Covering a set of points with fixed size hypersquares and related problems

T.F. Gonzalez

RUU-CS-90-32
October 1990

Covering a set of points with fixed size hypersquares and related problems

T.F. Gonzalez

Technical Report RUU-CS-90-32
October 1990

Department of Computer Science
Utrecht University
P.O.Box 80.089
3508 TB Utrecht
The Netherlands

ISSN:0924-3275

COVERING A SET OF POINTS WITH FIXED SIZE HYPERSQUARES AND RELATED PROBLEMS

by

Teofilo F. Gonzalez‡
Department of Computer Science
Utrecht University
the Netherlands

ABSTRACT: Let $P = \{p_1, p_2, ..., p_n\}$ be a set of points in d-space. We study the problem of covering with the minimum number of fixed size orthogonal hypersquares (CS_d for short) all points in P. We present an improved polynomial time approximation scheme and fast approximation algorithms that generate provably good solutions to these problems quickly. A variation of the CS_d problem is the CR_d, covering by fixed size orthogonal hyperrectangles, where the covering of the points is by hyperrectangles with dimensions $D_1, D_2, ..., D_d$ instead of hypersquares of size D. Another variation is the CD_d problem in which we cover the set of points with hyperdiscs of diameter D. Our algorithms can also be generalized to handle these two problems.

KEYWORDS: d-space, covering by hypersquares, hyperdiscs and hyperrectangles, efficient approximation algorithms, polynomial time approximation scheme.

‡ On Sabbatical leave from the University of California, Santa Barbara.

September 20, 1990

I. INTRODUCTION.

Let $P = \{p_1, p_2, \ldots, p_n\}$ be a set of points in the plane (E^2). Point p_i is located at $(x_1(p), x_2(p))$, and we assume that $x_j(p_i) \geq 0$, for all i and j. The problem of covering with fixed size orthogonal squares, CS_2, consists of finding a minimum cardinality set of D by D squares covering all points in P, i.e., each point in P must be inside or on the boundary of one of the squares in the cover. A generalization to d dimensions (the set of points belongs to d-dimensional space and instead of covering the points with squares one uses orthogonal hypersquares of dimension D) of the CS_2 problem is called the CS_d problem. We define I as the smallest hyperrectangle orthogonal to the axes that includes all points in P and assume the hyperrectangle includes the origin, otherwise the points can be translated. We use $I_1, I_2, \ldots, I_d$ to represent its dimensions. A variation of the CS_d problem is the CR_d, covering by fixed size hyperrectangles, where the covering of the points is by orthogonal hyperrectangles with dimension $D_1, D_2, \ldots, D_d$ instead of hypersquares of size D. Another variation is the CD_d problem in which we cover the points with hyperdiscs of diameter D. A related problem is that of packing squares is discussed in section V. In what follows when we refer to a square (rectangle) or hypersquare (hyperrectangle) we assume it is orthogonal to the coordinate axes.

These problems have many interesting applications ([FPT], [HM]). The most popular application is the problem of locating the least number of emergency facilities such that all potential users are located within a reasonable small distance from one of the facilities. This corresponds to the CD_2 problem. The CD_d, CR_d and CS_d problems for $d \geq 2$ are known to be NP-hard ([FPT], [MIH] and [S]). Johnson [J] discusses several variations of these problems. Heuristics to solve the CS_2 problem have been presented in [T] and [TF]. A polynomial time approximation scheme is given in [HM], i.e., for every constant $\varepsilon > 0$ they give an algorithm that generates solutions such that $F_{apx} / F_{opt} \leq 1 + \varepsilon$ with time complexity $O(n^{O(1/\varepsilon)})$.

For any integer $l \geq 1$, the algorithm for the CS_d problem given in [HM] has time complexity bound $O(l^d \; n^{dl^d+1})$ and the approximation bound is $F_{apx} / F_{opt} \leq (1 + 1/l)^d$. It is important to note that the above bound disagrees with the one in [HM] because there is a typo in that paper. To achieve an approximation bound of 2^d it takes $O(n^{d+1})$ time. To achieve an approximation bound of 2.25, for $d = 2$, it takes $O(n^9)$ time. The only approximation bound that can be guaranteed in practical situations is the bound of 2^d when d is small, since to guarantee a solution within 2^{d-1} the worst case case time complexity bound cannot even be bounded by $O(n^9)$ when $d = 2$.

For the CD_d problem and any integer $l \geq 1$ the algorithm in [HM] has worst case time complexity $O(l^d \; (l \sqrt{d})^d \; (2n)^{d\lceil l\sqrt{d}\rceil^d+1})$ and the approximation bound is $F_{apx} / F_{opt} \leq (1 + 1/l)^d$. For $d = 2$, to guarantee solutions within 4 of optimal, the worst case time complexity bound is $O(n^9)$ and a bound of 2.25 the time complexity is $O(n^{19})$. These large time complexity bounds make the algorithms unusable even when n is small.

In this paper we present fast algorithms for the CS_d problem with worst case approximation bound of 2^d and 2^{d-1}. We also discuss several implementations of these algorithms. The best of the implementations that requires only $O(n)$ space, takes $O(dn + n \log s)$ time, where s is a lower bound on the number of hypersquares in an optimal solution. The constant associated with the time and space complexity bounds are very small, thus usable in practical situations. We also present an efficient algorithm for the l_d-slab problem (which we define in section III). This algorithm can be easily combined with the polynomial time approximation scheme given in [MH] to generate a solution to the CS_d problem with approximation bound $(1 + 1/l)^{d-1}$ in $O(l^{d-1} d(2l^{d-1}-1)n^{d(2l^{d-1}-1)+1})$ time. Our new algorithms can also be adapted to the CD_d and the CR_d problems as well as the packing problem discussed in [MH]. The approximation and time complexity bounds for the CD_d problem are not identical to the ones for the CS_d problem; however, they are improvements over the ones for previous algorithms. We discuss these extensions in section IV. The approximation algorithms are presented in section II and the approximation scheme is given in section III.

II. APPROXIMATION ALGORITHMS FOR THE CS_d PROBLEM

In this section we present several new approximation algorithms for the CS_d problem. The worst case approximation bound of these algorithms is 2^d and 2^{d-1}. We also discuss several implementations for these algorithms. The best of the implementations that requires only $O(n)$ space takes $O(dn + n \log s)$ time, where s is a lower bound on the number of hypersquares in an optimal solution. The constants associated with the time and space complexity bounds are very small, thus usable in practical situations. This section is divided into four subsection. In each of these subsections we present a different approximation algorithm. Let us now discuss the first approximation algorithm and the different implementations for its basic operations. None of these algorithms performs better than the other ones on all problem instances. Each of the algorithms has a domain in which it outperforms the other algorithms. This is why we present all the algorithms together with their different implementations.

(A) MAXIMAL SET OF INDEPENDENT POINTS APPROACH.

Before presenting our algorithm based on finding a maximal set of independent points, we define some useful terms. Points p_i and p_j (in P) are said to be *independent* if no hypersquare (with side size D) can cover both of them. Otherwise, we say that point p_i is *dependent* on point p_j or vice-versa. A set $S \subseteq P$ is said to be a *maximal set of independent points* if every pair of distinct points in S is independent and every point in P is dependent on at least one point in S. Whenever we refer to S, a maximal set of independent points, we use s to denote its cardinality.

The idea behind our algorithm is to find a maximal independent set of points. Since no two points in an independent set of points S may be covered by the same square, we know that $F_{opt} \geq |S|$. For the case when $d = 2$, the set of points dependent on each point in set S can be covered by four squares (see figure 1). From this set of squares one can easily delete a maximal set of useless squares, i.e., squares that contain points each of which is inside at least another square. Therefore for $d = 2$, $F_{apx} \leq 4\,|S|$. Combining these two bounds, we know that $F_{apx} / F_{opt} \leq 4$. For $d > 2$ the approach is similar, but to cover all the points dependent on a point in set S one introduces 2^d hypersquares. Figure 1 shows the cover for the case when $d = 3$. Our first approximation algorithm (MS) is given below.

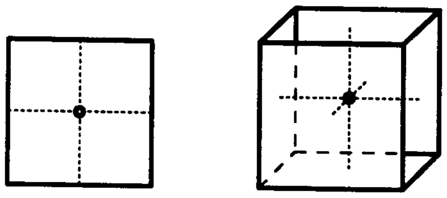

Figure 1: Covering of all points dependent on a point p.

```
Algorithm MS ( $P = \{p_1, p_2, ..., p_n\}, D$ )
 $S \leftarrow \emptyset$;
 for $i = 1$ to $n$ do
   if $p_i$ is independent from every point in $S$ then $S \leftarrow S \cup \{p_i\}$;
 endfor
 Cover all points dependent on each point in $S$ using the rule given in figure 1;
end of algorithm MS
```

In what follows we discuss several methods to implement the test condition in the **if** statement. Let us consider first a straight forward implementation of the test condition. Testing whether a point is independent from all other points in set S can be accomplished by comparing it against each point in set S. Each test can be accomplished by performing $O(d)$ arithmetic operations and comparisons. Therefore, the overall time complexity for procedure MS is $O(d\ s\ n)$.

The above implementation of the algorithm does not take advantage of properties of the set of points in S. Let us now consider these properties in more detail. We begin with the case when $d = 2$. For any point $p \in S$, all points located at a distance D along each axis from it are dependent on point p. Therefore, the test "if q is independent from each point in S" can be reduced to testing whether or not a point is inside a set of squares with dimensions $2D$ by $2D$, where the center of each of these squares is a point in set S. These squares are called *t-squares (target squares)*. The set of *t-squares* is not static, as the algorithm progresses the number of *t-squares* increases. One important property of the *t-squares* is that all of them have identical dimensions ($2D$ by $2D$). Another important property is the fact that the center of a *t–square* is never inside another *t–square*.

Partition the positive quadrant uniformly into squares of size D by D in such a way that a square has its bottom-left corner located on the origin. The squares are called *g–squares (grid squares)*. We refer to each *g–square* by a tuple (i,j) for $i,j \geq 0$ (from left to right and bottom-up, with the bottom-left square labeled (0,0)). Since $x_j(p_i) \geq 0$, the function $i_j(p_i)$ defined as $\lfloor x_j(p_i)/D \rfloor$ determines the *g–square* where point p_i belongs. We use $int(p_i)$ to identify the *g–square* where p_i belongs. Since the *t–squares* have dimensions $2D$ by $2D$ and no center of a *t–square* is inside another *t–square*, there is at most one center of a *t–square* inside a *g–square*. I.e., the maximum number of independent points in each *g–square* is at most one. Determining whether a point p is inside a *t–square* can be tested as follows. First we find the *g–square* $(i_1(p), i_2(p))$ to which it belongs. Then, we construct the set S' of all independent points in S that belong to neighboring *g–squares*, i.e., belong to *g–square* (i'_1, i'_2) such that $|i'_1 - i_1(p)| \leq 1$ and $|i'_2 - i_2(p)| \leq 1$. Clearly, for $d = 2$, there are at most $3^2 = 9$, and in general there are at most 3^d of such neighbors. Since the dimension along each axis of each of the neighboring *g–squares* is D, it then follows that p is independent from each *t–square* iff p is independent from each *t–square* whose center is in set S'.

Let us now define the abstract data type RD. The ADT consists of the set T of d-dimensional integer points. The operations to be performed on T are:

INSERT(p)	[add a point to the set of points T],
DELETE(p)	[delete a point from the set of points T],
MEMBER(p)	[test whether or not p is in the set T], and
RANGE-1(p)	[list all points in T that differ from p along each axis by at most one unit].

Let us now rewrite our algorithm in terms of the above ADT.

```
Algorithm NMS ( $P = \{p_1, p_2, ..., p_n\}, D$ )
$S \leftarrow \emptyset$;
for $i = 1$ to $n$ do
  $S' \leftarrow$ RANGE-1($int(p_i)$)
  if $p_i$ is independent from each point associated with a point in $S'$ then
            associate $p_i$ with $int(p_i)$;
            INSERT($int(p_i)$) in $S$;
endfor
Cover all points dependent on each point in $S$ using the rule given in figure 1;
end of algorithm NMS
```

Let us now consider several implementations for ADT *RD*. If we represent S by a sequential list, then RANGE-1 can be easily implemented to take $O(d\ s)$ time and INSERT takes $O(1)$ time. Therefore, the overall time complexity bound is $O(d\ s\ n)$. We also refer to this implementation as the *straight forward*.

Gonzalez [G] discusses several implementations for the d-dimensional dictionary ADT (*RD* without RANGE-1 queries). Each d-dimensional dictionary operation takes $O(d + \log m)$ time, where m is the number of elements in the set. Any implementation for such ADT can be used for *RD* by simulating each RANGE-1 operation with either 3^d MEMBER operations or s element comparisons. Using Gonzalez' [G] implementation (or an equivalent implementation [G]) for d-dimensional dictionaries results in an algorithm with time complexity $O(\min\{3^d, s\}\ (d + \log s)\ n)$. We call this implementation the *balanced tree* implementation of algorithm *NMS*.

When there is a large amount of memory we may use the double-array representation suggested by Aho, Hopcroft and Ullman [AHU] (exercise 2.12, page 71), in which a set of t integers in the range $[1,w]$ can be represented by two arrays; one of size t and the other of size w; and determining whether an element is present in the set takes constant time. In this case the time complexity bound can be reduced as follows. The set S is represented by a double-array with the d-dimensional elements compressed into a single integer by row-major (or column-major) order. We assume that the maximum value along each dimension is known, otherwise it can be easily computed. A simple formula computable in $O(d)$ time can perform the mapping. The total time reduces to $O(\min\{3^d, s\}\ d\ n)$. It is important to note that the amount of space required in this implementation may be very large ($O(\prod \lceil l_j/D \rceil)$). We call this implementation the *double-array* implementation of algorithm NMS.

One may reduce the additional space at the expense of increasing the time. First let us define the term "connected components". Then we explain how "pseudo connected components" may be efficiently constructed. Define the graph $G = (N, E)$, where each node in N represents a point in set P and for $i \neq j$ there is an edge from n_i to n_j iff the distance from p_i to p_j is at most D along each of the d axes. The term connected component is the same as the one in graph theory, i.e., two nodes belong to the same connected component iff there is a path from one node to the other in G. It is important to note that if points p_i and p_j belong to different connected components, then they are independent. Therefore, one can solve the problem on each component independently. The straight forward implementation of the above procedure has worst case complexity $\Omega(d\ n^2)$ time.

Since finding the set of connected components is as hard as solving our original problem, we construct a set of "pseudo" connected components which can be computed efficiently and it will help the algorithm as much as the connected components. We shall define this set by construction. The first algorithm transforms each point to a tuple that identifies the *g-square* where it is located. The procedure then sorts the points along the first component and partitions them whenever consecutive points are not identical or adjacent integers. It then sorts along the second component each of the resulting sets, further refining the sets. The final partition of the set of points (nodes) are the *pseudo connected components*. This algorithm takes $O(d\ n\ log\ n)$ time. We call this algorithm *PCCS (pseudo connected components by sorting)*. The second algorithm, *PCCDA (pseudo connected components by double arrays)*, finds the connected components without sorting the elements, but by following a procedure similar to PCCS. The partition along the first dimension is obtained by using the double-array structure. In the first pass each point in P is added to its corresponding bin. In the second pass we visit each point (which has not been marked) and retrieve all points in adjacent bins. To avoid reporting elements more than once, whenever we retrieve an element we mark it. It is simple to show that the time complexity of the above procedure is $O(d\ n)$ and the amount of space even though could be very large it is smaller than the implementation called double-array. The amount to space is $O(\max\lceil l_j/D \rceil)$.

Once the set of points has been partitioned into pseudo connected components, by using a double-array representation one can solve the original problem in $O(d\ \min\{3^d, s\}\ n)$ time. The amount of space is bounded by $O(n^d)$. Table I summarizes our algorithms. It is worth while to mention that by preprocessing the data, for example partitioning along the largest gap, one may further reduce the amount of space.

Table I. Time and Space Complexity implementations of algorithm NMS.

implementation	time complexity	space complexity
straight forward	$O(d\ s\ n)$	$O(n)$
balanced trees	$O(min\{3^d, s\}\ (d + log\ s)\ n)$	$O(n)$
double-array	$O(\ d\ min\{3^d, s\}\ n)$	$O(\prod \lceil l_j/D \rceil\ )$
PCCS + double-array	$O(d\ n\ log\ n + d\ min\{3^d, s\}\ n)$	$O(n^d)$
PCCDA + double-array	$O(d\ min\{3^d, s\}\ n)$	$O(n^d + \max\lceil l_j/D \rceil)$

Theorem 1: Algorithm *NMS* generates a solution for the CS_d problem with approximation bound 2^d. The time complexity for different implementations of the algorithm is given in Table I.
Proof: By the above discussion.

□

(B) SIMPLE AGGREGATION.

This algorithm is the simplest of the four and it generates its solutions quickly. Its worst case approximation bound is identical to the one of the algorithm in the previous subsection and it has a smaller worst case time complexity bound. The only drawback is that the solutions it generates are "normally" inferior, with respect to the objective function criteria, to the one generated by the algorithms in the previous sections. This was our conclusion after 20000 experiments. However, in many instances its solutions are superior.

The algorithm, which we refer to as *SA*, is very simple. First it finds the *g−square* where each point belongs. Then it introduces a square for each *g−square* with at least one point in it. Let $F_{apx} = t$ be the total number of squares introduced. We claim that $F_{opt} \geq t/2^d$ because no square in an optimal solution may have points from more than 2^d of the *g−squares*. Therefore, the approximation bound is 2^d.

Let us now consider different implementations. Finding the *g−squares* where the points belong takes $O(d\ n)$ time. Once we have the *g−squares*, we eliminate multiple entries by sorting the *g−squares* with points. The sorting takes $O(dn + n \log t)$ time when using the procedures discussed in [G]. Therefore the overall time complexity of this implementation, which we call balanced tree implementation is $O(dn + n \log t)$ time. The other implementations for the second algorithm are similar to the ones for the first algorithm. For brevity we shall not discuss them in detail. The performance of our algorithms is summarized in Table II.

Table II. Time and Space Complexity for several implementations of algorithm SA.

implementation	time complexity	space complexity
balanced tree	$O(d\ n + n \log t)$	$O(n)$
double-array	$O(d\ n)$	$O(\prod \lceil l_j/D \rceil)$
PCCDA + double-array	$O(d\ n)$	$O(n^d + \max \lceil l_j/D \rceil)$

Theorem 2: Algorithm *SA* generates a solution for the CS_d problem with approximation bound 2^d. The time complexity for different implementations of the algorithm is given in Table II.
Proof: By the above discussion.

□

For any of the implementations discussed for this algorithm there are problem instances that achieve the worst case approximation bound. This algorithm outperforms all the algorithms when t is large and there are many points per g*−square*.

(C) ORDERED MAXIMAL SET OF INDEPENDENT POINTS APPROACH.

Let us consider the third algorithm. The idea behind this algorithm is similar to the first one, except that we find a maximal independent set in an orderly fashion. Let us consider first the case when $d = 2$. We sort the points with respect to their x-coordinate values (in case of ties the order is not important). We shall traverse the points in that order. Our third approximation algorithm (OMS) is given below.

```
Algorithm OMS( P , D )
 Rearrange the set of points with respect to their first coordinate value;
 // I.e., $x_1(p_1) \le x_1(p_2) \le \ldots \le x_1(p_n)$ //
 $S \leftarrow \emptyset$;
 for $i = 1$ to $n$ do
   if $p_i$ is independent from every point $p$ in $S$ then $S \leftarrow S \cup \{p_i\}$;
 endfor
 Cover all points dependent on each point in $S$ using the rule given in figure 2;
end of algorithm OMS
```

As in the case of the analysis of procedure MS, it is simple to show that $F_{opt} \ge |S|$. For the case when $d = 2$, the set of points dependent on a point p in S that are not located to its left can be covered by two squares as shown in figure 2. It is important to note that when considering point p all the points previously visited can be covered by adding two squares for each point in the current maximal independent set. Therefore, $F_{apx} \le 2\,|S|$. Combining these two bounds, we know that $F_{apx} / F_{opt} \le 2$. For $d > 2$, the approach is similar, but to cover all the points dependent on a point in set P one introduces no more than 2^{d-1} hypersquares. Figure 2 shows the cover for the case when $d = 3$.

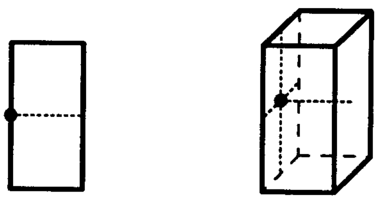

Figure 2: Covering the set of points dependent on p_i and located to the right of p_i.

The straight forward implementation of OMS, which is similar to the one for MS has overall time complexity $O(n \log n + d\ s\ n)$. The above implementation of the algorithm does not take advantage of special properties of the sets we are dealing with. Let us now consider these properties in more detail.

As with the case of algorithm MS, there are several implementations for algorithm OMS. The implementations are similar to the ones in subsection (A). The main difference is that instead of *t−squares* we have *t−rectangles* with dimensions D by $2D$ (rather than $2D$ by $2D$). Also, when considering point p we only need to consider those *t−rectangles* whose distance (along the first coordinate value) is at most D from it. Therefore, the cardinality of the set of *t−rectangles* may increase or decrease as the algorithm advances. Remember that for algorithm MS the cardinality of the set of *t−squares* only increased in size, there were no deletions. The "center" of a *t−rectangle* is the point on the left boundary that is equidistant to the left corner points of the rectangle. We define the *g−squares* as a strip of width D that is constantly moving from left to right, in such a way that the current point is located on its right boundary. Only those *t−rectangles* whose "center" is in this strip need to be considered. The *g−squares* are denoted by integer (i), for $i \geq 0$ (the bottommost *g−square* is labeled 0). Since all of the *t−rectangles* have dimensions D by $2D$ and no "center" of a *t−rectangle* is inside another *t−rectangle*, there is at most one "center" of a *t−rectangle* inside a *g−square*. I.e., the maximum number of independent points in each *g−square* is at most one. Determining whether a point p is inside a *t−rectangle* can be determined as follows: first we find the *g−square* (i) to which p belongs. Then, we construct the set S' of all independent points that belong to neighboring *g−squares*, i.e., belong to *g−square* (i') such that $|i'-i| \leq 1$. For $d = 2$, there are 3, and in general there are 3^{d-1} of such neighbors. Since the x-length and the y-length of each of the neighboring *g−squares* is D, it then follows that p is independent from each *t−rectangle* iff it is independent from each *t−rectangle* whose "center" is in set S'.

Let us now explain how to implement the above test efficiently. We define the set of elements S, such that *g−square* (i') has a "center" of a *t−rectangle* inside it iff $i' \in S$. Associated with each element $k \in S$ there a point, denoted by t_k that represents the "center" of the *t−rectangle* inside *g−square* (k). Given a point $p_i \in P$, by simple arithmetic operations we determine (i'), the *g−square* that contains p_i. Then we search in the set S for the elements that differ from i' by at most 1. Once we have determined which *g−squares* in the neighborhood have a "center" of a *t−rectangle* inside them, we test whether or not p_i is inside any of those *t−rectangles*. If the answer is yes, we just proceed to the next point, otherwise, p_i is independent from all points in S, so we have found a new *t−rectangle*. Remember that its "center" belongs to *g−square* (i'). Therefore, we add (i') to S. We also associate p_i with the entry i' and proceed to point p_{i+1}. At this point we delete the *t−rectangles* that do not belong to the new grid. Deletion in done in the FIFO fashion. The worst case time complexity for the new method is similar to the one in (A), though it is not hard to see that in a large number of problem instances the third algorithm would be faster.

The other implementations for the third algorithm are similar to the ones for the first algorithm. For brevity we shall not discuss them in detail. The performance of our algorithms is summarized in Table III.

Table III. Time and Space Complexity for several implementations of algorithm OMS.

implementation	time complexity	space complexity
straight forward	$O(n \; log \; n + d \; s \; n)$	$O(n)$
balanced tree	$O(n \; log \; n + \min\{3^{d-1}, s\} \; (d + log \; s) \; n)$	$O(n)$
double-array	$O(n \; log \; n + d \; min\{3^{d-1}, s\} \; n)$	$O(\prod \lceil l_j / D \rceil)$
PCCS + double-array	$O(d \; n \; log \; n + d \; min\{3^{d-1}, s\} \; n)$	$O(n^d)$
PCCDA + double-array	$O(n \; log \; n + d \; min\{3^{d-1}, s\} \; n)$	$O(n^d + \max \lceil \; l_j / D \rceil)$

Theorem 3: Algorithm *OMS* generates a solution for the CS_d problem with approximation bound 2^{d-1}. The time complexity for different implementations of the algorithm is given in Table III.
Proof: By the above discussion.

□

(D) PARTITION INTO INDEPENDENT SUBPROBLEMS APPROACH.

The fourth algorithm is slightly different than the previous ones. Its approximation bound is identical to the one for the third algorithm, but it can be implemented to run faster. Before we explain it, let us consider a restricted version of it which we call the *slab problem*, i.e., all points are located inside a rectangle (whose sides are orthogonal to the axes) with height D. An optimal solution to the slab problem can be obtained by projecting all points to the bottom side of the rectangle and applying an optimal algorithm for the one dimensional case. An optimal algorithm for the one-dimensional problem considers all points in increasing order. The leftmost point is covered by the leftmost end point of a line segment of length D. All the points that fall inside this line segment are covered by the line. Then the leftmost uncovered point is covered in a similar fashion. This procedure, referred to as the *cover the leftmost point first (CLPF)*, has worst case time complexity of $O(n \; log \; n)$ and generates a minimum cardinality cover.

The CLPF procedure is not the procedure with the best worst case time complexity bound to find an optimal solution to the slab problem. Let us now present a better algorithm (**fastCLPF**) whose worst case time complexity bound is only O(n $\log s$), where s is the number of elements in an optimal cover. The idea behind the procedure is to sort the elements with respect to their $i_1(p)$ values. The elements are placed in sets $S_1, S_2, ..., S_k$, where all elements in set S_i have identical $i_1(p)$ values and the $i_1(p)$ value of the elements in set S_i is smaller than those in S_{i+1}. Clearly, no element in S_i can be with an element in S_{i+2} in the same square in a feasible solution. The idea behind the algorithm is to apply the CLPF procedure in such a way that it uses the above properties of the sets $S_1, S_2, ..., S_k$.

```
Algorithm fastCLPF( $P$, $D$ )
 Sort the elements with respect to $i_1(p)$ into sets $S_1, S_2, ... S_k$;
 $j \leftarrow 2$;  $R \leftarrow S_1 \cup S_2$;
 while $R \neq \emptyset$ do
  $q \leftarrow min\{x_1(p) \mid p \in R\}$;
  All points at a distance at most $D$ (with respect to $x_1$) from $q$ in $R$ and $q$ are removed from $R$; and
  we define $Q$ to contain only those elements;
  output($Q$)
  while $j < k$ and $R$ contains elements from at most one of the sets in $\{S_1, S_2, ..., S_k\}$ do
   $j \leftarrow j + 1$;  $R \leftarrow R \cup S_j$;
  endwhile
 endwhile
end of algorithm fastCLPF;
```

We claim that the cover generated by the above procedure is identical to the one generated by procedure CLPF. The reason is that when R has elements from two adjacent sets, S_i and S_{i+1}, q must be in set S_i and no element in S_{i+2} or higher indexed set can be dependent on point q. We also claim that the time complexity is O(n $\log s$). Sorting takes O(n $\log s$) time since at least there are $k/2$ independent points in $S_1 \cup S_2 \cup ... \cup S_k$. The *min* operation is performed on each element of S_i at most twice and the test in the while statement can be computed in constant time by associating with each element in R the set S_i where it belongs. Therefore, the overall time complexity is O(n $\log s$).

Now let us use the above procedure to obtain a fast approximation algorithm for the CS_2 and then for the CS_d problem. When $d = 2$, the set of points is partitioned into two sets, R_1 and R_2. For each of these two problems we find an optimal cover. For simplicity, let us explain our algorithm in terms of the *g−squares* introduced for the first algorithm. All the points that belong to the *g−squares* (i,j) such that j is odd belong to problem R_1 and the ones such that j is even belong to subproblem R_2. Let us consider problem R_1. It is simple to see that all the points in R_1 belong to slabs (with height D) and which are D units apart. Therefore, an optimal solution to R_1 consists of finding an optimal solution to each of the slabs which we know can be done in O(n $\log s$). Let us now formally define procedure PARTION-

FIRST based on the above strategy.

```
Algorithm PARTITION_FIRST( $P=\{p_1, p_2, ..., p_n\}, D$ )
 partition the elements with respect to $(i_2(p), ..., i_d(p))$ into sets $P_1, P_2, ..., P_k$;
 apply the fastCLPF procedure to each set;
end of algorithm PARTITION_FIRST
```

The partition can be done by sorting the elements. Sorting can be done by the algorithm in [G] (or an equivalent algorithm) in $O(d\ n + n\ log\ s)$ time. The second part applies procedure fastCLPF which takes overall time $O(n\ log\ s)$. Therefore, the overall time complexity of our procedure is $O(dn+n\ log\ s)$. We call this implementation the ***balanced tree*** implementation. Let us now consider the ***double array*** implementation of procedure PARTITION_FIRST. By using the double-array structure one can find the partition of the elements in $O(d\ n)$ time. Procedure first-CLPF can also be implemented in $O(d\ n)$ time by using double arrays. The idea is to first use the double-arrays to partition the elements into classes that have the same $i_1(p)$ value. Then "adjacent" classes are grouped together. The problem defined over each group of classes can be solved independently by applying the fast-CLPF procedure. The outcome would be the same as in the previous implementation, the only difference is the order in which the output is generated by the algorithm. By using pseudo connected components one can also decrease the space complexity. Table IV lists the time and space complexity for several implementations of our procedure.

Table IV. Time and Space Complexity for several implementations of algorithm PARTITION_FIRST.

implementation	time complexity	space complexity
balanced tree	$O(d\ n + n\ log\ s)$	$O(n)$
double-array	$O(\ d\ n\ )$	$O(\lceil I_1/D \rceil + \prod_{j=2}^{d} \lceil I_j/D \rceil\ )$
PCCS + double-array	$O(d\ n\ log\ n)$	$O(n^d)$
PCCDA + double-array	$O(d\ n)$	$O(n^d + max\lceil\ I_j/D \rceil\ )$

Theorem 4: Algorithm PARTITION-FIRST generates a solution for the CS_d problem with approximation bound 2^{d-1}. The time complexity for different implementations of the algorithm is given in Table IV.

Proof: By the above discussion. □

III. POLYNOMIAL TIME APPROXIMATION SCHEME.

Let us now consider the l_2-*slab* problem, i.e., all points lie in E^2 inside a rectangle with height lD, for some integer $l \geq 1$. First we show that the l_2-slab problem can be solved in $O(4ln^{4l})$ time via dynamic programming. Then we generalize the method to solve the l_d-slab problem (i.e., all points lie in E^d inside a hyperrectangle in which all dimensions, except the first, have length lD). Finally, we present an improved polynomial time approximation scheme by showing how to combine the algorithm for the l_d-slab problem with the polynomial time approximation scheme in [HM].

The idea behind the dynamic programming procedure is to find all covers that satisfy certain properties and cover at least points $p_1, p_2, ..., p_i$. The fact that the cover satisfies some special properties (which we specify later on) is important as otherwise there is an infinite number of such covers. The properties are very important because if we only specify simple properties then the number of distinct covers could be exponential on n. As we shall see, a careful definition reduces the number of covers to a number bounded by a polynomial on n.

Let us assume that the set of points are sorted with respect to their first coordinate value, i.e., $x_1(p_1) \leq x_1(p_2) \leq ... \leq x_1(p_n)$. A square in c_i in a cover C is said to be an *a−square (anchored square)* if there is a point r in P located on the left side of c_i and a point s in P (not necessarily different than r) located on the top side of c_i that are not contained (are not located inside or on the boundary) in any other square in the cover C. For an *a−square* c_i in cover C, the point p_j with least index located on the left (top) boundary of c_i which is not contained in any other square in C is called the *left (top) anchor of* c_i *in* C. We shall also refer to the two points as simply *anchors*. A cover is said to be an *a−cover (anchor cover)* if all the squares in it are *a−squares*. It is simple to show that any cover can be transformed to an *a−cover* without increasing the number of squares in it. Let C_{opt} be the set of optimal *a−covers* for some problem instance. Assume that the number of squares in each of these covers is t. We define b_i as the x_1 coordinate value of the left anchor of square c_i in the optimal *a−cover* C. Assume without loss of generality that each optimal *a−cover* C has been rearranged so that $b_1 \leq b_2 \leq ... \leq b_t$.

Define C_t as $\{ C \mid C \in C_{opt}$ and b_t for C is maximum amongst all covers in $C_{opt} \}$ and

C_j as $\{ C \mid C \in C_{j+1}$ and b_j for C is maximum amongst all covers in $C_{j+1}\}$.

A cover C in C_1 is said to be a *maximum index cover*. We say that an *a−cover* is a *s−cover* if every vertical line does not partition into two nonempty parts more than $2l$-1 squares in the cover. To show that the time complexity of our algorithm is bounded by a polynomial on n, we prove the following lemma which states that every problem instance has an *s−cover* among the set of optimal covers.

Lemma 1: Every l_2-slab problem has an *s−cover* among the set of optimal covers.

Proof: As mentioned above, every problem instance has at least one *a−cover* among the set of optimal covers. We now show that at least one of these *a−covers* in the set of optimal covers is an *s−cover*. The proof is by contradiction. Suppose that all optimal *a−covers* are not *s−covers*. Let C' be a maximum index cover among all optimal *a–covers*. By assumption there is a vertical line that partitions into two nonempty parts at least $2l$ squares in C'. Transform C' by the rule given in figure 3 and then move the newly introduced squares down and to the right until all of them are anchored or some can be deleted because all points have been covered. Let C be the new cover. If C has fewer squares than C', then it contradicts that C' is an optimal cover. So assume that there are the same number of squares in C and C'. So C is an *a–cover* in the set of optimal covers. If C is not an *s–cover*, then we contradict the fact that C' is an *a–cover* of maximum index because the maximum index cover in $\{C, C'\}$ is C. So it must be that C is an *s–cover* in the set of optimal covers. This contradicts the assumption that there is no *s–cover* in the set of optimal covers and thus completes the proof of the lemma. □

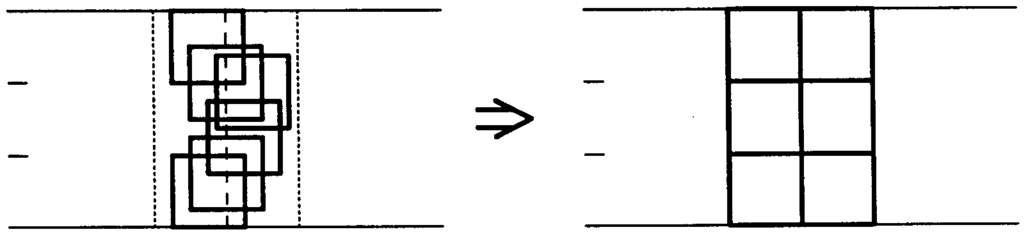
Figure 3: Transformation rule when l = 3.

Before presenting our algorithm we need to make a few more definitions. Deleting a subset of squares from an *a–cover* C results in a *partial a–cover* which we call C'. If points $p_1, p_2, \dots p_i$ are covered by a *partial a–cover* C', and all the left anchors of the squares in C' are points from the set $\{p_1, p_2, \dots, p_i\}$, then C' is called an *i-partial a–cover*. Two *i-partial a–covers* are said to be ***equivalent*** if the left and top anchors of every square partitioned into two nonempty parts by any vertical line "immediately to the right" (any line between p_i and p_j, where j is the smallest index of a point that appears to the right of p_i) of p_i are identical in the two *i-partial a–covers*. The *i-partial a–cover* C *dominates i-partial a–cover* C' iff C and C' are equivalent and C has fewer squares that C'. An *i-partial a–cover* is said to be an *optimal i-partial a–cover* if there is an *s–cover* in the set of optimal covers with an *i-partial a–cover* equivalent to it. Procedure **DP** processes the points in the order $p_1, p_2, \dots, p_n$. During the ith iteration we construct the set of all irreducible *i-partial a–covers*, i.e., set of all *i-partial a–covers* such that no one of them dominates another and no two of them with the same number of squares are equivalent. To show correctness we prove in lemma 2 that all optimal *i-partial a–covers* are in the set of irreducible *i-partial a–covers*.

```
Algorithm DP( $p_1, p_2, ..., p_n$ )
 sort the elements so that $x_1(p_1) \le x_1(p_2) \le ... \le x_1(p_n)$;
 $FS \leftarrow \{\emptyset\}$; /* a set with one element whose value is an empty list of squares */
 for $i = 1$ to $n$ do
  $newFS \leftarrow \emptyset$;
  for each partial $a$-cover $C$ in $FS$ do
   case
    :$p_i$ is covered in $C$: Add $\{C\}$ to $newFS$;
    :there are less than $2l-1$ squares partitioned by a vertical line "immediately to the right" of $p_i$:
              Add { $C'$ | $C'$ is $C$ plus an $a$-square with $p_i$ as its left anchor } to $newFS$;
    :else: /* $newFS$ remains unchanged */;
   endcase
  endfor;
  Let $FS$ be a maximal cardinality subset of irreducible $a$-covers in $newFS$;
 endfor;
 Output any cover with the least number of squares in $FS$
end of algorithm DP
```

Lemma 2: For every l_2-slab problem, algorithm DP outputs an optimal cover.
Proof: It is simple to show that the proof of the lemma reduces to showing that at the end of each iteration FS contains the set of optimal *i-partial a−covers*. This can be proved by induction.

□

To establish our time complexity bound we need to specify some implementation details. We find a maximal subset of irreducible *a−covers* in *newFS* as follows. Each *a−cover* in *newFS* is characterized by the anchors of the squares that extend to the right of p_i and the number of squares in the cover. We sort via radix sort the elements in *newFS* using as keys the indices of the points which are anchors (in sorted order) of the squares that extend to the right of p_i. Associated with each key there is the number of squares in the cover. Deletion of dominated *i-partial a−covers* and equivalent *i-partial a−covers* with the same number of elements can be easily done by traversing the list once. Thus, if *newFS* has m elements, the overall time taken to execute the step is $c(m+n)$ time, where c is the maximum number of anchor points in an *i-partial a−cover*. The above bound can be achieved through radix sort since the keys are c dimensional points whose value along each axis is an integer in the range $[1,n]$.

Lemma 3: For every l_2-slab problem, algorithm DP takes $O(4l\ n^{4l})$ time.
Proof: By lemma 1 and the fact that there are at most two anchors per square, $|FS| \le n^{4l-2}$. Since for each element in FS at most n elements are added to $newFS$, $|newFS| \le n^{4l-1}$. Using the procedure discussed above the maximal set of irreducible covers in $newFS$ can be identified in $O((4l-2)n^{4i-1})$ time. The above operation is repeated n times, therefore the overall time complexity is $O(4l\ n^{4l})$.

□

For the l_d-slab problem, the number of squares partitioned by a hyperplane passing through p_i is at most $2l^{d-1}$-1 instead of $2l$-1, and the number of anchors per square is d instead of two. Therefore, the time complexity for the l_d-slab problem is $O(d(2l^{d-1}-1)n^{d(2l^{d-1}-1)+1})$.

This algorithm can be easily incorporated with the polynomial time approximation scheme given in [HM]. The idea is to apply the algorithm to l^d problem instances of the l_d-slab problem. The l^d problem instances can be partitioned into l^{d-1} groups of problems such that in each group the number of points in all the problems is n. This results in an overall time complexity bound $O(l^{d-1}\ d(2l^{d-1}-1)n^{d(2l^{d-1}-1)+1})$. The approximation bound is $(1+1/l)^{d-1}$.

Theorem 5: Combining algorithm DP with the polynomial time approximation scheme in [HM] results in a procedure that generates a solution to the CS_d problem with approximation bound $(1+1/l)^{d-1}$ and time complexity $O(l^{d-1}\ d(2l^{d-1}-1)n^{d(2l^{d-1}-1)+1})$.
Proof: By the above discussion.

□

IV. DISCUSSION.

Let us now discuss some postprocessing procedures for the approximation algorithms in subsections (A) and (C) in section II. We only discuss the case when $d = 2$, since the other cases are similar. The procedure in subsection A (C) introduces four (two) squares for each independent point. An improved solution may be obtained by noting that one can cover all points without having to introduce $4s$ squares. Heuristics to generate least cardinality covers can be easily developed.

All our techniques can also be adapted to the CR_d and the CD_d problem. For the CR_d problem the approximation and time complexity bounds are identical. For the CD_d problem the algorithms in section II have identical time complexity bounds; however, the approximation bounds are $\lceil 2\sqrt{d}\rceil^d$, $\lceil 2\sqrt{d}\rceil^d$, $\lceil 2\sqrt{d}\rceil^{d-1}\lceil\sqrt{d}\rceil$, and $2^{d-1}\lceil\sqrt{d}\rceil^d$, respectively. For the polynomial time approximation scheme, the approximation bound is $2(1 + 1/l)^{d-1}$ and the time complexity bound is

$O(l^{d-1}d\lceil 2\sqrt{d}\rceil \lceil l\sqrt{d}\rceil^{d-1} n^{d\lceil 2\sqrt{d}\rceil^{d-1}+1})$. The same techniques can be adapted to the packing problem studied in [MH]. The approximation and time complexity bounds are smaller than for the CS_d problem. For brevity we do not discuss this further. As pointed out in [HM] it is important to develop fast approximation algorithms for the l_d-slab problem. These algorithms would imply a much better time complexity bound for the CS_d problem at the expense of a slightly larger approximation bound.

From our algorithms one can define a heuristic which we believe has a reasonable behavior with respect to the time complexity and the approximation. The idea is to assign the points to the *g−squares* first. Then the *g−squares* with points are partitioned into two groups. Those with a single point and those with at least one point. The heuristic then applies our polynomial time approximation scheme to the problem defined by all the points belonging to single point *g−squares*. An optimal solution will be generated quickly for this subproblem. The remaining problem is solved by any of the approximation algorithms in section II.

V. REFERENCES.

[AHU] Aho, A., J. Hopcroft and J. Ullman, *The Design and Analysis of Computer Algorithms,* Addison-Wesley, 1975.

[FPT] Flower, R. J., M. S. Paterson, and S. L. Tanimoto, "Optimal Packing and Covering in the Plane are NP-complete," Information Processing Letters 12, 1981, 133 - 137.

[G] Gonzalez, T., "The On-Line d-Dimensional Dictionary Problem", Technical Report, University of Utrecht, July, 1990.

[HM] Hochbaum D. S. and W. Maass, "Approximation Schemes for Covering and Packing Problems in Image Processing and VLSI," *Journal of the ACM,* Vol. 32, No. 1, January 1985, 130 - 136.

[J] Johnson, D., "The NP-Completeness Column: An Ongoing Guide," *Journal of Algorithms* 3, 182 - 195, 1982.

[MIH] Masuyama, S., T. Ibaraki, and T. Hasegawa, "The Computational Complexity of the m-center problems on the Plane," *Transactions IECE of Japan* E64, 1981, 57-64.

[S] Supowit, K. J., "Topics in Computational Geometry," Report No. UIUCDCS-R-81-1062, Department of Computer Science, University of Illinois, Urbana, Ill., 1981.

[T] Tanimoto, S. L, "Covering and Indexing an Image Subset," Proceedings of the 1979 IEEE Computer Society Conference on Pattern Recognition and Image Processing, 239 - 245, 1979.

[TF] Tanimoto, S. L. and R. J. Fowler, "Covering Image Subsets with Patches," Proceedings of the 5th International Conference on Pattern Recognition, 835 - 839, 1980.