On Space Constrained Set Selection Problems

Themis Palpanas

University of Trento

Nick Koudas

University of Toronto

Alberto Mendelzon

University of Toronto

Abstract

Space constrained optimization problems arise in a variety of applications, ranging from databases to ubiquitous computing. Typically, these problems involve selecting a set of items of interest, subject to a space constraint.

We show that in many important applications, one faces variants of this basic problem, in which the individual items are sets themselves, and each set is associated with a benefit value. Since there are no known approximation algorithms for these problems, we explore the use of greedy and randomized techniques. We present a detailed performance and theoretical evaluation of the algorithms, highlighting the efficiency of the proposed solutions.

Key words: space constrained set selection problem, optimization problem, ubiquitous computing, data warehouses

1 Introduction

In recent years there has been a proliferation in the amount of information that is being produced. The data that are being gathered and stored involve several aspects of human activity. Retailers register individual transactions in their

Email addresses: `themis@disi.unitn.eu` (Themis Palpanas), `koudas@cs.toronto.edu` (Nick Koudas), `mendel@cs.toronto.edu` (Alberto Mendelzon).

stores, businesses keep track of the interactions with customers and suppliers, scientific laboratories record various measurements of interest. Obviously, the volume of data generated in such situations is huge.

It is often times the case that not all data are stored, because of their sheer size. In order to save space people usually aggregate data and only store the aggregates. A typical scenario in a retail store data warehouse, which records user transactions, would be to aggregate the sales at the day level [14]. Similarly, scientific measurements such as temperature, precipitation, air pollution, are aggregated at the hour level. In many cases even the size of the aggregated data is too large, so that portions of it have to be moved to tertiary storage or permanently deleted. Then, we have to choose which parts of the data to keep around.

Limited space can become a restricting factor even when the amount of data is not large, but when the actual available space is limited. Characteristic examples of this scenario are handheld devices, which have small storage capacity, and for which space allocation should be done with care. Mobile users in general face an analogous problem in terms of bandwidth. When the available bandwidth or connection time resources are limited, we have to choose carefully which bytes to transmit over the communication link.

Observe that all the above examples lead to space constrained optimization problems. We wish to fill the given space with the most useful of our data, or in other words with the data that will give us the highest benefit. The benefit is determined by the number and importance of the queries that we can answer based on the stored data. The view selection problem [10] in the context of data warehouses is an instance of such an optimization problem, and has been studied extensively in the literature.

In this paper we study the above optimization problem for the case when the benefit we get for admitting items in the solution is associated to *sets* of items, and not to individual items. In both the Knapsack and the view selection problems the benefit increases with each item that is added to the solution. However, this is not true for the class of applications that we consider. In this setting there are additional constraints inherent in the problem, which dictate that when we select a new item to insert in the solution the added benefit is zero, unless a set of related items are inserted as well. The above constraints make the problem harder, and in Section 6 we discuss how they affect the solution procedure. For the rest of the paper we refer to this optimization problem as the *Constrained Set Selection (COSS)* problem.

Several problems that have been presented in the literature [21,24,17] are related to the *COSS* problem. Nevertheless, none of the known results or techniques seem to be applicable in this case [16]. This study is the first thorough examination of practical algorithms that solve the *COSS* problem. Our experimental evaluation can serve both as a practitioner's guide, and also provide intuition about the nature of the problem from a theoretical perspective.

In this paper, we make the following contributions.

- We formulate optimization problems concerning the selection of *sets* of items that under a space constraint yield the highest benefit, where benefits are associated to sets of items. This kind of problems appear in various domains, and are very interesting in practice.
- We derive the complexity of the above optimization problems, and propose several algorithms for their solution. Since there are no known polynomial time approximation algorithms for these problems, we examine the use of known optimization principles in this context [7], such as greedy and randomization.
- We explore the properties of the above techniques with an experimental evaluation. Our results illustrate the behavior of the algorithms under different settings, and highlight the benefits of each approach.
- We discuss some theoretical properties of the proposed algorithms. Based on our analysis, we present worst case scenarios for the algorithms. This offers insight into the operation of the algorithms, and provides a practical guide for selecting among the techniques proposed.

The outline of the paper is as follows. Section 2 illustrates examples in which the *COSS* problem is applicable, and Section 3 reviews the related work. In Section 4 we present the formulation of the optimization problem, and Section 5 proposes algorithms for their solution. In Section 6 we show experimental results evaluating the performance and the utility of the proposed algorithms. Finally, in Section 7 we present a theoretical analysis of the algorithms and discuss their relative performance.

2 Applications of the *COSS* Problem

In the following sections we present with more detail two specific example applications of the problem. The first application comes from the world of data warehousing, and the second from pervasive computing.

2.1 Aggregate Selection for Approximate Querying in Datacubes

The volume of data stored in OLAP tables is typically huge, often times in the order of multiple terabytes to petabytes. In large organizations, where terabytes of data are generated every day, it is common practice to aggregate

			7 All			
			5 Levi's		6 CK	
			1 men's	2 women's	3 men's	4 women's
7 All	5 NY	1 Queens	40	10	50	5
		2 New York	15	45	65	10
	6 Ontario	3 Toronto	15	40	65	18
		4 Ottawa	55	40	35	30

(a) The entire dataset.

			55	55	115	15
NY	Queens	105	x	x	x	x
	New York	135	x	x	x	x

(b) Aggregated values for the upper half of the dataset.

Fig. 1. Example dimension hierarchies on two dimensional sales data.

these data in order to save storage space. During this procedure the detailed information that produced the aggregates is lost. In other cases the detailed data is moved to tertiary storage, which makes the task of accessing them very cumbersome. Nevertheless, users are often interested in inquiring about the data that generated the summarized form. In such cases, generating good estimates for the original data in response to queries is a pressing concern. Efficient solutions to the above problem have been proposed in the literature [5,25,4,22], where the reconstruction techniques are based solely on the aggregated information in order to produce estimates for the detailed values.

Example 1 *Figure 1(a) shows a typical OLAP table, with two dimension attributes (location and jeans), and hierarchies defined in each dimension. Assume that for the state of NY (i.e., the upper half of the table) we only store the* aggregated *sales for each city and for each category as shown in Figure 1(b), and that we have deleted all the detailed values. The users might want to inquire about the number of redtab jeans sold in Queens NY (a point query), or they might request the number of any kind of jeans sold in each city of NY state (a range query). It turns out that we can use the information that is stored in the aggregates, in order to provide approximate answers to the above queries. The reconstruction algorithm will need both of the aggregates shown in Figure 1(b), and will be able to produce estimates for all the values marked as "x".*

Now consider the case where we are allowed only a limited amount of space for storing the aggregates. We can compute the space requirements of the aggregates, either by computing them, or by applying estimation techniques [26].

In this setting, we would like to materialize the subset of the aggregates that satisfies the given space constraint, and at the same time is necessary for the reconstruction of the important queries. The importance, or benefit, of each query can be determined automatically by observing the system workload, or it can be manually set by the user. Note that, unlike the view selection problem in data warehouses [10], in this case we get no partial benefits if we only materialize a subset of the aggregates. This is because the algorithms that reconstruct the detailed values [25,4,22] require a specified set of aggregates in its entirety.

2.2 Profile-Driven Data Management

We draw our second example from the area of *pervasive* or *mobile computing* [2,29,23]. The premise of pervasive computing is that all the relevant information (at each point in time and space) should be accessible to any mobile user. The users specify what information is relevant by using a profile language, which allows each user to define a number of queries that wants to be answered by the data items in the cache of the computer. Each query is associated with a benefit value, which reflects the query's utility to the user, and needs a set of the data items in order to be answered. The data items have specified storage requirements. Furthermore, each data item may help answer more than one queries.

One of the problems in the aforementioned context is what data to send to a mobile computer. The choice of the data to download is crucial, because it determines which of the queries in the profile can be answered, and consequently the total benefit that can be achieved. This is an interesting problem, in which we have to take into account both the user profile that describes what data are useful, and a space restriction on the mobile computer, or equivalently a connection time restriction (in both cases we can only transmit a limited specified amount of data).

Example 2 *Consider a user visiting a new city [2]. The profile of such a user could contain the following 3 statements (or queries).*

(1) Find location, requires map.
(2) Find rental car, requires car agency list, and map.
(3) Find restaurant, requires restaurant list, and map.

In this example, if only one of the two required data items for queries 2 and 3 is present in the mobile computer, then we cannot answer those queries. There is no use having a list of restaurants if there is no map available, and vice versa. Therefore, the benefit gained is zero. Also observe that the map item is useful in answering all 3 queries.

3 Related Work

In the problem of view selection for data warehouses [10] we want, given a space constraint, to materialize a set of views in order to minimize the response time of queries to the data warehouse. In this case however, by selecting one of the views required by a query we get a fraction of the associated benefit. A subsequent study shows that the heuristics for the view selection problem do not provide any competitiveness guarantee against the optimal solution [13], and proposes the experimental comparison of the available algorithms. Many other studies [9,27,28,1] deal with the view selection problem as well.

A similar optimization problem appears in the context of database design under the name of vertical partitioning [19,20], where the *Bond Energy Algorithm* [18] has been employeed for its solution. In this domain the problem can be stated as follows. We have a set of applications accessing a particular set of attributes in the relations of a database. Each attribute requires a certain amount of space (to store the values of the attribute in the relation), and may be used by more than one applications. We want to find an allocation of the attributes to the memory hierarchy (e.g., main memory, local disk, network disk) so as to minimize the execution time of the applications. Similar to the view selection problem, partial benefits are credited, which is not true for our case.

The *COSS* problem is related to a family of optimization problems known as the Weighted constrained Maximum Value sub-Hypergraph problem [21], and the studies show that even simple instances of the problem do not accept polynomial solutions. We are not aware of any algorithms proposed for the *COSS* problem in that area.

Much work has been devoted to the *Set Cover* problem [6], which is similar to the optimization problem we are trying to solve, and efficient approximation algorithms [24] have been proposed for its solution. However, the same techniques do not seem to be applicable in our case. Furthermore, the *COSS* problem is also similar to the *k-Catalog Segmentation* problem [17]. But no interesting positive results are known for this problem either. In this study we try instead to examine the *COSS* optimization problem from a practical point of view, and we hope that the results we report will stimulate more work in this area.

The *COSS* problem has also appeared in the domains of aggregate selection for approximate querying in datacubes [22], and profile-driven data management [2], which exemplify its diverse practical applications. However, the solution of the optimization problem was not studied. All the algorithms described in this paper can be applied for the solution of the problem in these domains.

4 Problem Formulation

In the following paragraphs we establish the terminology used in the rest of the paper, we present the formal statement of the problem, and discuss its complexity.

Let *components* be the individual data items that are available for use, and *objects* be the consumers of the components. When all the components required by an object are selected in the solution we say that the object is *satisfied*. Each object is associated with a benefit, which is claimed when the object is satisfied. Components are associated with a space requirement, which is the storage space they will occupy when selected. When a component is selected we say that it is *materialized*.

Example 3 *According to the terminology we introduced above, the problem of view selection for approximate querying in datacubes (see Example 1) can be translated as follows. The queries that we want to be able to reconstruct are the objects, while the sets of aggregates needed by the reconstruction algorithm are the components.*

Similarly, in the profile-driven data management problem (see Example 2) the objects are the queries contained in the user profile, and the components are the data items required to answer the queries.

We can construct a bipartite graph $G(U, V, E)$, where U is the set of objects, V the set of components, and E the edge set of the graph. An edge exists between $u \in U$ and $v \in V$ if component v is required by object u. An example of the general form of the graph is shown in Figure 2.

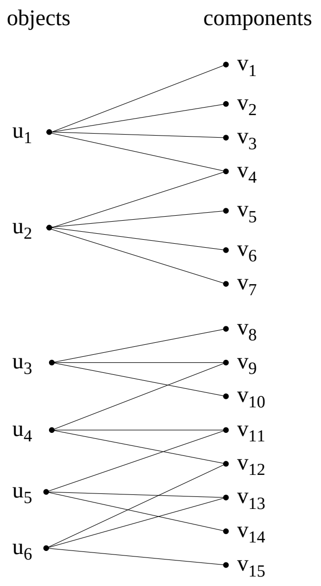


Fig. 2. An example of the bipartite graph G.

Let $V[i], 1 \leq i \leq |V|$ be a binary vector having 1 in position i if and only if we have selected component v_i for materialization, and $s(v_i), 1 \leq i \leq |V|$ be a function determining the space requirements of component v_i. Let $U[j], 1 \leq j \leq |U|$, be a binary vector having 1 in position j if and only if all the components required by object u_j have been materialized. Each object u_j is also associated with a value $b(u_j)$, which specifies its benefit, and is a measure of its importance. Given a constraint W on the total space available for the marginals, we are interested in maximizing the total benefit of queries answered, while satisfying the space constraint. Then, the *Constrained Set Selection (COSS)* optimization problem can be stated as follows.

Problem 1 *Maximize* $\sum_j U[j]b(u_j)$, *subject to* $\sum_i V[i]s(v_i) \leq W$.

It is easy to show that the Knapsack problem reduces to a special case of the *COSS* problem. Hence, according to the following lemma, the optimization problem is NP-Hard.

Lemma 1 *The* COSS *problem is NP-Hard.*

Proof: We will show that the integral *Knapsack* problem reduces to *COSS*$'$, a special case of *COSS*. Hence, *COSS*$'$ and subsequently *COSS* are NP-Hard. In the *Knapsack* problem, we are given a space constraint B, and a set D of data items, where item d_k has space requirement $s(d_k)$ and benefit $b(d_k)$. We wish to select a subset D' of the items, so as to maximize the total benefit $\sum_{d \in D'} b(d_k)$, subject to the constraint $\sum_{d \in D'} s(d_k) \leq B$. Let *COSS*$'$ be an instance of *COSS*, where each and every object u_k in U has unit benefit (i.e., $b(u_k) = 1$), and requires one and only one component v_k from V with space requirement $s(v_k)$. Associate each component v_k with $b(d_k)$ objects from U, and let $s(v_k) = s(d_k)$. Then, the mapping of an item d_k into a component v_k concludes the reduction. $\square$

Observe that in the special case where all the queries have the same benefit, the problem is one of trying to satisfy the largest number of objects possible given the space constraint.

5 Algorithms for the *COSS* Problem

In the following paragraphs we propose several different algorithms for the solution of the *COSS* problem. We start by discussing optimal algorithms, and then present efficient greedy heuristics that can scale up to realistic sizes of the problem. We also explore the applicability of *simulated annealing*, a randomized algorithm, and *tabu search*, a meta-heuristic technique. The above methods have been used extensively in the past for solving a variety of hard problems, including scheduling, routing, and graph optimization [12,11,8]. Such algorithms have the ability to explore a larger area in the solution space than greedy algorithms, without getting stuck in local minima.

5.1 On Finding the Optimal Solution

There are two reasons we include an exhaustive algorithm in our discussion. First, it will demonstrate the dramatic difference in execution time between the optimal algorithm and the heuristics. The exhaustive algorithm has to test $O(2^{|U|})$ number of possible solutions, where $|U|$ is the number of objects. Second, and most important, it will serve as a basis for comparison of the quality of the results with the heuristic approaches. Unfortunately, this comparison will only be feasible for very small sizes of the problem.

Evidently, in order to find the optimal solution we do not need to enumerate and examine every single possible answer in the solution space. Solutions with sufficiently low cardinality (such that they are much lower than the space constraint) are bound to be sub-optimal, because we can always add new objects to the solution. Similarly, for answers that have already exceeded the space constraint we can safely prune all the solutions with higher cardinality. Finally, we can also prune the solutions that we know will never exceed the current best solution. These are the partial solutions that even if we fill up all the remaining space with the smallest objects, while assuming that these objects carry the maximum benefit, their total benefit will be smaller than the highest benefit we have seen so far. We will call the straight forward exhaustive algorithm *Naive*, and the one that prunes the search space *NaivePrune*.

Dynamic programming has been applied for the solution of the *Knapsack* problem, some instances of which can be solved in pseudo-polynomial time [6]. However, this technique cannot be applied in our case. We will revisit this issue in Section 7.2.

What makes the *COSS* problem difficult is the fact that in order to satisfy an object we need *all* the corresponding components. Materializing a subset of the components for some object does not credit us part of the object's benefit.

5.2 Solutions Based on Bond Energy

We now present algorithms based on the *Bond Energy Algorithm* [18]. This technique has been used for the vertical partitioning problem in databases [19]. A high level description of the algorithm we propose is shown in Figure 3. The algorithm starts by computing a measure of interrelation between each pair

```
Objective: Compute a solution for the COSS problem.
Input: A set $U$ of objects, a set $V$ of components, a space constraint $W$, and the bipartite
       graph $G(U, V, E)$, where $e_{ij} \in E$ denotes that component $v_j$ is used by object $u_i$.
Output: A set of components $S \subset V$ to materialize.
1 procedure SolveCOSS()
2    let square matrix $A[\cdot] = 0$;
3    let $S = \emptyset$;                /* selected components $S \subset V$ */
4    for i,j=1 to $|V|$
5      $A[ij]$ =interrelation measure between components i and j;
6    $A[\cdot]$ = MakeBlockDiag($A[\cdot]$);
7    $S$ = Split($A[\cdot]$);
8    return($S$);

9 procedure MakeBlockDiag($A[\cdot]$)
10   let matrix $A'[\cdot]$ be empty;
11   for i=1 to $|V| - 1$
12     select one of the remaining columns in $A[\cdot]$;
13     let $f_{max} = 0$;     /* maximum increase in bond energy so far */
14     for j=1 to i+1
15       place new column in $A'[\cdot]$ in position j;
16       $f_{max} = max$(bond energy increase after placing new column in position j, $f_{max}$);
17     keep the $A'[\cdot]$ corresponding to $f_{max}$;
18   return($A'[\cdot]$);

19 procedure Split($A[\cdot]$)
20   let $f_{max} = 0$;     /* total benefit of best solution so far */
21   for i=1 to $|V|$
22     make first column of $A[\cdot]$ last, and first row last;
23     let j:=1;
24     let $S = \emptyset$;                /* selected components $S \subset V$ */
25     while (j< $|V|$ $\wedge$ components in $S$ satisfy the space constraint)
26         $S = S \cup \{$component $v$ corresponding to j-th column of $A[\cdot]\}$;
27         $f_{max} = max$(total benefit of objects satisfied by components in $S$, $f_{max}$);
28   return($S$ corresponding to $f_{max}$);
```

Fig. 3. The *BondEn* algorithm.

of components $v_i, v_j \subset V$. The interrelation measures for all possible pairs of components is captured in the square matrix A (line 5 of the algorithm):

$$A[i, j] = \sum_{u_k | u_k \in U, e_{ki} \in E, e_{kj} \in E} b(u_k).$$

This measure is a function of the number and the benefit of the objects that require both components, v_i and v_j. The larger the number of objects and their benefits, the stronger the connection between the pair of components is.

Then, the procedure *MakeBlockDiag*() permutes the columns of A (or equivalently the rows since A is symmetric), and transforms it into a semiblock diagonal form, where large interrelation values tend to be grouped together. This transformation is expressed by the formula

$$\max\left(\sum_{k=1}^{|V|}\sum_{l=1}^{|V|} A[k,l](A[k,l-1]+A[k,l+1]+A[k-1,l]+A[k+1,l])\right),$$

which is the mathematical representation of the bond energy. We are seeking for the maximum value of this expression over all possible arrangements of the columns of matrix A. The algorithm proceeds greedily by considering a single column at a time. The above procedure is very fast, avoiding to examine the entire exponential search space, and is still able to find a near optimal form for A [18].

In the final step the algorithm splits the set of components V into S and $V-S$. Essentially, S is the set of components that can be used to satisfy the largest fraction of high-benefit objects while restricting the amount of available space. Just a single split point in A determines S and $V-S$. The loop in line 21 makes sure that we won't miss good solutions for S even if S was originally situated in the centre of A. We will refer to this algorithm as *BondEn (Bond Energy)*. Its time complexity is $O(|V|^3)$.

Note that in order to transform matrix A into a semiblock diagonal form, we add one column at a time in the matrix position that results in the largest increase to the overall bond energy. However, the selection of the column to add is arbitrary. Instead, we can enhance the technique by using a greedy column selection approach. That is, choosing in every step among all the available columns (i.e., the ones not already placed) the one that leads to the highest bond energy. We will call this version of the algorithm *BondEnGr (Bond Energy Greedy)*, and its time complexity now becomes $O(|V|^4)$. We also experimented with another two variations of the algorithm, where we make sure that during the split step we do not include in S any components required by objects that are not satisfied. We call the above variations *BondEn-SpAll* and *BondEnGr-SpAll*, which are the extensions of *BondEn* and *BondEnGr* respectively.

5.3 Solutions Based on Greedy Algorithms

The greedy algorithms provide a very fast alternative to solving the *COSS* problem, and are particularly appealing even for very large instances of the problem. The skeleton of the family of greedy algorithms we propose is depicted in Figure 4. They start with an empty solution set, and at each step

```
Objective: Compute a solution for the COSS problem.
Input: A set $U$ of objects, a set $V$ of components, a space constraint $W$, and the bipartite
        graph $G(U, V, E)$, where $e_{ij} \in E$ denotes that component $v_j$ is used by object $u_i$.
Output: A set of components $S \subset V$ to materialize.
1 procedure SolveCOSS()
2     let $O = \emptyset$;        /* selected objects $O \subset U$ */
3     let $S = \emptyset$;        /* selected components $S \subset V$ */
4     while ($|S| < |V| \wedge$ components in $S$ satisfy the space constraint)
5         among all the objects $\{u | u \in U \wedge u \notin O\}$ select $u_i$ that satisfies the greedy condition;
6         let $O = O \cup \{u_i\}$;
7         let $S = S \cup \{v_j | e_{ij} \in E \wedge v_j \notin S\}$;
8     return($S$);
```

Fig. 4. The greedy algorithm.

they add to the solution set those components that satisfy the greedy condition. Before discussing the alternatives, we introduce some new notation that will be necessary for the mathematical formulation. Let S^k be the set of components that the algorithm has selected during the past k iterations, and O^k be the set of objects that are satisfied given the components in S^k. We are interested in deciding how to update those sets during iteration $k+1$. Let $CO^k = U - O^k$ be the set of candidate objects for selection during iteration $k+1$. Let $C_l = \{v_j | e_{lj} \in E \wedge v_j \notin S^k\}$ be the set of components, which are not in S^k, required for object $u_i \in CO^k$. The additional amount of space required by these components is given by the formula $f(u_l) = \sum_{v_j \in C_l} s(v_j)$. As before, we assume that we know the graph $G(U, V, E)$, where U is the set of objects, V is the set of components, and E is the set of edges indicating which components are needed in order to satisfy each object. The greedy step can take any of the following four forms.

1. Accept to the solution set those components that will satisfy a new object, and that require the least amount of space. More formally, for each object $u_i \in CO^k$ calculate the additional space required for storing the corresponding components, given by $f(u_i)$. Let $u_l = \arg\min_{u_i} f(u_i)$ be the object that requires the least additional space. Then, $O^{k+1} = O^k \cup u_l$, and $S^{k+1} = S^k \cup C_l$.

This approach is trying to satisfy as many objects as possible. The intuition is that if there are many objects answered then the total benefit of those objects will be high. However, this may not be true if all the satisfied objects happen to have low benefit values. The time complexity of this algorithm is $O(|U|^2)$, and we will refer to it as *GrSp (Greedy Space)*.

2. Accept to the solution set those components that will satisfy the new object with the highest benefit attached to it. More formally, let $u_l = \arg\max_{u_i \in CO^k} b(u_i)$ be the object with the highest benefit among all the candidate objects. Then,

$O^{k+1} = O^k \cup u_l$, and $S^{k+1} = S^k \cup C_l$.

The goal of this alternative is to answer as many of the high-benefit objects as possible. This approach is likely to fail if the required components have unexpectedly high space requirements. We will refer to this algorithm as *GrBen (Greedy Benefit)*. Its time complexity is $O(|U|^2)$.

3. Accept to the solution set those components that will satisfy a new object, and the ratio of the object benefit over the total space required by the selected components is minimal. More formally, for each object $u_i \in CO^k$ calculate the additional space required for storing the corresponding components, given by $f(u_i)$. Let $u_l = \arg\max_{u_i} \frac{b(u_i)}{f(u_i)}$ be the object that has the highest benefit per unit of additional space required by its components. Then, $O^{k+1} = O^k \cup u_l$, and $S^{k+1} = S^k \cup C_l$.

This variation of the algorithm is trying to account for the extreme cases we identified as weaknesses to the previous two alternatives. We will refer to it as *GrBenSp (Greedy Benefit per unit Space)*. Its time complexity is $O(|U|^2)$.

4. Accept to the solution at each iteration an individual component. Select the component that fits in the remaining space and yields the maximum B/s ratio, where B is the total benefit of all the objects that are now satisfied because of the selection of the new component, and s is the space requirements of the new component. If the selection of no component causes any new objects to be satisfied then the algorithm picks the component with the smallest space requirements.

This algorithm explores the applicability of choosing components instead of objects. We do not expect it to perform well when most of the objects require more than one component in order to get satisfied. The time complexity of this algorithm is $O(|V|^2)$ and we will refer to it as *GrComp (Greedy Component)*.

5.4 Simulated Annealing

Simulated annealing [15] is a randomized hill climbing algorithm. With a certain probability, that declines over time, this algorithm is allowed to follow directions in the solution space that result in solutions worse than those seen so far. This technique enables the algorithm to avoid local minima and stabilize in a final state that is close to optimal.

The simulated annealing algorithm that solves the *COSS* problem is depicted in Figure 5. We start with an initial solution (lines 2 and 3) that is provided by one of the greedy algorithms we presented earlier. Lines 7-16 implement the hill climbing procedure. In simulated annealing apart from uphill moves, downhill moves are also allowed under certain circumstances. More specifically,

```
Objective: Compute a solution for the COSS problem.
Input: A set $U$ of objects, a set $V$ of components, a space constraint $W$, and the bipartite
       graph $G(U,V,E)$, where $e_{ij} \in E$ denotes that component $v_j$ is used by object $u_i$.
Output: A set of components $S \subset V$ to materialize.
1 procedure SolveCOSS()
2   let $O = O_0$;   /* selected objects $O \subset U$ */
3   let $S = S_0$;   /* selected components $S \subset V$, corresponding to $O$ */
4   let $T = T_0$;
5   let $delta = 0$;
6   let $f_{max} = 0$;   /* total benefit of best solution so far */
7   while ($T > 1$)
8     for $i$=1 to total number of iterations
9       $O_{new}$ =GetNewSimAnSolution($O$);
10      let $delta$ =(benefit of $O_{new}$)$-$ (benefit of $O$);
11      if($delta \geq 0$)
12        $O = O_{new}$;   /* accept the new solution */
13      if ($delta < 0$) then with probability $e^{-\frac{delta}{T}}$
14        $O = O_{new}$;   /* accept the new solution */
15      $f_{max} = max$(benefit of $O$, $f_{max}$);
16    $T = \alpha T$;   /* $\alpha < 1$ */
17  $S$ = marginals needed to answer queries in the $O$ corresponding to $f_{max}$;
18  return($S$);

19 procedure GetNewSimAnSolution($O$)
20   let $S$ = marginals needed to answer the queries in $O$;
21   pick object $O_{admit} \in \{U - O\}$ at random, and insert it to $O$;
22   if(total space needed by $S > W$)
23     pick an object $O_{evict} \in \{O - O_{admit}\}$ at random, and remove it from $O$;
24   if (total space needed by $S > W$)
25      penalize the total benefit of $O$ proportionally to the benefit per space ratio of $O_{admit}$
            and the amount by which $W$ is exceeded;
26   return($O$);
```

Fig. 5. The *SimAn* algorithm.

a downhill move is accepted with probability $e^{-(delta)/T}$ (line 13), where *delta* is the decrease in the cost function from the previous iteration, and T is a time-varying parameter controlling the above probability. When T is high (in the beginning of the process) the probability of accepting downhill moves is high. Then, T is slowly decreased (line 16), and when the system freezes ($T < 1$) no further moves are considered. Note that this process involves two loops. While T remains fixed, the inner loop (lines 8-15) searches for solutions. In our implementation the number of iterations executed is a constant number (dependent on the problem size).

The function *GetNewSimAnSolution()* (lines 19-26) determines what the proposed solution for the next iteration of the algorithm is going to be. The next solution is chosen at random among all the neighbours of the current solution. We term two solutions as neighbours if we can derive one from the other by adding a single object to one of the solutions, optionally followed by a deletion of another object. This last step ensures that the current solution satisfies the space constraint (lines 24-25). Nevertheless, the requirement that the current solution should satisfy the space constraint at every step is not strict. In some cases, when the new solution seems promising, we allow it to violate the space constraint at the cost of a small penalty to the total benefit.

The complexity of the simulated annealing approach is determined by the number of iterations. The work done in each iteration is really minimal, and we can safely consider it as constant. The number of iterations is controlled by the user, who sets the parameter T, and defines the way it is reduced. In our experiments the parameter T was initially set to four times the total benefit of the initial solution, the α parameter controlling the decrease of T was set to 0.95, and for each fixed value of T the inner loop of the algorithm executed a number of iterations equal to 1/3 of the total number of objects. Varying the above parameters did not have significant effects on the quality of solutions found.

5.5 Tabu Search

Tabu search [8] is a metastrategy for guiding known heuristics to overcome local optimality. A structure called *tabu list*, used as auxiliary memory, describes a set of moves that are not permitted. This way the algorithm can avoid visiting solutions that has already visited in the past, and thus it is less probable to get stuck in local minima or cycles.

The outline of the tabu search algorithm for the *COSS* problem is shown in Figure 6. The algorithm starts with an initial solution (lines 2 and 3), which in our implementation is derived using the greedy heuristics we presented earlier. Then, the main loop (lines 6-10) iterates over the proposed solutions in order to pick the best one. The function *GetNewTabuSolution()* (lines 13-20) selects the solution (defined in terms of the selected objects) that will be examined during the next iteration of the algorithm. The last step of the function is to make sure that the proposed solution satisfies the space constraint (lines 18 and 19). However, as with simulated annealing, this requirement is not strict.

Note that the changes that lead to the new solution cannot involve any of the objects in the tabu list Q_{tabu}. In our implementation the tabu list keeps track of the recently added or deleted objects. Nevertheless, this restriction can be overridden when the new solution is the best obtained so far. When the new solution is available, we update the tabu list (line 10), and proceed to the next

```
Objective: Compute a solution for the COSS problem.
Input: A set $U$ of objects, a set $V$ of components, a space constraint $W$, and the bipartite
        graph $G(U,V,E)$, where $e_{ij} \in E$ denotes that component $v_j$ is used by object $u_i$.
Output: A set of components $S \subset V$ to materialize.
1 procedure SolveCOSS()
2   let $O = O_0$;    /* selected objects $O \subset U$ */
3   let $S = S_0$;    /* selected components $S \subset V$, corresponding to $O$ */
4   let $f_{max} = 0$;    /* total benefit of best solution so far */
5   let $O_{tabu} = \emptyset$;    /* set of tabu objects */
6   while (search not finished)
7       $O_{new}$ =GetNewTabuSolution($O, O_{tabu}$);
8       $f_{max} = max$(benefit of $O_{new}$, $f_{max}$);
9       $O = O_{new}$;
10      update $O_{tabu}$ based on the changes to $O$;
11  $S$ = components needed to satisfy objects in the $O$ corresponding to $f_{max}$;
12  return($S$);

13 procedure GetNewTabuSolution($O, O_{tabu}$)
14   let $S$ = components needed to satisfy objects in $O$;
15   pick an object $O_{admit} \in \{U - O - O_{tabu}\}$, and insert it to $O$;
16   if(total space needed by $S > W$)
17     pick an object $O_{evict} \in \{O - O_{admit} - O_{tabu}\}$, and remove it from $O$;
18   if (total space needed by $S > W$)
19       penalize the total benefit of $O$ proportionally to the benefit per space ratio of $O_{admit}$
             and the amount by which $W$ is exceeded;
20   return($O$);
```

Fig. 6. The *Tabu* algorithm.

iteration.

Similar to the simulated annealing method, the complexity of tabu search is controlled by the user. An important difference in the case of tabu search is the selection of the next solution. It can be as simple as a random change in the current solution (like simulated annealing), or as involved as exhaustive enumeration. Therefore, there is a tradeoff between the number of iterations the algorithm will execute, and the complexity of each iteration. In our experiments we use a greedy algorithm to select the new solution.

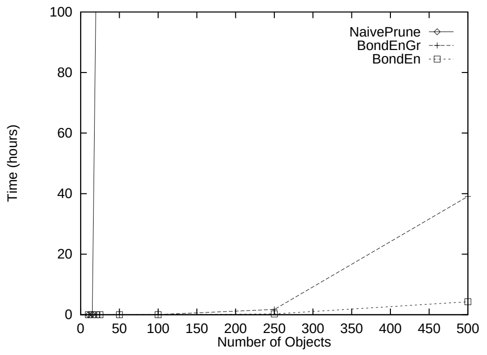
6 Experimental Evaluation

For the evaluation of the efficiency and behavior of the algorithms we use synthetic datasets, where the number of objects range from 10 to 1000. The number of components is in each case twice the number of objects, and more than half of the components help satisfy multiple objects. In order to assign benefit values to objects and space requirements to components we generated random numbers following uniform, Gaussian, and Zipfian distributions.

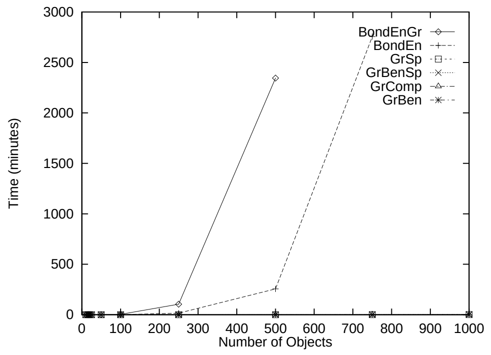
In the experiments we measure the sum of the benefits of the objects that are satisfied given the components that are selected by the algorithms. We are also interested in the computation time of the proposed algorithms. In all cases we report the benefit of the solutions normalized by the total benefit of all the objects in the problem. Similarly, the space constraint is normalized by the total space requirements of all the components in the problem.

6.1 Scalability of the Algorithms

The first set of experiments examines the efficiency of the algorithms in terms of the time required to produce the solution. Figure 7 shows how the run-time of the algorithms changes when the number of objects increases. In Figure 7(a)



(a) *NaivePrune* and *BondEn*



(b) *BondEn* and greedy

Fig. 7. Scalability of the algorithms.

we depict the tremendous difference in the computation time needed by the naive approach and the rest of the algorithms. *NaivePrune*, which is represented by the vertical line at the very left of the graph, is able to produce answers in a reasonable time-frame only for problems involving fewer than 20 objects. As shown in Figure 7(b) the bond energy algorithms scale more gracefully. Nevertheless, when the problem size becomes large, i.e., more than 600 objects for *BondEn* and more than 400 objects for *BondEnGr*, these algorithms require more than 24 hours to produce a solution. Evidently, it is only the greedy algorithms that are able to scale to thousands of objects. The time they required was under 2 minutes in all cases we tested.

In this set of experiments we evaluate the quality of the solutions produced by the proposed algorithms. First we compare the variations of the bond energy family of algorithms. Figure 8 illustrates the benefits for the solutions produced by *BondEn*, *BondEnGr*, *BondEn-SpAll*, and *BondEnGr-SpAll* for various values of the space constraint. The differences among the algorithms are

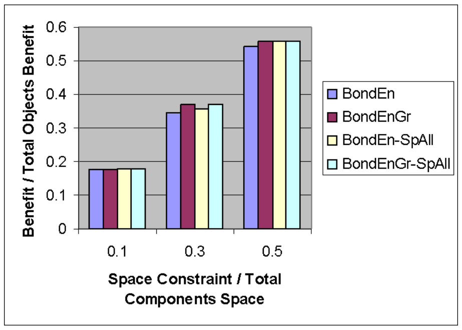
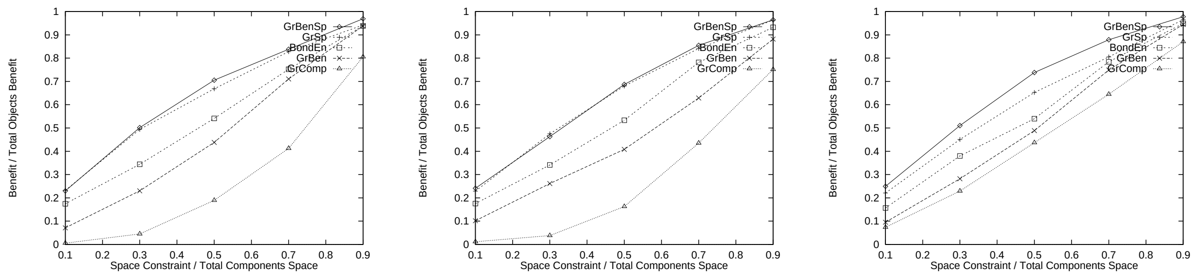


Fig. 8. Performance of the *BondEn* algorithms.

minimal in all the cases we considered. Observe also that the greedy method of constructing the bond energy matrix (*BondEnGr*) in some cases improves the quality of the solutions. However, the more involved split procedure (represented by *BondEnGr-SpAll*) is not able to achieve a better solution than the simpler approach (*BondEnGr*). Since the above algorithms perform without significant differences, for the rest of the experiments we only demonstrate *BondEn*, which is the fastest among them.

In the next set of experiments we compare the quality of the solutions of the bond energy and the greedy algorithms. The graphs in Figure 9 illustrate the normalized total benefit of the solutions when we vary the space constraint, and the graph G remains the same. The two graphs correspond to the cases



(a) uniform distribution (b) Gaussian distribution (c) Zipfian distribution

Fig. 9. Varying the space constraint.

where the assigned object benefits and component space requirements follow uniform, and Zipfian distributions, respectively. Figure 10 depicts the relative ordering of the algorithms in terms of the quality of the solution, when we vary

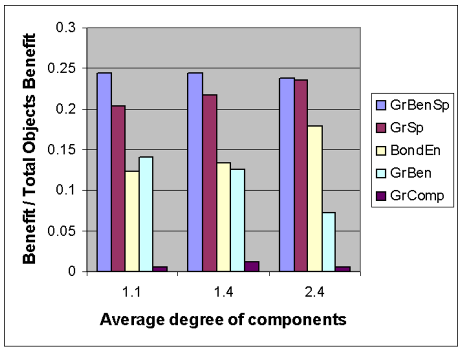
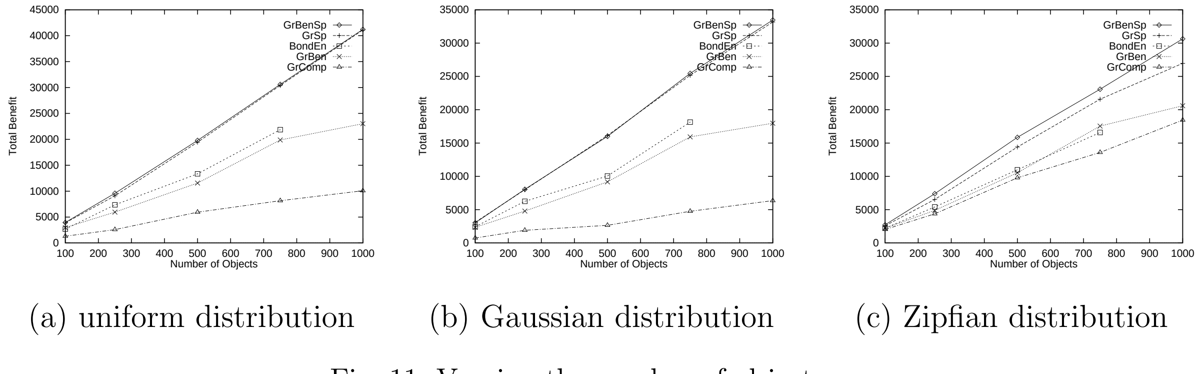


Fig. 10. Varying the graph G.

the graph G. As we move in the graph from left to right there is an increase in the average number of objects that are connected to each component (i.e., we increase the degrees of the component nodes in G). The best performance across all experiments is achieved by *GrBenSp*, closely followed by *GrSp*. The *BondEn* and *GrBen* algorithms perform in the middle range, while *GrComp* performs the worst.

The poor performance of *GrComp* is explained by the nature of the algorithm, which builds the solution one component at a time, instead of sets of components like the rest of the algorithms. This choice restricts *GrComp*, and does not allow it to start accumulating benefit until all the components related to a specific object are brought in the solution. This explains the slow rate at which the algorithm improves the solution at the lower left part of the graph in Figure 9(a).

We also conducted a series of experiments where we varied the number of objects. Figure 11 depicts the total benefit achieved by each algorithm. We report the results of running the algorithms on the same graph G, where the query benefits and the component space requirements were produced from uniform (Figure 11(a)), and Zipfian (Figure 11(b)) distributions. These experiments show that the relative ordering in performance for all the algorithms we consider remains the same across various problem sizes.



(a) uniform distribution (b) Gaussian distribution (c) Zipfian distribution

Fig. 11. Varying the number of objects.

In Table 1 we report the results of the experiments with *SimAn* and *Tabu*. We use the solution provided by *GrBenSp* as the base solution, and report the improvement on this solution achieved by each one of the two algorithms. The initial solution for both *SimAn* and *Tabu* is provided by *GrBenSp*, and this is also the method used by *Tabu* to select solutions at each iteration. We experimented with varying the number of objects and the space constraint, and we allowed the algorithms to run an equal amount of time to the time required by *GrBenSp* to produce the solution.

		GrBenSp	**SimAn**	**Tabu**
objects	*constraint*			
100	*0.5*	1	**1.012**	**1.012**
250	*0.5*	1	1	**1.003**
500	*0.5*	1	**1.003**	**1.003**

(a) Varying the number of objects

		GrBenSp	**SimAn**	**Tabu**
objects	*constraint*			
250	*0.1*	1	1	**1.05**
250	*0.3*	1	1	**1.002**
250	*0.5*	1	1	**1.003**

(b) Varying the space constraint

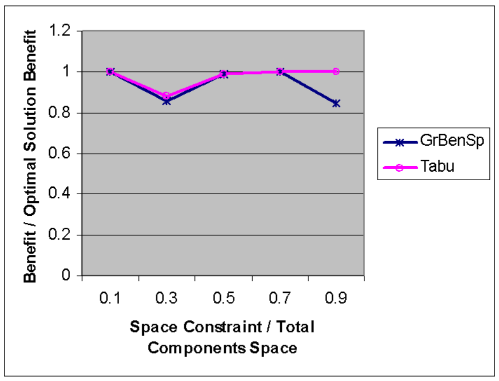
Table 1
Improvement in the *GrBenSp* solution by *SimAn* and *Tabu*.

Unlike *SimAn*, *Tabu* was able to improve on the initial solution in all the cases tested. We believe that the reason *Tabu* outperforms *SimAn* is because the solution space is too large, and this makes it extremely difficult for *SimAn* to head to the correct direction. Remember that *SimAn* moves in the solution space by selecting at random one of the numerous solutions that are neighbors of the current solution. On the other hand, the *Tabu* algorithm directs its search in the solution space more effectively, because it employs a more structured way of taking steps at each iteration.

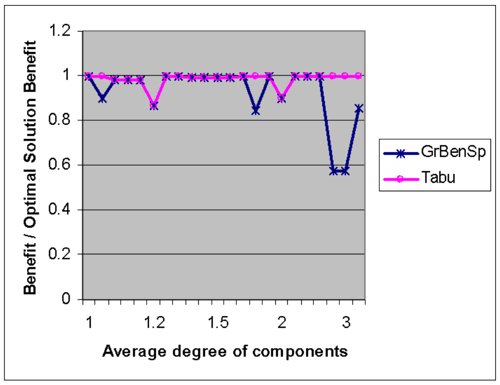
6.2.1 Comparison to Optimal Solution

An interesting observation is the fact that both *SimAn* and *Tabu* improve the solution only by a small amount (less than or equal to 5% for the cases we tested). A natural question then is how close to optimal is the solution provided by *GrBenSp* in the first place. Unfortunately, it is not easy to find the optimal solution for the experiments we presented, because the size of the problem is prohibitively large. Therefore, we conducted a series of experiments where the number of objects was set to 10, for which we could get the optimal solution to compare against the other algorithms. The results of these experiments are depicted in Figure 12. The graphs show the benefit of the solutions produced by *GrBenSp* and *Tabu* normalized by the benefit of the optimal solution (which is always 1). In the graph shown in Figure 12(a) we vary the space constraint. In Figure 12(b) each point in the graph represents an experiment with a different graph G connecting objects to components. As we move in the graph from left to right there is an increase in the average number of objects that are connected to each component. The left-most point in the graph corresponds to the case where each component is connected to a single object, and the problem degenerates to the Knapsack problem.

Note that in many settings in both graphs *GrBenSp* finds a solution very close to optimal. Though, there are also cases where it achieves slightly more than half the benefit of the optimal. These experiments indicate that the *GrBenSp* algorithm is in many circumstances effective at finding near-optimal solutions. This explains the fact that *SimAn* and *Tabu* could not find much better solutions than the greedy algorithm in our previous experiments (see Table 1). Nevertheless, the graphs show that the *Tabu* algorithm is able to improve upon *GrBenSp*, and find a very good solution in almost all the cases where *GrBenSp* performs poorly.



(a) Varying the space constraint



(b) Varying the graph G

Fig. 12. *GrBenSp* and *Tabu* compared to optimal.

7 Discussion

In the following paragraphs, we investigate in more detail some properties of the algorithms. These are properties relevant to the ability of the algorithms to provide high quality solutions under different circumstances. More specifically, we examine the behavior of the proposed solutions when the space constraint is not a strict one. That is, when a solution that uses a little bit more of space is acceptable. Furthermore, we try to quantify the quality of the solutions in some worst case scenarios, when compared to the optimal solution. This analysis indicates how poor the performance of the algorithms can become under certain conditions. Finally, we conclude by summarizing the benefits and drawbacks of each approach, and by providing a practical guide for selecting among the proposed solutions.

7.1 Soft Space Constraints

So far we have made the assumption that the space constraint is a fixed number given in the input of the *COSS* problem, and the algorithm that provides the solution is not supposed to violate this constraint. However, it is not always the case that a strict space constraint is what we are after. Often, the space constraint is merely an indication or approximation of the amount of space that is available. In such cases, we may allow the solutions produced by the algorithms to exceed the space constraint by a small amount. Since we cannot easily quantify what a suitable small amount would be in each case, it is left to the user to examine the solutions, and decide whether the increase in benefit justifies the larger space requirements.

We now describe how we can change the algorithms in order to operate in this environment. The goal is to alter the algorithms so that they incrementally produce solutions which occupy more space and, naturally, have larger total benefit.

Bond Energy: The bond energy algorithms cannot readily provide the new solution, because they have to rerun the *Split* procedure (see Figure 3), which determines which components should be in the solution. Running the *Split* procedure each time we allow more space for the solution is an expensive operation, requiring time $O(|V|^2)$. Each new solution is not necessarily a superset of the previous one, since we select the components from scratch.

Greedy: It suffices to change the condition in line 4 (see Figure 4), which checks whether the solution satisfies the space constraint. Once we remove this condition, the greedy algorithms can produce with linear complexity at each iteration a new solution with increased space requirements. At each step we are only adding new objects or components, thus, we get a solution that is a superset of the solution of the previous iteration.

Simulated Annealing: In this case we only need to remove the conditions in lines 22 and 24 (see Figure 5) that check if the current solution requires space more than the space constraint. Since the process of forming a solution involves a randomization step, there is no guarantee that the new solution will be a superset of the old one.

Tabu Search: Similar to simulated annealing, the only change is to remove the conditions of lines 16 and 18 (see Figure 6). Once again, the solutions produced by the algorithm are not guaranteed to be supersets of the previous solutions.

In the following section, we focus on the greedy algorithms. More specifically, we examine the performance of these algorithms (in terms of the quality of solutions) in relation to the performance of the greedy solution for the *Knapsack* problem.

7.2 On the Optimality of the Greedy Solutions

As we have already discussed, the *COSS* problem degenerates to the *Knapsack* problem when each component satisfies at most one object. In this case, we can consider the set of components required by each object as a single super-component, and attach to it a space requirement equal to the sum of the space requirements of the corresponding components.

We can also show that the following theorem holds.

Theorem 1 *Consider the* Knapsack *problem with n objects. Let the space constraint W be equal to the space needed by the i objects, $1 \leq i \leq n$, with the highest benefit per space ratio. In this case the greedy algorithm that chooses items based on their benefit per space ratio finds the optimal solution.*

Proof: Figure 13 illustrates a linear ordering of the n objects according to their benefit per space values. We name u_1 the object that has the highest

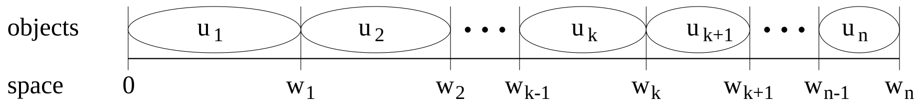


Fig. 13. A linear ordering of n objects for the *Knapsack* problem.

benefit per space ratio, and u_n the object that has the lowest. This linear ordering represents the order in which the greedy algorithm will select objects to insert in the solution. Assume that we set the space constraint to $W = w_{k+1}$, which is the total space required by the first $k + 1$ objects. Then we solve the *Knapsack* problem using this space constraint. We know that the greedy algorithm that chooses items based on their benefit per space ratio is optimal (in benefit) for the *fractional Knapsack* problem [3]. Observe though, that in our case the space constraint is carefully set so that all the selected objects will fit exactly in the available space, and none will need to be divided. (Remember that the greedy algorithm will select the objects from left to right, in the order they appear in Figure 13.) Note that the same is true for all the choices of

$W = w_i$, where $1 \leq i \leq n$. Therefore, the greedy algorithm that chooses items based on their benefit per space ratio is also optimal for the *Knapsack* problem when the space constraint is set to w_i, where $1 \leq i \leq n$. □

This theorem is interesting in our context. It says that, when we can relax the space constraint, we can solve the *Knapsack* problem using the greedy benefit per space algorithm, stop after selecting any number of items, and we are guaranteed that the solution we have is optimal for the amount of space used. Given the above analysis, a question that arises naturally is whether this optimality property carries over to the *COSS* problem and the corresponding algorithm *GrBenSp*.

Unfortunately, the answer is negative. The reason is that unlike *Knapsack*, in the *COSS* problem each component may help satisfy more than one object. The result of these interdependencies is that a solution that is optimal for some space constraint W_1 is not necessarily a subset of the optimal solution for a space constraint $W_2 > W_1$. Consequently, there is no linear ordering of the objects in the sense depicted in Figure 13. Hence, the optimality property of the greedy algorithm does not hold for the *COSS* problem. Theorem 2 formally states this observation.

Theorem 2 *The* COSS *problem does not have the optimality property. That is, the optimal solution for some space constraint W_1 is not necessarily a subset of the optimal solution for a space constraint $W_2 > W_1$.*

Proof: We prove this theorem using a counter-example. Consider the instance of the *COSS* problem depicted in Figure 14. We indicate the benefit of each object and the space requirements for each of the components by the numbers in parentheses below the names of the objects and components. Assume that the space constraint is $W_1 = 12$ units. Then, the optimal solution for this particular instance of the *COSS* problem is the set of objects $O_1 = \{u_3, u_4\}$ with total benefit 17 units. Now let the space constraint be $W_2 = 16$ units. In this case, the optimal solution is composed of the set of objects $O_2 = \{u_1, u_2, u_3\}$ with total benefit 24 units. We observe that the optimal solution O_2 is *not* a superset of O_1, even though the space constraint W_2 is greater than W_1. □

7.3 Analysis of the Greedy Algorithms

In this section, we present theoretical results on the behaviour of the greedy algorithms when compared to optimal. These results are interesting, because they indicate what the worst performance of the algorithms might be, and they can help us choose among those algorithms.

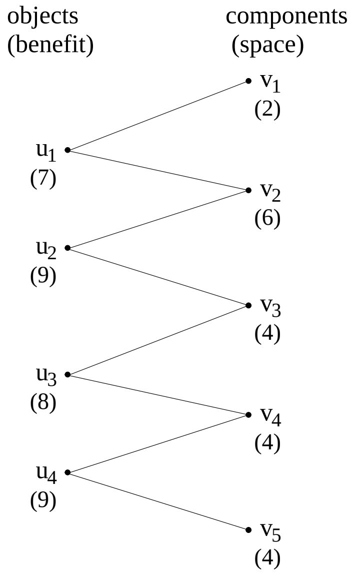


Fig. 14. Counter-example that demonstrates the fact that for the *COSS* problem, the optimal solution for one space constraint value is not necessarily a subset of the optimal solution for larger space constraint values. The numbers in parentheses indicate the benefit and required space of the objects and components respectively.

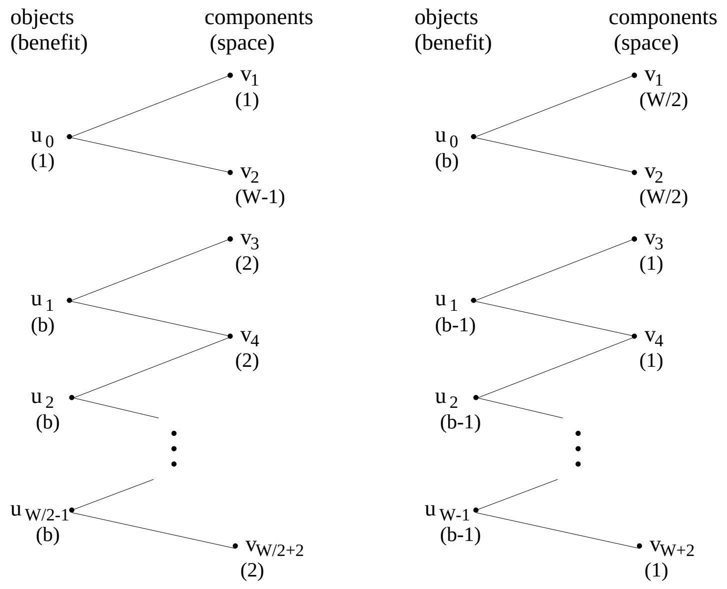
In Figures 15 and 16 we illustrate examples where the algorithms perform poorly in terms of quality of the solution. The benefit of each object and the space requirements for each of the components are indicated by the numbers in parentheses below the names of the objects and components. In all cases b represents a measure of the objects' benefit, while the space constraint is set to W units.

Theorem 3 *In the worst case, the* GrComp *algorithm provides a solution that is at least* $(\frac{W}{2}-1)b$ *times worse than the optimal solution, for any* $b > 0$ *and* $W > 3$.

Proof: In the instance of the *COSS* problem depicted in Figure 15(a) *GrComp* will first select component v_1, because the selection of no component will yield any benefit, and v_1 has the minimum space requirements among all the components. Subsequently, it will select component v_2 in order to get the benefit from satisfying object u_0, for a total benefit of 1. After that there are no more objects that can be satisfied, since the total space required by the set of the selected components, i.e., $\{v_1, v_2\}$, is equal to the space constraint W. On the contrary, the optimal algorithm will select the components $\{v_3, \ldots, v_{W/2-1}\}$ ($W/2$ components in total), for a total benefit of $(\frac{W}{2}-1)b$. Therefore, *GrComp* can be $(\frac{W}{2}-1)b$ times worse than the optimal solution. $\square$

Theorem 4 *In the worst case, the* GrBen *algorithm provides a solution that is at least* $(W-1)(1-\frac{1}{b})$ *times worse than the optimal solution, for any* $b > 0$ *and* $W > 2$.

Proof: In Figure 15(b) we give an example for *GrBen*. In this case the algorithm will accept in the solution object u_0, because it has the highest benefit value among all the objects. The space required by u_0, or equivalently $\{v_1, v_2\}$, is W. Then the algorithm terminates, since it will have used up all the available space. The benefit for this solution is b. The optimal solution includes the objects $\{u_1, \ldots, u_{W-1}\}$, for a total benefit of $(W-1)(b-1)$. This means that the solution of *GrBen* can be $(W-1)(1-\frac{1}{b})$ times worse than the optimal solution. □



(a) *GrComp* (b) *GrBen*

Fig. 15. Example scenarios for *GrComp* and *GrBen*.

Theorem 5 *In the worst case, the* GrSp *algorithm provides a solution that is at least* $1.33b$ *times worse than the optimal solution, for any* $b > 0$ *and in the limit as* $W \to \infty$.

Proof: For the *GrSp* algorithm consider the scenario depicted in Figure 16(a). The algorithm will select the objects $\{u_1, \ldots, u_{W/2}\}$. The components needed to satisfy these objects have the lowest space requirements, i.e., they occupy space of 2 units per object. In contrast, all the rest of the objects require space of 3 units each. The selected objects are $W/2$ altogether, they yield a total benefit of $W/2$, and the corresponding components occupy all the available space W. The optimal solution in this case is comprised of the objects $\{u_{W/2+1}, \ldots, u_{W/2+2W/3-1}\}$. The number of the selected objects is $(\frac{2W}{3}-1)$, and their total benefit is $(\frac{2W}{3}-1)b$. Thus, *GrSp* can be $\frac{(2W/3-1)}{W/2}b$ times worse than optimal. For sufficiently large values of W the above quantity is approximated by $1.33b$. □

Theorem 6 *In the worst case, the* GrBenSp *algorithm provides a solution that is at least* 1.77 *times worse than the optimal solution, in the limit as* $b \to \infty$.

Proof: Consider the example illustrated in Figure 16(b), for which we assume that the space constraint is $W = 2b$. During the first iteration the *GrBenSp* algorithm will select object u_3, because it has the largest ratio of benefit per space required by the corresponding components, which are $\{v_3, v_4\}$. This ratio is 1 for u_3, and less than 1 for all the other objects.

In the next iteration the object u_4 is selected. Note that in order to satisfy u_4 only the component v_5 is needed, since v_4 was selected in the previous iteration. The object u_4 is selected because its benefit per space ratio, $\frac{5b+2}{3b}$, is higher than the one of u_2, which requires component v_2, and has a ratio of $\frac{5b-4}{3b}$. The other two choices have lower ratios, as well. The object u_1, which requires components $\{v_1, v_2\}$, has a ratio of $\frac{b-1}{b}$, and object u_5, which requires components $\{v_5, v_6\}$, has a ratio of $\frac{1}{b}$.

In the third iteration the algorithm selects object u_5, which now only needs component v_6, since v_5 is already selected. Observe that the benefit per space ratio of this choice ($\frac{2}{b}$) is not better than the ration of u_1 ($\frac{b-1}{b}$). However, it is the only viable choice at this point, because the remaining available space is just $b/2$.

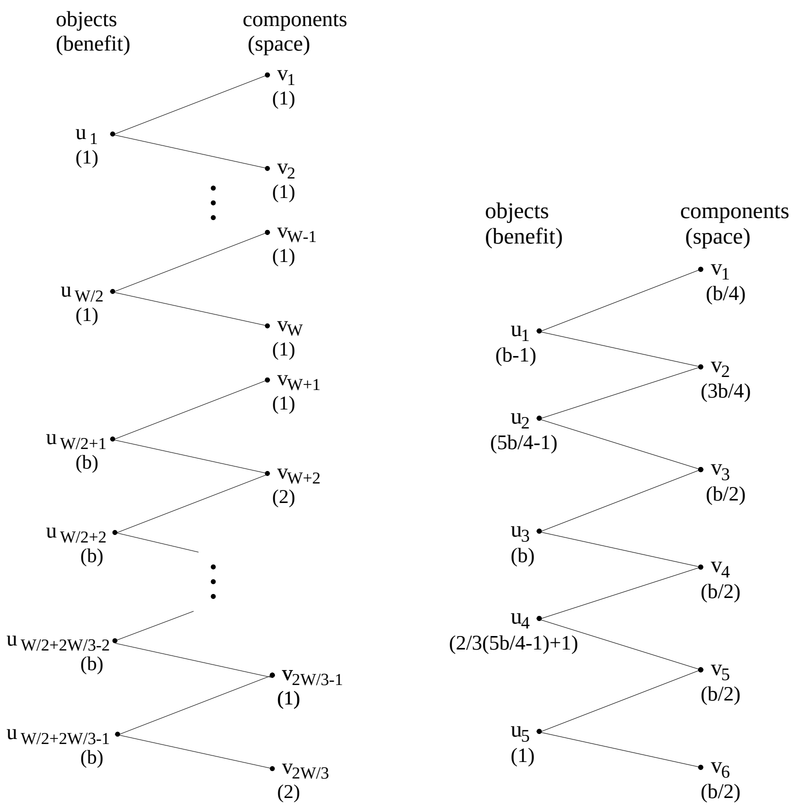
In summary, the *GrBenSp* algorithm selects the set of objects $\{u_3, u_4, u_5\}$, and the corresponding components $\{v_3, v_4, v_5, v_6\}$, which occupy space $2b = W$. The total benefit of this solution is $\frac{11b+8}{6}$.

The optimal solution for this example consists of the set of objects $\{u_1, u_2, u_3\}$. The required components occupy all the available space W, and the total benefit of the solution is $\frac{13b-8}{4}$.

Therefore, the solution of *GrBenSp* can be $\frac{39b-24}{22b+16}$ times worse than optimal. For sufficiently large values of b the above quantity is approximated by 1.77. □

7.4 Choosing Among the Alternatives

The experiments demonstrate that the bond energy algorithms perform worse than some of the greedy approaches. At a first glance this is a rather surprising result, given that the *BondEn* algorithms have higher complexity and seem to make choices with greater care. However, we observe that all the decisions that *BondEn* makes are based on just *pairs* of components, despite the fact



(a) *GrSp* (b) *GrBenSp*

Fig. 16. Example scenarios for *GrSp* and *GrBenSp*.

that in many cases it is a larger number of components that are interrelated. The experiments show that this prevents *BondEn* from capturing the true, more complex, associations inherent in the problem, narrows the information upon which the algorithm operates, and leads to poor decisions. Thus, *BondEn* should not be the algorithm of choice, especially since it does not scale as well as the greedy algorithms.

The greedy approaches are better than the bond energy algorithms for another reason as well. They are able to incrementally compute a new solution when the space constraint is relaxed. This is an important factor in the selection of the algorithm, since it removes the requirement of a hard space constraint, and gives the user the flexibility to choose among a range of slightly different space requirements which may lead to solutions with significant variations in the benefit values. On the contrary, the bond energy algorithms cannot readily provide the new solution, because they have to rerun the split procedure (see Figure 3), which is expensive.

The distinction among the greedy algorithms is quite clear between the best (*GrBenSp* and *GrSp*) and the worst (*GrBen* and *GrComp*) performers (see Figures 8(b) and 9(a)). However, it is not clear from the experiments whether *GrBenSp* or *GrSp* is a better choice, since they both have the same time complexity and provide solutions of similar quality.

In order to answer the above question we have to turn our attention to the analysis presented in Section 7.3. This analysis shows that three of the greedy approaches we examined, including *GrSp*, may perform arbitrarily poorly under certain circumstances. The same does not seem to be true for the *GrBenSp* algorithm. We believe that *GrBenSp* is a better choice in this sense, since it avoids making poor decisions that lead to solutions very far from optimal.

Finally, we should note that *Tabu* can in many cases improve on the solution provided by the greedy algorithms. An interesting observation is that when we turned off the tabu list functionality the performance of the algorithm deteriorated. This indicates that *Tabu* is indeed making well-calculated moves, and that the tabu list enables the algorithm to avoid local minima and explore new areas in the solution space. Therefore, it is beneficial to run this algorithm when the time allows it.

8 Conclusions

Space constrained optimization problems are still significant despite the advances in storage technologies. In this paper, we focused on the specific problem of *COSS*, where benefit values are associated to *sets* of items, instead of individual items. We derived the complexity of this problem, and since there are no known approximation algorithms for these problems, we explored the use of greedy and randomized techniques.

We presented an extensive experimental evaluation of the algorithms that illustrates the relative performance of the different approaches, and demonstrates the scalability of the greedy solutions. Finally, we examined some properties of the algorithms from a theoretical perspective, and presented a worst-case analysis. The results of this analysis can be useful in practice for choosing among the alternatives. Both our experimental and theoretical analysis demonstrate that *GrBenSp* is a practical solution for the *COSS* problem.

References

[1] E. Baralis, S. Paraboschi, and E. Teniente. Materialized Views Selection in a Multidimensional Database. In *VLDB*, pages 156–165, Athens, Greece, Sept. 1997.

[2] M. Cherniack, M. J. Franklin, and S. B. Zdonik. Data Management for Pervasive Computing. In *VLDB*, page Tutorial, Rome, Italy, Sept. 2001.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* New York: McGraw-Hill, 2001.

[4] A. Deshpande, M. N. Garofalakis, and R. Rastogi. Independence is Good: Dependency-Based Histogram Synopses for High-Dimensional Data. In *ACM SIGMOD*, pages 199–210, Santa Barbara, CA, USA, May 2001.

[5] C. Faloutsos, H. V. Jagadish, and N. Sidiropoulos. Recovering Information from Summary Data. *VLDB, Athens, Greece*, pages 36–45, Aug. 1997.

[6] M. Garey and D. Johnson. *Computers and Intractability*. W. H. Freeman, 1979.

[7] P. E. Gill, W. Murray, and M. H. Wright. *Practical Optimization*. New York: Academic Press, 1981.

[8] F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, 1997.

[9] H. Gupta. Selection of Views to Materialize in a Data Warehouse. In *ICDT International Conference*, pages 98–112, Delphi, Greece, Jan. 1997.

[10] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing Data Cubes Efficiently. In *ACM SIGMOD*, pages 205–216, Montreal, Canada, June 1996.

[11] Y. E. Ioannidis and E. Wong. Query Optimization by Simulated Annealing. In *ACM SIGMOD*, pages 9–22, San Francisco, CA, USA, May 1987.

[12] D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon. Optimization by Simulated Annealing: An Experimental Evaluation. *Operations Research*, 37(6):865–892, 1989.

[13] H. J. Karloff and M. Mihail. On the Complexity of the View-Selection Problem. In *ACM PODS International Conference*, pages 167–173, Philadelphia, PA, USA, May 1999.

[14] R. Kimball. *The Data Warehouse Toolkit: Practical Techniques for Building Dimensional Data Warehouses*. John Wiley & Sons, 1996.

[15] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, 1983.

[16] J. Kleinberg. Personal Communication. 2002.

[17] J. M. Kleinberg, C. H. Papadimitriou, and P. Raghavan. Segmentation Problems. In *ACM Symposium on the Theory of Computing*, pages 473–482, Dallas, TX, USA, May 1998.

[18] W. T. McCormick, P. J. Schweitzer, and T. W. White. Problem Decomposition and Data Reorganization by a Clustering Technique. *Operations Research*, 20(5):993–1009, 1972.

[19] S. Navathe, S. Ceri, G. Wiederhold, and J. Dou. Vertical Partitioning Algorithms for Database Design. *ACM Transactions on Database Systems*, 9(4):680–710, 1984.

[20] S. B. Navathe and M. Ra. Vertical Partitioning for Database Design: A Graphical Algorithm. In *ACM SIGMOD*, pages 440–450, Portland, OR, USA, May 1989.

[21] D. Nehme and G. Yu. The Cardinality and Precedence Constrained Maximum Value Sub-Hypergraph Problem and its Applications. *Discrete Applied Mathematics*, 74:57–68, 1997.

[22] T. Palpanas and N. Koudas. Entropy Based Approximate Querying and Exploration of Datacubes. In *International Conference on Scientific and Statistical Database Management*, pages 81–90, Fairfax, VA, USA, July 2001.

[23] E. Pitoura and P. K. Chrysanthis. Caching and replication in mobile data management. *IEEE Data Eng. Bull.*, 30(3):13–20, 2007.

[24] S. Rajagopalan and V. V. Vazirani. Primal-Dual RNC Approximation Algorithms for Set Cover and Covering Integer Programs. *SIAM Journal on Computing*, 28(2):525–540, 1998.

[25] S. Sarawagi, R. Agrawal, and N. Megiddo. Discovery-driven Exploration of OLAP Data Cubes. In *EDBT*, pages 168–182, Valencia, Spain, Mar. 1998.

[26] A. Shukla, P. Deshpande, J. F. Naughton, and K. Ramasamy. Storage Estimation for Multidimensional Aggregates in the Presence of Hierarchies. In *VLDB*, pages 522–531, Mumbai (Bombay), India, Sept. 1996.

[27] D. Theodoratos and T. K. Sellis. Data Warehouse Configuration. In *VLDB*, pages 126–135, Athens, Greece, Sept. 1997.

[28] J. Yang, K. Karlapalem, and Q. Li. Algorithms for Materialized View Design in Data Warehousing Environment. In *VLDB*, pages 136–145, Athens, Greece, Sept. 1997.

[29] D. Zhang, G. Karabatis, Z. Chen, B. Adipat, L. Dai, Z. Zhang, and Y. Wang. Personalization and visualization on handheld devices. In *SAC*, pages 1008–1012, 2006.