach, lower bound L_2, and quality criterion $S(D)$. One run was, in addition, using the initial upper-bound heuristic. From 129 benchmark files, 111 were solved most quickly when applying the initial heuristic.[1] There were 18 benchmarks where runtime increases when applying the heuristic; 14 of these had an overall runtime of less than 5 s. On average, using the heuristic improves runtime by 12.9% over all 129 benchmarks and by 3.2% if we only consider those that ran 5 s or more. In Table II, we show how much time it costs for a pure branch-and-bound algorithm to be at least as good as the initial upper bound. Runtimes given in the table are accumulated over all 140 benchmark sets.

[1]Because of the limited resolution of the internal clock, only 129 out of 140 tests had a reported runtime higher than "0.0" s in both runs.

Table II. Time after which the Pure Branch-and-Bound Finds a Solution that is as Least as Good as the Initial Upper Bounds[a]

Setting	L_1, $S(P)$	L_1, $S(D)$	L_2, $S(D)$	L_1, $S(\Delta E)$	L_2, $S(P)$	L_2, $S(\Delta E)$
Runtime	33.66	34.72	62.62	65.28	249.89	290.77

[a]Times are accumulated over 140 weighted benchmark instances.

As can be seen, there is a great variance in helpfulness, measured as the time to pass until the branch-and-bound approach also detects a solution of similar or better quality. The most effective settings in our test ($S(D)$, L_1 and $S(P)$, L_1) turned out to profit least from upper-bound heuristics, whereas those combinations that costs a lot of time anyway (L_2), will benefit more.

5.6.2 *Impact of the Improvement Heuristic.* We applied the improvement heuristic (Section 4.3.3) to all improving solutions, as well as to all solutions that were, at most, 10% worse than the incumbent solution. Depending on the chosen lower bound and quality criterion, between 20 and 100 solutions were candidates.

Using lower bound L_1 and $S(D)$, we only slightly benefit from the heuristic: For the 129 instances used before we have a runtime decrease in 94 cases, when using the improvement heuristic, in the remaining cases runtime increases. The overall runtime gain compared to not using any heuristic, however, is only 1.05% and, thus, smaller than the gain achieved by using an initial upper bound. This is because of the fact that the positive effect of finding an upper bound somewhat earlier is exhausted by the additional runtime of calling the improvement heuristic several times during the search.

5.7 Summary of the Numerical Results

We will now summarize the results of our experiments by stating which algorithm should be used to get the fastest computation possible. Clearly, one should select a branch-and-bound algorithm based on our new indirect approach using the lower bound L_1 and as quality criterion either $S(D)$ or $S(P)$. If a weighted problem instance is given, a cost, based variable fixing should be applied.

Runtime of the algorithm is moderately improved by using initial upper-bound heuristics. Improving solutions found during branch-and-bound, on the other hand, will only have negligible effects. If weighted problem instances are given that are bigger than our benchmark instances, one should try whether lower bounds based on Lagrangian relaxations speed up the algorithm.

A generalized MIP solver cannot compete at all (unweighted instances) or only in few cases (weighted instances) in our experiments.

6. CONCLUSIONS

In this paper, we proposed several ideas to speed up branch-and-bound algorithms for the test-cover problem. We presented a simple, yet fast data structure, for these problems that is based on the idea to describe TCPs by SCPs. Experimental results show a typical speedup of up to a factor of 10, compared to an implementation proposed earlier using the standard data structure.

Furthermore, the SCP representation of a TCP allows us to use techniques that were originally introduced for SCPs in our TCP algorithm.

In contrast to branch-and-bound algorithms proposed earlier, our algorithm is also able to handle *weighted* test-cover problems. The cost-based fixing techniques turned out to be quite helpful because they reduce the number of explored branch-and-bound nodes by a factor of 2 to 6 and running time by a factor of 2 to 4.

APPENDIX

A. NUMERICAL REPRESENTATION OF EXPERIMENTAL RESULTS

The following tables contain the same information as the previously shown figures. Numerical results typically are better read on tables. Since the results in De Bontridder et al. [2002] have been shown in figures, we decided to show figures in the paper and to present the tables in the appendix.

The following tables list runtimes measured (s) for the given approach and benchmark sets (see also Section 5).

Table AI. Indirect Approach on Weighted Benchmark Sets (See Figure 4)

		D			$S(D)$		
Instances	p	L_4	L_2	L_1	L_4	L_2	L_1
49 objects, 50 tests	0.1	2033.257	127.078	117.165	10.206	0.755	0.347
	0.25	27.977	1.990	0.818	2.402	0.270	0.116
	0.5	0.458	0.056	0.029	0.181	0.032	0.014
		ΔE			$S(\Delta E)$		
Instances	p	L_4	L_2	L_1	L_4	L_2	L_1
49 objects, 50 tests	0.1	3222.855	391.526	1509.881	11.383	1.083	0.902
	0.25	26.658	2.565	2.098	2.552	0.328	0.173
	0.5	0.499	0.072	0.047	0.167	0.028	0.023
		P			$S(P)$		
Instances	p	L_4	L_2	L_1	L_4	L_2	L_1
49 objects, 50 tests	0.1	2621.564	141.814	191.226	11.846	0.730	0.361
	0.25	28.431	2.020	0.865	2.396	0.277	0.119
	0.5	0.472	0.053	0.025	0.161	0.024	0.015
		$S(D)$		$S(E)$		$S(P)$	
Instances	p	L_2	L_1	L_2	L_1	L_2	L_1
99 objects, 50 tests	0.1	11.922	4.148	17.469	15.877	11.928	4.714
	0.25	7.064	3.328	8.761	4.455	6.689	3.110
	0.5	0.312	0.219	0.309	0.225	0.286	0.199

Table AII. Indirect Approach on Unweighted Benchmark Sets (See Figure 5)

		$S(D)$		$S(E)$		$S(P)$	
Instances	p	L_2	L_1	L_2	L_1	L_2	L_1
99 objects, 50 tests	0.1	10.735	8.227	15.071	39.248	10.855	9.378
	0.25	97.183	26.272	111.429	61.932	91.707	25.540
	0.5	188.267	42.654	227.104	120.149	185.388	48.615
49 objects, 100 tests	0.1	2003.972	314.739	1883.531	291.716	4518.816	330.956
	0.25	6.346	32.836	6.193	30.675	13.750	38.947
	0.5	4.113	18.943	3.809	18.363	6.032	19.503

Table AIII. Comparing Approaches on Unweighted Benchmark Sets (See Figure 6)

		DirB&B	IndB&B		
Instances	p	$(S(D),L_2)$	$(S(D),L_1)$	$(S(D),L_2)$	CPLEX
49 objects,	0.25	0.080	0.019	0.048	3.306
25 tests	0.5	0.081	0.016	0.051	6.629
24 objects,	0.1	0.036	0.046	0.029	1.168
50 tests	0.25	0.039	0.014	0.047	9.546
	0.5	0.028	0.005	0.022	10.772
49 objects,	0.1	1.771	1.158	1.098	20.212
50 tests	0.25	3.527	0.904	3.248	737.997
	0.5	0.444	0.126	0.379	745.769
99 objects,	0.1	26.401	8.227	10.735	798.601
50 tests	0.25	154.143	26.272	97.183	43649.410
	0.5	382.581	42.654	188.267	94667.450
49 objects,	0.1	224.331	2003.972	314.739	>86400.000
100 tests	0.25	28.498	6.346	32.836	>86400.000
	0.5	19.669	4.113	18.943	128120.800

Table AIV. Direct Approach on Weighted Benchmark Sets (See Figure 7)

		$S(D)$		$S(E)$		$S(P)$	
Instances	p	L_2	L_1	L_2	L_1	L_2	L_1
99 objects,	0.1	41.205	50.971	114.506	179.041	42.800	53.029
50 tests	0.25	20.540	17.789	26.992	25.967	19.247	17.539
	0.5	0.645	0.556	0.723	0.658	0.614	0.549
49 objects,	0.1	739.593	1273.399	1570.407	3448.706	680.426	1320.279
quad 100 tests	0.25	0.843	0.684	1.113	1.003	0.756	0.684
	0.5	0.168	0.114	0.195	0.145	0.175	0.116

Table AV. Comparing Approaches on Weighted Benchmark Sets (See Figure 8)

Instances	p	DirB&B $(S(D),L_2)$	IndB&B $(S(D),L_1)$	CPLEX
49 objects,	0.25	0.081	0.012	1.077
25 tests	0.5	0.035	0.005	1.449
49 objects,	0.1	1.945	0.347	1.833
50 tests	0.25	0.587	0.116	4.544
	0.5	0.063	0.014	1.859
99 objects,	0.1	41.205	4.148	40.008
50 tests	0.25	20.540	3.328	88.340
	0.5	0.645	0.219	15.620
49 objects,	0.1	739.593	395.886	35.252
100 tests	0.25	0.843	0.342	3.702
	0.5	0.168	0.057	2.319

Table AVI. Comparing Direct Approach with and w/o Cost Fixing (CF), (See Figure 9)

		Runtime (s)				Number of Nodes			
		$(S(D),L_2)$		$(S(D),L_1)$		$(S(D),L_2)$		$(S(D),L_1)$	
Instances	p	w/o CF	CF	w/o CF	CF	w/o CF	CF	w/o CF	CF
99 objects,	0.1	79.342	41.205	92.349	50.971	311837	126937	546018	217225
50 tests	0.25	84.346	20.540	55.877	17.789	286005	35931	329044	39516
	0.5	5.347	0.645	3.373	0.556	15920	1019	17768	1052

Table AVII. Comparing Indirect Approach with and w/o Cost Fixing (CF) (See Figure 10)

		Runtime (s)				Number of nodes			
		$(S(D), L_2)$		$(S(D), L_1)$		$(S(D), L_2)$		$(S(D), L_1)$	
Instances	p	w/o CF	CF	w/o CF	CF	w/o CF	CF	w/o CF	CF
99 objects, 50 tests	0.1	23.418	11.922	5.958	4.148	257772	115582	373869	193349
	0.25	33.160	7.064	7.555	3.328	178727	32491	189266	35547
	0.5	1.536	0.312	0.553	0.219	5898	917	6057	947

ACKNOWLEDGMENTS

We would like to thank K. M. J. De Bontridder who provided us with the original benchmark sets used in [De Bontridder et al. 2002]. Furthermore, we would like to thank the anonymous referees for their helpful comments on this and an earlier version of this work.

REFERENCES

Andersen, E. and Andersen, K. 1995. Presolving in linear programming. *Mathematical Programming 71*, 221–245.

Beasley, J. E. 1990. A lagrangian heuristic for set-covering problems. *Naval Research Logistics 37*, 151–164.

Berman, P., DasGupta, B., and Kao, M.-Y. 2004. Tight approximability results for test set problems in bioinformatics. In *Proceedings of the 9th Scandinavian Workshop on Algorithm Theory (SWAT '04)*. Lecture Notes in Computer Science, vol. 3111. Springer, New York. 39–50.

Caprara, A., Fischetti, M., and Toth, P. 2001. Algorithms for the set-covering problem. *Annals of Operations Research 98*, 353–371.

Chang, H. Y., Manning, E., and Metze, G. 1970. *Fault Diagnosis of Digital Systems*. Wiley, New York.

De Bontridder, K. M. J., Lageweg, B. J., Lenstra, J. K., Orlin, J. B., and Stougie, L. 2002. Branch-and-bound algorithms for the test-cover problem. In *Proceedings of the 10th European Symposium on Algorithms (ESA '02)*. Lecture Notes in Computer Science, vol. 2461. Springer, New York. 223–233.

De Bontridder, K. M. J., Halldórsson, B. V., Halldórsson, M. M., Hurkens, C. A. J., Lenstra, J. K., Ravi, R., and Stougie, L. 2003. Approximation algorithms for the test-cover problem. *Mathematical Programming 98*, 477–491.

Fischer, M. L. 1981. The lagrangian relaxation method for solving integer programming problems. *Management Science 27,* 1, 1–18.

Garey, M. R. and Johnson, D. S. 1979. *Computers and Intractability*. Freeman, San Francisco, CA.

Halldórsson, B. V., Halldórsson, M. M., and Ravi, R. 2001a. On the approximability of the minimum test collection problem. In *Proceedings of the 9th European Symposium on Algorithms (ESA '01)*. Lecture Notes in Computer Science, vol. 2161. Springer, New York. 158–169.

Halldórsson, B. V., Minden, J. S., and Ravi, R. 2001b. PIER: Protein identification by epitope recognition. *Currents in Computational Molecular Biology 2001*, 109–110.

Held, M. and Karp, R. M. 1971. The traveling-salesman problem and minimum spanning trees: Part II. *Mathematical Programming 1*, 6–25.

Johnson, D. S. 1974. Approximation algorithms for combinatorial problems. *Journal of Computer and Systems Sciences 9*, 256–278.

Lageweg, B. J., Lenstra, J. K., and Rinnooy Kan, A. H. G. 1980. Uit de practijk van de besliskunde. In *Tamelijk briljant; Opstellen aangeboden aan Dr. T. J. Wansbeek*. Amsterdam.

Lemaréchal, C. 2001. Lagrangian relaxation. In *Computational Combinatorial Optimization—Optimal or Provably Near-Optimal Solutions*. Lecture Notes in Computer Science, vol. 2241. Springer, New York. 112–156.

Moret, B. and Shapiro, H. 1985. On minimizing a set of tests. *SIAM Journal on Scientific and Statistical Computing 6*, 983–1003.

PAYNE, R. W. 1981. Selection criteria for the construction of efficient diagnostic keys. *Journal of Statistical Planning and Information 5*, 27–36.

RESCIGNO, A. AND MACCACARO, G. A. 1961. The information content of biological classification. In *Information Theory: Fourth London Symposium*. Butterworths, London. 437–445.

RYPKA, E. W., CLAPPER, W. E., BROWN, I. G., AND BABB, R. 1967. A model for the identification of bacteria. *Journal of General Microbiology 46*, 407–424.

RYPKA, E. W., VOLKMAN, L., AND KINTER, E. 1978. Construction and use of an optimized identification scheme. *Laboratory Magazine 9*, 32–41.

TIEMANN, K. 2002. Ein erweiterter Branch-and-Bound-Algorithmus für das Test-Cover Problem. B.S. thesis, University of Paderborn.

Received September 2005; revised January 2007; accepted November 2005