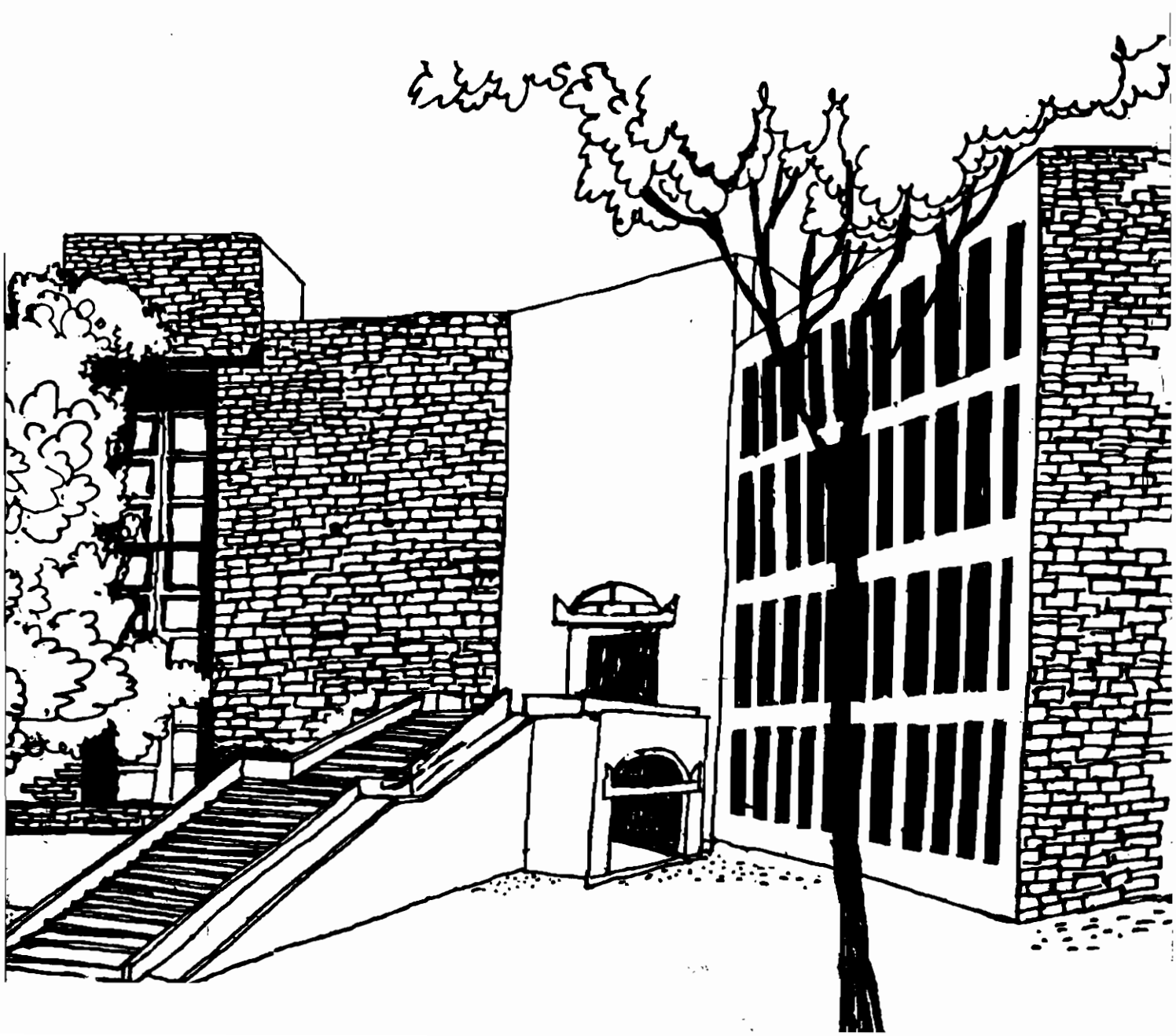
विद्याविनियोगाद्विकासः
IIM
AHMEDABAD

Working Paper



Data Correcting Algorithms for Combinatorial Optimization

Boris Goldengorin
Diptesh Ghosh
Gerard Sierksma

W.P. No. 2004-04-05
April 2004 /1808

INDIAN INSTITUTE OF MANAGEMENT
AHMEDABAD-380 015
INDIA

Data Correcting Algorithms in Combinatorial Optimization

Boris Goldengorin[1] *Diptesh Ghosh*[2] *Gerard Sierksma*[1]

1. Faculty of Economic Sciences
   University of Groningen, 9700AV Groningen, The Netherlands
   E-mail: {b.goldengorin, g.sierksma}@eco.rug.nl
2. P&QM Area, Indian Institute of Management
   Vastrapur, Ahmedabad 380015, Gujarat, India
   E-mail: diptesh@iimahd.ernet.in

Abstract

This paper describes data correcting algorithms. It provides the theory behind the algorithms and presents the implementation details and computational experience with these algorithms on the asymmetric traveling salesperson problem, the problem of maximizing submodular functions, and the simple plant location problem.

1 Introduction

Polynomially solvable special cases have long been studied in the literature on combinatorial optimization problems (see, for example, Gilmore *et al.* [19]). Apart from being mathematical curiosities, they often provide important insights for serious problem-solving. In fact, the concluding paragraph of Gilmore *et al.* [19] states the following, regarding polynomially solvable special cases for the traveling salesperson problem.

> " ··· **We believe, however, that in the long run the greatest importance of these special cases will be for approximation algorithms. Much remains to be done in this area.**"

This chapter describes a step in the direction of incorporating polynomially solvable special cases into approximation algorithms. We review *data correcting algorithms* — approximation algorithms that make use of polynomially solvable special cases to arrive at high-quality solutions. The basic

insight that leads to these algorithms is the fact that it is often easy to compute a bound on the difference between the costs of optimal solutions to two instances of a problem, even though it may be hard to compute optimal solutions for the two instances. These algorithms were first reported in the Russian literature (see Goldengorin [9, 10, 11, 12, 13]).

The approximation in data correcting algorithms is in terms of an *accuracy parameter*, which is an upper bound on the difference between the objective value of an optimal solution to the instance and that of a solution returned by the data correcting algorithm. Note that this is *not* expressed as a fraction of the optimal objective value for this instance as in common ϵ-optimal algorithms but as actual deviations from the cost of optimal solutions.

Although we suggest the use of data correcting algorithms to solve NP-hard combinatorial optimization problems, they form a general problem solving tool and can be used for functions defined on a continuous domain as well. We will in fact, motivate the algorithm in the next section using a function defined on a continuous domain, and having a finite range. We then show in Section 3, how this approach can be adapted for combinatorial optimization problems. In the next three sections we describe actual implementation of data correcting algorithms to three problems, the asymmetric traveling salesperson problem, the maximization of a general submodular function, and the simple plant location problem.

2 Data Correcting for Real-Valued Functions

Consider a real-valued function $f : D \rightarrow \Re$, where D is the domain on which the function is defined. We assume that f is not analytically tractable over D, but is computable in polynomial time for any $x \in D$, and concern ourselves with the problem of finding α-minimal solutions to the function f over D, i.e. the problem of finding a member of $\{x | x \in D, f(x) \leq f(x^*) + \alpha\}$, where $x^* \in \arg\min_D \{f(x)\}$, and α is the pre-defined *accuracy* parameter. The discussion here is for a minimization problem; the maximization problem can be dealt with in a similar manner.

Let us assume a partition $\{D_1, \ldots, D_p\}$ of the domain D. Let us further assume that for each of the sub-domains D_i of D, $i = 1, \ldots, p$, we are able to find functions $g_i : D_i \rightarrow \Re$, which are easy to minimize over D_i, and such that

$$|f(x) - g_i(x)| \leq \frac{\alpha}{2} \quad \forall x \in D_i. \tag{1}$$

We call such easily minimizable functions *regular*.

Theorem 2.1 demonstrates an important relationship between the regular functions g_i and the original function f. It states that the function value of f at the best among the minima of all the g_i's over their respective domains is close to the minimum function value of f over the domain.

Theorem 2.1: Let $x_i^\alpha \in \arg\min_{x \in D_i}\{g_i(x)\}$, $x^\alpha \in \arg\min_i\{f(x_i^\alpha)\}$, and let $x^* \in \arg\min_{x \in D}\{f(x)\}$. Then

$$f(x^\alpha) \leq f(x^*) + \alpha.$$

Proof: Let $x_i^* \in \arg\min_{x \in D_i}\{f(x)\}$. Then for $i = 1, \ldots, p$, $f(x_i^\alpha) - \frac{\alpha}{2} \leq g_i(x_i^\alpha) \leq g_i(x_i^*) \leq f(x_i^*) + \frac{\alpha}{2}$, i.e. $f(x_i^\alpha) \leq f(x_i^*) + \alpha$. Thus $\min_i\{f(x_i^\alpha)\} \leq \min_i\{f(x_i^*)\} + \alpha$, which proves the result. ■

Notice that x^* and x^α do not need to be in the same sub-domain of D.

Theorem 2.1 forms the basis of the data correcting algorithm to find an approximate minimum of a function f over a certain domain D. The procedure consists of three steps: the first in which the domain D of the function is partitioned into several sub-domains; the second in which f is approximated in each of the sub-domains by regular functions following the condition in expression (1) and a minimum point of the regular function is obtained; and a third step, in which the minimum points computed in the second step are considered and the best among them is chosen as the output. This procedure can be further strengthened by using lower bounds to check if a given sub-domain can possibly lead to a solution better than any found thus far. The approximation of f by regular functions g_i is called *data correcting*, since an easy way of obtaining the regular functions is by altering the data that describe f. A pseudocode of the algorithm, which we call DC-G, is provided below.

Procedure DC-G(f, D, α)
Output: $x^\alpha \in D$ such that $f(x^\alpha) \leq \min\{f(x) | x \in D\} + \alpha$.

Code:

```
1. begin
2.     create a partition {D_1, ..., D_n} of D;
3.     for each sub-domain D_i
4.     begin
5.         f_i := a lower bound to f(x), x ∈ D_i;
6.         if f_i ≥ bestvalue
7.             continue;
```

```
8.          construct a regular function g_i(x) obeying (1);
9.          x_i^α ∈ arg min_{x∈D_i}{g_i(x)};
10.     end;
11.     bestvalue := ∞;
12.     if f(x_i^α) < bestvalue;
13.     begin
14.         x^α := x_i^α;
15.         bestvalue := f(x^α);
16.     end;
17.     return x^α;
18. end.
```

Lines 5 through 7 in the code carry out the bounding process, and lines 8 and 9 implement the process of computing the minima of regular functions over each sub-domain. These steps are enclosed in a loop, so that at the end of line 10, all the minima of the regular functions are at hand. The code in lines 11 through 16 obtain the best among the minima obtained before. By Theorem 2.1, the solution chosen by the code in lines 11 through 16 is an α-minimum of f, and therefore, this solution is returned by the algorithm in line 17.

We will now illustrate the data correcting algorithm through an example. The example that we choose is one of a real-valued function of one variable, since these are some of the simplest functions to visualize.

Consider the problem of finding an α-minimum of the function f shown in Figure 1. The function is assumed to be well-defined, though analytically intractable on the domain D.

The data correcting approach can be used to solve the problem above, i.e. of finding a solution $x^\alpha \in D$ such that $f(x^\alpha) \leq \min\{f(x) | x \in D\} + \alpha$.

Consider the partition $\{D1, D2, D3, D4, D5\}$ of D shown in Figure 2. Let us suppose that we have a regular function $g1(x)$ with $|g1(x) - f(x)| \leq \frac{\alpha}{2}$, $\forall x \in D1$. Assume also, that $x1$ is a minimum point of $g1(x)$ in $D1$. Since this is the best solution that we have so far, we store $x1$ as an α-minimal solution to $f(x)$ in the domain $D1$. We then consider the next interval in the partition, $D2$. We first obtain a lower bound on the minimum value of $f(x)$ on $D2$. If this bound is larger than $f(x1)$, we ignore this domain and examine domain $D3$. Let this not be the case in our example. So we construct a regular function $g2(x)$ with $|g2(x) - f(x)| \leq \frac{\alpha}{2}$, $\forall x \in D2$, and find $x2$, its minimum point over $D2$. Since $f(x2) \geq f(x1)$ (see Figure 2),

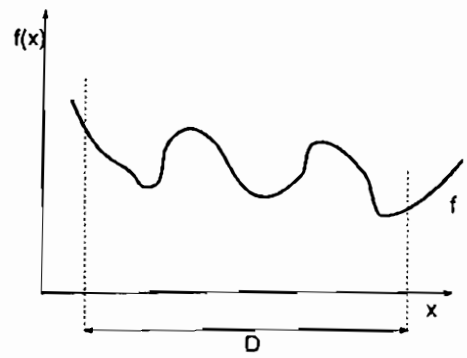


Fig. 1: A general function f

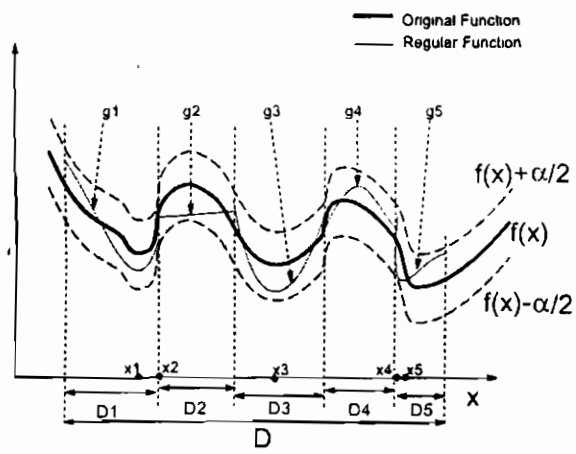


Fig. 2: Illustrating the Data Correcting approach on f

we retain $x1$ as our α-optimal solution over $D1 \cup D2$. Proceeding in this manner, we examine $f(x)$ in $D3$ through $D5$, compute regular functions $g3(x)$ through $g5(x)$ for these domains, and compute $x3$ through $x5$. In this example, $x3$ replaces $x1$ as our α-minimal solution after consideration of $D3$, and remains so until the end. At the end of the algorithm, $x3$ is returned as a value of x^α.

There are four points worth noting at this stage. The first is that we

need to examine all the sub-domains in the original domain before we return a near-optimal solution using this approach. The reason for this is very clear. The correctness of the algorithm depends on the result in Theorem 2.1, and this theorem only concerns the *best* among the minima of each of the sub-domains. For instance, in the previous example, if we stop as soon as we obtain the first α-optimal solution x_1 we would be mistaken, since Theorem 2.1 applies to $x1$ only over $D1 \cup D2$. The second point is that there is no guarantee that the near-optimal solution returned by DC-G will be in the neighborhood of a true optimal solution. There is in fact, nothing preventing the near-optimal solution existing in a sub-domain different from the sub-domain of an optimal solution, as is evident from the previous example. The true minimum of f lies in the domain $D5$, but DC-G returns $x3$, which is in $D3$. The third point is that the regular functions $g_i(x)$ approximating $f(x)$ do not need to have the same functional form. For instance in Example 1, $g1(x)$ is quadratic, while $g2(x)$ is linear. Finally, for the proof of Theorem 2.1, it is sufficient for $\{D_1, \ldots, D_n\}$ to be a cover of D (as opposed to a partition as required in the pseudocode of DC-G).

3 Data Correcting for Combinatorial Optimization Problems

The data correcting methodology described in the previous section can be incorporated into an implicit enumeration scheme (like branch and bound) and used to obtain near-optimal solutions to NP-hard combinatorial optimization problems. In this section we describe how this incorporation is achieved for a general combinatorial optimization problem.

We define a combinatorial optimization problem P as a collection of instances I. An instance I consists of a ground set $G = \{e_1, e_2, \ldots, e_n\}$ of n elements, a cost vector $C_I = (c_1^I, c_2^I, \ldots, c_n^I)$ corresponding to the elements in G, a set $S \subseteq 2^G$ of feasible solutions, and a cost function $f_I : S \rightarrow \Re$. The objective is to obtain a solution, i.e. a member of S that minimizes the cost function. For example, for an asymmetric traveling salesperson problem (ATSP) instance on a digraph $G = (V, A)$, with a distance matrix $D = [d_{ij}]$, we have $G = A$, $c_{ij}^I = d_{ij}$, S is the set of all Hamiltonian cycles in G, and $f_I(s) = \sum_{(i,j) \in s} d_{ij}$ for each $s \in S$.

Implicit enumeration algorithms for combinatorial problems include two main strategies, namely branching and fathoming. Branching involves partitioning the set of feasible solutions S into smaller subsets. This is done under the assumption that optimizing the cost function over a more restricted solution space is easier than optimizing it over the whole space. Fathoming involves one of two processes. First, we could compute lower bounds to the

value that the cost function can attain over a particular member of the partition. If this bound is not better than the best solution found thus far, the corresponding subset in the partition is ignored in the search for an optimal solution. The second method of fathoming is by computing the optimum value of the cost function over the particular subset of the solution space (if that can be easily computed for the particular subset). We see therefore that two of the main requirements of the data correcting algorithm presented in the previous section, i.e. partitioning and bounding, are automatically taken care of for combinatorial optimization problems by implicit enumeration. The only other requirement that we need to consider is that of obtaining regular functions approximating f_I over subsets of the solution space.

Notice that the cost function $f_I(s)$ is a function of the cost vector C. So if the values of the entries in C are changed, $f_I(s)$ undergoes a change as well. Therefore, cost functions corresponding to polynomially solvable special cases can be used as "regular functions" for combinatorial optimization problems. Also note that for the same reason, the accuracy parameter can be compared with a suitably defined distance measure between two cost vectors, (or equivalently, two instances). Consider a subproblem in the tree obtained by normal implicit enumeration. The problem instance that is being evaluated at that subproblem is a restricted version of the original problem instance, i.e., it evaluates the cost function of the original problem instance for a subset S_k of the original solution space S. If we alter the data of the problem instance in a way that the altered data corresponds to a polynomially solvable special case, while guaranteeing that the cost of an optimal solution to the altered problem in S_k is not more that an acceptable amount higher than the cost of an optimal solution to original instance in S_k, then the altered cost function can be considered to be a regular approximation of the cost function of the original instance in S_k.

For combinatorial optimization problems, let us define a *proximity measure* $\rho(I_1, I_2)$ between two problem instances I_1 and I_2, as an upper bound for the difference between $f_{I_1}(s_1^\star)$ and $f_{I_1}(s_2^\star)$, where $s_1^\star$ and $s_2^\star$ are optimal solutions to I_1 and I_2 respectively. The following lemma shows that the *Hamming distance* between the cost vectors of the two instances is a proximity measure when the cost function is of the sum type or the max type.

Lemma 3.1: If the cost function of an instance I of a combinatorial optimization problem is of the *sum* type, (i.e. $f_I(s) = \sum_{e_k \in s} c_k^I$) or the *max*

type, (i.e. $f_I(s) = \max_{e_k \in s} c_k^I$) then the measure

$$\rho(I_1, I_2) = \sum_{e_i \in G} |c_i^{I_1} - c_i^{I_2}| \tag{2}$$

between two instances I_1 and I_2 of the problem is an upper bound to the difference between $f_{I_1}(s_1^*)$ and $f_{I_1}(s_2^*)$, where s_1^* and s_2^* are optimal solutions to I_1 and I_2, respectively.

Proof: We will prove the result for sum type cost functions. The proof for max type cost functions is similar.

For sum type cost functions, it is sufficient to prove the result when the cost vectors C_{I_1} and C_{I_2} differ in only one position. Let $c_k^{I_1} = c_k^{I_2}$ for $k = 1, 2, \ldots, j-1, j+1, \ldots, n$, and $c_j^{I_1} \neq c_j^{I_2}$. Consider any solution $s \in S$. There are two cases to consider:

- $e_j \in s$: In this case, $|f_{I_1}(s) - f_{I_2}(s)| = \sum_{e_k \in S} |c_k^{I_1} - c_k^{I_2}| = |c_j^{I_1} - c_j^{I_2}|$.
- $e_j \notin s$: In this case it is clear that $f_{I_1}(s) = f_{I_2}(s)$.

Therefore, $|f_{I_1}(s) - f_{I_2}(s)| \leq \sum_{e_i \in G} |c_i^{I_1} - c_i^{I_2}| = \rho(I_1, I_2)$ for any solution $s \in S$, which automatically implies that $\rho(I_1, I_2)$ as defined in the statement of Lemma 3.1 is an upper bound for the difference between $f_{I_1}(s_1^*)$ and $f_{I_1}(s_2^*)$, where s_1^* and s_2^* are optimal solutions to I_1 and I_2, respectively. ■

At this point, it is important to point out the difference between a fathoming step and a data correcting step. The bounds used in fathoming steps consistently overestimate the objective function of optimal solutions in maximization problems and underestimate it for minimization problems. The amount of over- or underestimation is not bounded. In data correcting steps however, the "regular function" may overestimate or underestimate the objective function, regardless of the objective of the problem. However, there is always a bound on the deviation of the "regular function" from the objective function of the problem.

One way of implementing the data correcting for a NP-hard problem instance I is the following. We first execute a data correcting step. We construct a polynomially solvable relaxation I_L of the original instance, and obtain an optimal solution x_L to I_L. Note that x_L need not be feasible to I. We next construct the best solution x to I that we can, starting from x_L. (If such a solution is not possible, we conclude that the instance does not admit a feasible solution.) We also construct an instance I_C of the problem, which will have x as an optimal solution. The proximity measure

$\rho(I, I_C)$ then is an upper bound to the difference between the costs of x and of an optimal solution to I. I_C is called a *correction* of the instance I. If the proximity measure is not more that the allowable accuracy, then we can output x as an adequate solution to I. If this is not the case, then we partition the feasible solution space of I (these are formed by adding constraints to the formulation of I) and apply the data correction step to each of these subproblems.

The similarity in the procedural aspects of the data correcting step described above (and illustrated in the example) to fathoming rules used in branch and bound implementations makes it convenient to incorporate data correcting in the framework of implicit enumeration. We present the pseudocode of a recursive version of branch and bound incorporating data correcting below. The initial input to this procedure is the data for the original instance I, the feasible solution set S, any solution $s \in S$, and the accuracy parameter α. Notice that the data correcting step discussed earlier in this section is implemented in lines 6 through 10 in the pseudocode.

Algorithm DC(I, S, α)
Output: $x^\alpha \in S$ such that $f_I(x^\alpha) \leq \min\{f_I(x) | x \in S\} + \alpha$.

Code:

```
1. begin
2.      s := a solution in S;
3.      lb := a lower bound on the value of f_I(x) over S;
4.      if f_I(s) = lb return s;
5.      compute an optimal solution s_L to a polynomially solvable
6.      relaxation I_L to I;
7.      if possible then construct a solution s to I starting from s_L;
8.      else return "infeasible";          (* No solution s can be constructed *)
9.      construct an instance I_C that has s as an optimal solution;
10.     if ρ(I_C, I) ≤ α
11.           return s;
12.     else begin
13.           partition S into subsets S_1 through S_n;
14.           for i := 1 to n                                    (* Branch *)
15.                 s_i := DC(I, S_i, α);
16.           return the best solution from among s_1 through s_n;
17.     end;
18. end.
```

The algorithm described above is a prototype. We have not specified how the lower bound is to be computed, or which solution to choose in the feasible region, or how to partition the domain into sub-domains. These are details that vary from problem to problem, and are an important part in the engineering aspects of the algorithm. Note that this is just one of many possible ways of implementing a data correcting algorithm.

We can now describe our implementation of data correcting on specific combinatorial optimization problems. The next section deals with asymmetric traveling salesperson problems. The implementation of the data correcting algorithm for this problem closely follows the pseudocode above. Sections 5 and 6 deal with the maximization of general submodular functions and the simple plant location problem. Our implementations of data correcting for these two problems are slightly different, in which the data correction is done in an implicit manner.

4 The Asymmetric Traveling Salesperson Problem

In an asymmetric traveling salesperson problem (ATSP) instance we are given a weighted digraph $G = (V, A)$ and a $|V| \times |V|$ distance matrix $D = [d_{ij}]$ and our objective is output a least cost Hamiltonian cycle in this graph. This is one of the most studied problems in combinatorial optimization, see Lawler *et al.* [26] and Gutin and Punnen [20] for a detailed introduction.

4.1 The Data Correcting Algorithm

The data correcting algorithm (DC) presented in the previous section can be easily mapped to the ATSP. Lemma 3.1 takes the following form for ATSP instances.

Lemma 4.1: Consider two ATSP instances I_1 and I_2 defined on digraphs $G_1 = (V_1, A_1)$ and $G_2 = (V_2, A_2)$ respectively, with $|V_1| = |V_2|$. Let $D_1 = [d^1_{ij}]$ and $D_2 = [d^2_{ij}]$ be the distance matrices associated with I_1 and I_2. Further let T_1 and T_2 be optimal solutions to I_1 and I_2, and let L_1 and L_2 respectively represent the lengths of T_1 and T_2 in instance I_1. Then

$$L_2 - L_1 \leq \sum_{(i,j)\in A} |d^1_{ij} - d^2_{ij}|.$$

Before presenting a pseudocode for the algorithm, let us illustrate the data correcting step for the ATSP with an example. Consider the 6-city

ATSP instance with the distance matrix $D = [d_{ij}]$ shown below. (This **corresponds** to I in the discussion in the previous section.)

D	1	2	3	4	5	6
1	-	10	16	19	25	22
2	19	-	10	13	13	10
3	10	28	-	22	16	13
4	19	25	13	-	10	19
5	16	22	19	13	-	11
6	13	22	15	13	10	-

If we allow subtours in solutions to the ATSP, we get the assignment problem relaxation. Solving the assignment problem on D, using the Hungarian method, we get the following reduced distance matrix $D^H = [d^H_{ij}]$. (The assignment problem with the distance matrix D corresponds to I_L.)

D^H	1	2	3	4	5	6
1	-	0	6	7	16	12
2	9	-	0	1	4	0
3	0	18	-	10	7	3
4	8	14	2	-	0	8
5	5	11	8	0	-	0
6	2	11	4	0	0	-

This leads to a solution with two cycles (1231) and (4564) (corresponding to x_L). Using patching techniques (see for example Lawler *et al.* [26]), we obtain a solution (1245631) (corresponding to x). Notice that (1245631) would be an optimal solution to the assignment problem if d^H_{24} and d^H_{63} had been set to zero in D^H, and that would have been the situation, if d_{24} and d_{63} were initially reduced by 4 and 1 respectively, i.e. if the distance matrix in the original ATSP instance was D^P defined below. (This corresponds to I_C.)

D^P	1	2	3	4	5	6
1	-	10	16	19	25	22
2	19	-	10	9	13	10
3	10	28	-	22	16	13
4	19	25	13	-	10	19
5	16	22	19	13	-	11
6	13	22	14	13	10	-

Therefore D^P is the distance matrix of the correction of the instance with distance matrix D. The proximity measure $\rho(D, D^P) = \sum_{i=1}^{6} \sum_{j=1}^{6} |d_{ij} - d^P_{ij}| = |d_{24} - d^P_{24}| + |d_{63} - d^P_{63}| = 4 + 1 = 5$.

A proximity measure is an upper bound to the difference between the costs of two solutions for a problem instance, so the stronger the bound, the better would be the performance of any enumeration algorithm dependent on such bounds. It is possible to obtain stronger performance measures for ATSP instances, for example

$$\rho_1(D, D^P) = \min\left\{\sum_{i=1}^{n} \max_{1\le j\le n} |d_{ij} - d^P_{ij}|, \sum_{j=1}^{n} \max_{1\le i\le n} |d_{ij} - d^P_{ij}|\right\} \qquad (3)$$

is a better proximity measure than the one defined in (2).

Given the data correcting step, the DC algorithm presented in the previous section can be modified to solve ATSP instances. The pseudocode for a recursive version of this algorithm is given below.

Algorithm DCA-ATSP(G, α)
Output: A tour x^α such that the difference between the cost of x^α and the cost of an optimal tour is not more than α

Code:

```
1. begin
2.       s := an arbitrary tour in G;
3.       lb := a lower bound on the cost of an optimal tour;
4.       if f_I(s) = lb return s;
5.       s_L := an optimal solution to the assignment problem on G;
6.       construct a solution s from s_L through patching;
7.       using s_L, compute the proximity measure ρ;
8.       if ρ ≤ α
9.             return s;
10.      else begin                                   (* Branch *)
11.            branch according to a pre-decided branching rule;
12.            for each subproblem i generated
13.                  s_i := the solution output by DCA-ATSP on subproblem i;
14.            return the best solution from among all s_i's;
15.      end;
16. end.
```

Note that a good lower bound can be incorporated into DCA-ATSP to make it more efficient.

We next illustrate the DCA-ATSP algorithm above on an instance of the ATSP. Consider the 8-city ATSP instance with the distance matrix $D = [d_{ij}]$ shown below. (This example was taken from Balas and Toth [1], p. 381).

D	1	2	3	4	5	6	7	8
1	-	2	11	10	8	7	6	5
2	6	-	1	8	8	4	6	7
3	5	12	-	11	8	12	3	11
4	11	9	10	-	1	9	8	10
5	11	11	9	4	-	2	10	9
6	12	8	5	2	11	-	11	9
7	10	11	12	10	9	12	-	3
8	7	10	10	10	6	3	1	-

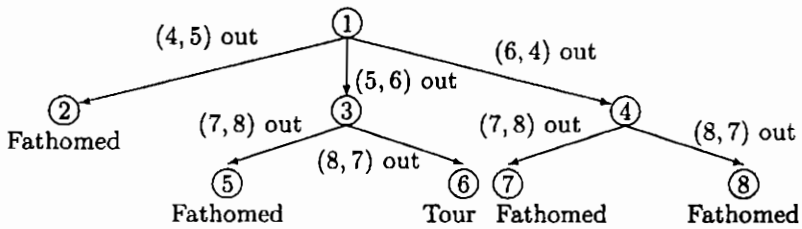
We use

- the proximity measure ρ (see Expression (2)) for data correction,
- the assignment algorithm to compute lower bounds for subproblems,
- a patching algorithm to create feasible solutions, and compute proximity measures, and
- the patched solution derived from the assignment solution as a feasible solution in the domain.

The branching rule used in this example is as follows. At each subproblem, we construct the assignment problem solution and then patch it. We also correct the original matrix to a new matrix that would output the patched solution if the assignment problem is solved on it. We next identify the arc corresponding to the entry in the new matrix that had to be corrected by the maximum amount. The tail of this arc is identified, and we branch on all the arcs in the subtour containing that vertex. For example, in this problem, the assignment solution is (1231)(4564)(787), which is patched to (123786451) and the cost of patching is 9. If we correct the problem data, we will see that the entry d_{51} (corresponding to arc $(5,1)$) contributes the maximum amount (7) to the patching. Hence we branch on each arc in the cycle (4564), and construct three subproblems, the first with the additional constraint that arc $(4,5)$ be excluded from the solution, the second with the additional constraint that arc $(5,6)$ be excluded from the solution, and the third with the additional constraint that arc $(4,5)$ be excluded from the solution.

The polynomially solvable special case that we consider is the set of all ATSP instances for which the assignment procedure gives rise to a cyclic permutation.

Using the branching rule described above, depth-first branch and bound generates the enumeration tree of Figure 3. The nodes are labeled according to the order in which the problems at the corresponding nodes were evaluated.



Subproblem at node	Upper bound	Lower bound	Assignment solution	Patched tour	Cost of patching	Revised bound
1	∞	17	(1231)(4564)(787)	(123786451)	9	26
2	26	29	Fathomed by bounds		–	26
3	26	25	(12631)(454)(787)	(126453781)	6	26
4	26	25	(12631)(454)(787)	(126453781)	6	26
5	26	31	Fathomed by bounds		–	26
6	26	26	(123786451)	Patching not required		26
7	26	31	Fathomed by bounds		–	26
8	26	27	Fathomed by bounds		–	26

Fig. 3: Branch and bound tree for the instance in the example.

Since the cost of patching equals the value of ρ, we can now evaluate the performance of data correcting on this example. If the allowable accuracy parameter α is set to 0, then the enumeration tree constructed by DC will be the one shown in Figure 3 and evaluates 8 subproblems. However if the value of α is set to 1, then enumeration stops after node 4, since the lower bound obtained is 25 which is one less than the solution we have at hand.

The previous example shows that the data correcting algorithm can be a very attractive alternative to branch and bound algorithms. In the next subsection we report experiences of the performance of the data correcting algorithm on ATSP instances from the TSPLIB [30].

4.2 Computational Experience with ATSP Instances

TSPLIB has twenty seven ATSP instances, out of which we have chosen **twelve** for our experiments. These twelve can be solved to optimality within **five** hours using an ordinary branch and bound algorithm. Eight of these belong to the 'ftv' class of instances, while four belong to the 'rbg' class. We implemented DCA-ATSP in C and ran it on a Intel Pentium based computer running at 666MHz with 128MB RAM.

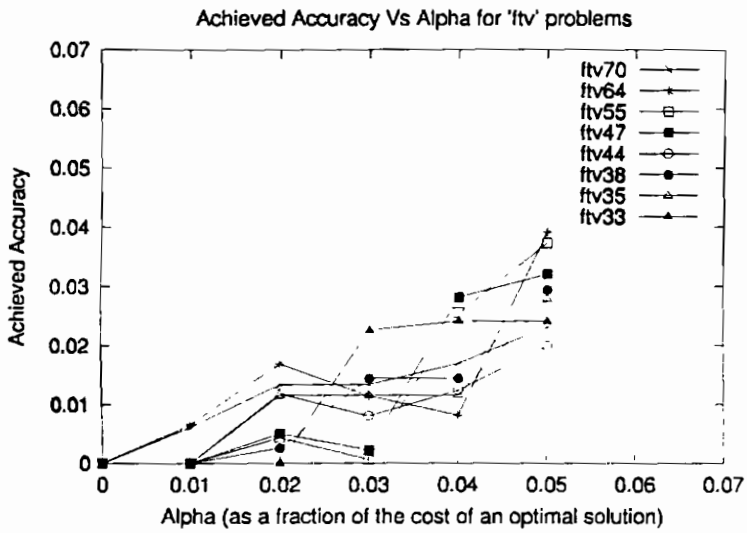


Fig. 4: Accuracy achieved versus α for ftv instances

The results of our experiments are presented graphically in Figures 4 through 7. In computing accuracies, (Figures 4 and 6) we have plotted the accuracy and deviation of the solution output by the data correcting algorithm from the optimal (called 'achieved accuracy' in the figures) as a fraction of the cost of an optimal solution to the instance. We observed that for each of the twelve instances that we studied, the achieved accuracy is consistently less than 80% of the pre-specified accuracy.

There was a wide variation in the CPU time required to solve the different instances. For instance, $ftv70$ required 17206 seconds to solve to optimality, while $rbg323$ required just 5 seconds. Thus, in order to maintain uniformity while demonstrating the variation in execution times with respect to changes in α values, we represented the execution times for each instance for each α value as a percentage of the execution time required to solve that instance to optimality. Notice that for all the ftv instances when α was 5% of the cost

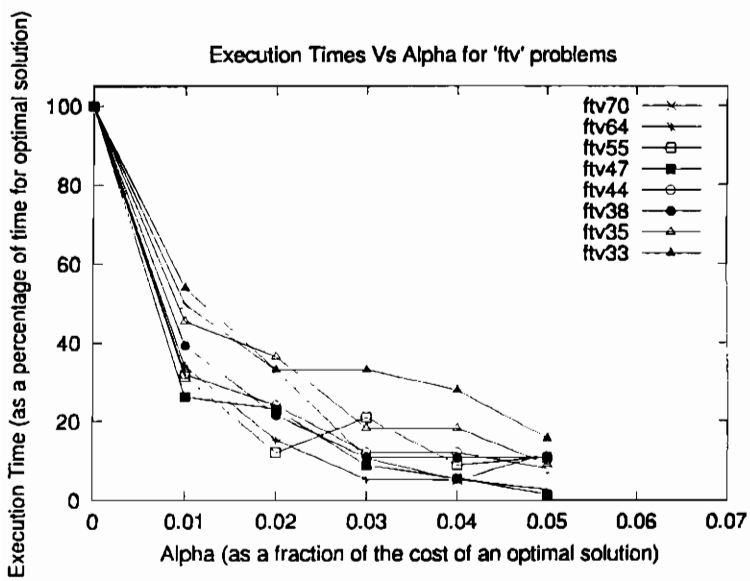


Fig. 5: Variation of execution times versus α for ftv instances

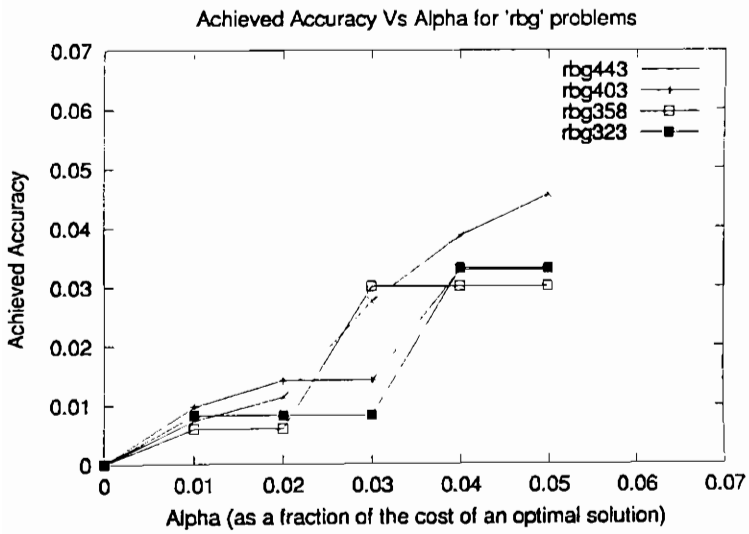


Fig. 6: Accuracy achieved versus α for rbg instances

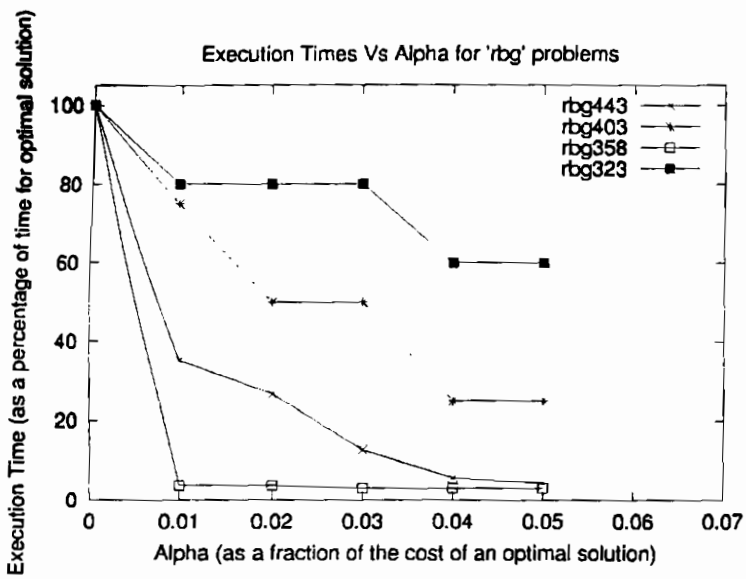


Fig. 7: Variation of execution times versus α for *rbg* instances

of the optimal solution, the execution time reduced to 20% of that required to solve the respective instance to optimality. The reduction in execution times for *rbg* instance was equally steep, with the exception of *rbg*323 which was in any case an easy instance to solve.

In summary, it is quite clear that data correcting is an effective methodology for solving ATSP instances. There are usually steep reductions in execution times even when the allowed accuracy is very small. This makes the method very useful for solving real world problems where a near-optimal solution is often acceptable provided the execution times are not too long.

5 Maximization of General Submodular Functions

Let $N = \{1, 2, \ldots, n\}$ and 2^N denote the set of all subsets of N. A function $z : 2^N \rightarrow \Re$ is called *submodular* if for each $I, J \in 2^N$, $z(I)+z(J) \geq z(I \cup J)+ z(I \cap J)$. The solution process of many classical combinatorial optimization problems, like the generalized transportation problem, the quadratic cost partition (QCP) problem with nonnegative edge weights, and set covering, can be formulated as the maximization of a submodular function (MSF), i.e. the problem:

$$\max\{z(I)|T \subseteq N\}.$$

Although the general problem of the maximization of a submodular function is known to be NP-hard (see Lovasz [28]), there has been a sustained research effort aimed at developing practical procedures for solving medium and large-scale problems in this class. In the remainder of this section we suggest two data correcting algorithms for solving the problem. Note that Lemma 3.1 assumes the following form for this problem.

Lemma 5.1: Consider two submodular functions $z_1(\overline{x}) = \sum_{i=1}^{n}\sum_{j=i}^{n} c_{ij}^1 \prod_{k=i}^{j} x_k$ and $z_2(\overline{x}) = \sum_{i=1}^{n}\sum_{j=i}^{n} c_{ij}^2 \prod_{k=i}^{j} x_k$. Let $\overline{x_1^*}$ and $\overline{x_2^*}$ be the maximum points of $z_1(\overline{x})$ and $z_2(\overline{x})$ respectively. Then

$$z_1(\overline{x_1^*}) - z_1(\overline{x_2^*}) \leq \sum_{i=1}^{n}\sum_{j=1}^{n} |c_{ij}^1 - c_{ij}^2|.$$

The algorithm described in Section 5.1 have been published in Goldengorin *et al.* [14] while that described in Section 5.2 have been published in Goldengorin and Ghosh [18]. For each of the two algorithms in we first describe a class of polynomially solvable instances for submodular function maximization problems. We then describe the data correcting algorithms that uses this class of polynomially solvable instances to solve a general submodular function maximization problem. The classes of polynomially solvable instances are algorithmically defined, i.e. they are classes of instances that are solved to optimality using a pre-specified polynomial algorithm.

5.1 A Simple Data Correcting Algorithm

The class of polynomially solvable instances that we describe here is defined using a polynomial time algorithm called the Preliminary Preservation (PP) algorithm. Normally these algorithms terminate with a subgraph of the Hasse diagram of the original instance which is guaranteed to contain the maximum. However, for instances where PP returns a subgraph with a single node, that node *is* the maximum, and the instance is said to have been solved in polynomial time. Instances such as these make up the class of polynomially solvable instances that we consider here.

Let z be a real-valued function defined on the power set 2^N of $N = \{1, 2, \ldots, n\}$; $n \geq 1$. For each $S, T \in 2^N$ with $S \subseteq T$, we define

$$[S,T] = \{I \in 2^N \mid S \subseteq I \subseteq T\}.$$

Note that $[\emptyset, N] = 2^N$. Any *interval* $[S,T]$ is a subinterval of $[\emptyset, N]$ if $\emptyset \subseteq S \subseteq T \subseteq N$. We denote this using the notation $[S,T] \subseteq [\emptyset, N]$. In this section an interval is always a subinterval of $[\emptyset, N]$. It is assumed that z attains a finite maximum value on $[\emptyset, N]$ which is denoted by $z^*[\emptyset, N]$, and $z^*[S,T] = \max\{z(I) | I \in [S,T]\}$ for any $[S,T] \subseteq [\emptyset, N]$. We also define $d_k^+(I) = z(I+k) - z(I)$ and $d_k^-(I) = z(I-k) - z(I)$.

The following theorem and corollaries from Goldengorin *et al.* [14] act as a basis for the Preliminary Preservation (PP) algorithm described therein.

Theorem 5.2: Let z be a submodular function on $[S,T] \subseteq [\emptyset, N]$ and let $k \in T \backslash S$. Then the following assertions hold.

1. $z^*[S+k,T] - z^*[S,T-k] \leq z(S+k) - z(S)) = d_k^+(S)$.
2. $z^*[S,T-k] - z^*[S+k,T] \leq z(T-k) - z(T) = d_k^-(T)$.

Corollary 5.3: (Preservation rules of order zero). Let z be a submodular function on $[S,T] \subseteq [\emptyset, N]$, and let $k \in T \backslash S$. Then the following assertions hold.

1. First Preservation Rule: If $d_k^+(S) \leq 0$, then $z^*[S,T] = z^*[S,T-k] \geq z^*[S+k,T]$.
2. Second Preservation Rule: If $d_k^-(T) \leq 0$, then $z^*[S,T] = z^*[S+k,T] \geq z^*[S,T-k]$.

The PP algorithm accepts an interval $[S,T]$, $S \subseteq T$ and tries to apply Corollary 5.3 repeatedly. It returns an interval $[X,Y]$, $S \subseteq X \subseteq Y \subseteq T$, such that $z^*[S,T] = z^*[X,Y]$. The pseudocode for this algorithm is given below.

Algorithm PP($[S,T]$)
Output: A subinterval of $[S,T]$ containing the maximum of z over $[S,T]$.

Code:
1. begin
2. if $T = S$ return $[S,S]$;

```
3.      while T ≠ S do begin
4.          d⁺_max := max{d⁺_k | k ∈ T \ S};
5.          d⁻_max := max{d⁻_k | k ∈ T \ S};
6.          if d⁺_max ≤ 0 then begin
7.              k⁺_max := arg min{k ∈ T \ S | d⁺_k = d⁺_max};
8.                  T := T − k⁺_max;
9.          end
10.          else if d⁻_max ≤ 0 then begin
11.              k⁻_max := arg min{k ∈ T \ S | d⁻_k = d⁻_max};
12.              S := S + k⁻_max;
13.          end
14.          else return [S, T];
15.      end;
16. end.
```

The PP algorithm is called repeatedly by the DCA-MSF to generate a solution to the MSF instance within the prescribed accuracy level α. The pseudocode for DCA-MSF is given below. As in the case of ATSP, a good problem-specific upper bound will improve the performance of the algorithm.

Algorithm DCA-MSF$([S,T],\alpha)$
Output: $x^\alpha \in [S,T]$ such that $z(x^\alpha) \geq z^*[S,T] - \alpha$.

Code:

```
1. begin
2.      [S, T] := PP([S, T]);
3.      if T = S return S;
4.      d⁺_max := max{d⁺_k | k ∈ T \ S};
5.      d⁻_max := max{d⁻_k | k ∈ T \ S};
6.      if d⁺_max ≤ d⁻_max then begin
7.          if d⁺_max ≤ α then begin
8.              k⁺_max := arg min{k ∈ T \ S | d⁺_k = d⁺_max};
9.              return DCA-MSF([S, T − k⁺_max], α − d⁺_max) (* Correction *)
10.         end;
11.         else begin                                    (* Branch *)
12.             x1 := DCA-MSF([S + k⁺_max, T], α);
13.             x2 := DCA-MSF([S, T − k⁺_max], α);
14.             if z(x1) ≥ z(x2) return x1
```

```
15.                 else return x_2;
16.             end;
17.         end
18.         else begin
19.             if d^-_max ≤ α then begin
20.                 k^-_max := arg min{k ∈ T \ S | d^-_k = d^-_max};
21.                 return DCA-MSF([S + k^-_max, T], α − d^-_max) (* Correction*)
22.             end;
23.             else begin                                        (* Branch *)
24.                 x_1 := DCA-MSF([S + k^-_max, T], α);
25.                 x_2 := DCA-MSF([S, T − k^-_max], α);
26.                 if z(x_1) ≥ z(x_2) return x_1
27.                 else return x_2;
28.             end;
29.         end;
30. end.
```

5.2 A Data Correcting Algorithm based on Multi-Level Search

The preservation rules mentioned in Corollary 5.3 look at a level which is exactly one level deeper in the Hasse diagram than the levels of S and T. However, instead of looking one level deep we may look r levels deep in order to determine whether we can include or exclude an element. Let

$$\begin{aligned} M_r^+[S,T] &= \{I \in [S,T] \mid |I \backslash S| \leq r\}, \\ M_r^-[S,T] &= \{I \in [S,T] \mid |T \backslash I| \leq r\}. \end{aligned}$$

The set $M_r^+[S,T]$ is a collection of all sets representing solutions containing more elements than S, and which are no more than r levels deeper than S in the Hasse diagram. Similarly, the set $M_r^-[S,T]$ is a collection of all sets representing solutions containing less elements than T, and which are no more than r levels deeper than T in the Hasse diagram. Let us further define the collections of sets

$$\begin{aligned} N_r^+[S,T] &= M_r^+[S,T] \backslash M_{r-1}^+[S,T], \\ N_r^-[S,T] &= M_r^-[S,T] \backslash M_{r-1}^-[S,T]. \end{aligned}$$

The sets $N_r^+[S,T]$ and $N_r^-[S,T]$ are the collection of sets which are located exactly r levels above S and below T in the Hasse diagram, respectively.

Further, let $v_r^+[S,T] = \max\{z(I)|I \in M_r^+[S,T]\}$, $v_r^-[S,T] = \max\{z(I)|I \in M_r^-[S,T]\}$, $w_{rk}^+[S,T] = \max\{d_t^+(I)|I \in N_r^+[S+k,T]\}$ and $w_{rk}^-[S,T] = \max\{d_t^-(I)|I \in N_r^-[S,T-k]\}$.

Theorem 5.4: Let z be a submodular function on $[S,T] \subseteq [\emptyset,N]$ with $k \in T\backslash S$ and let r be a positive integer. Then the following assertions hold.

1. If $|N_r^+[S+k,T]| > 0$, then $z^*[S+k,T] - \max\{z^*[S,T-k], v_r^+[S,T]\} \leq \max\{w_{rk}^+[S,T], 0\}$.
2. If $|N_r^-[S,T-k]| > 0$, then $z^*[S,T-k] - \max\{z^*[S+k,T], v_r^-[S,T]\} \leq \max\{w_{rk}^-[S,T], 0\}$.

Proof: We prove only part 1 since the proof of the part 2 is similar. We may represent the partition of interval $[S,T]$ as follows:

$$[S,T] = M_r^+[S,T] \cup \bigcup_{I \in N_r^+[S,T]} [I,T].$$

Using this representation on the interval $[S+k,T]$, we have $z^*[S+k,T] = \max\{v_r^+[S+k,T], \max\{z^*[I+k,T]|I \in N_r^+[S,T]\}\}$. Let $I(k) \in \arg\max\{z^*[I+k,T]|I \in N_r^+[S,T]\}$.

There are two cases to consider: $z^*[I(k)+k,T] \geq v_r^+[S+k,T]$, and $z^*[I(k)+k,T] < v_r^+[S+k,T]$.

In the first case $z^*[S+k,T] = z^*[I(k)+k,T]$. For $I(k) \in N_r^+[S,T]$ we can apply Theorem 5.2(a) on the interval $[I(k),T]$ to obtain $z^*[I(k)+k,T] - z^*[I(k),T-k] \leq d_k^+(I(k))$, so that in this case $z^*[S+k,T] - z^*[I(k),T-k] \leq d_k^+(I(k))$. Note that for $[I(k),T-k] \subseteq [S,T-k]$ we have $z^*[S,T-k] \geq z^*[I(k),T-k]$, which implies that $z^*[S+k,T] - z^*[S,T-k] \leq d_k^+(I(k))$. Adding two maximum operations we get

$$z^*[S+k,T] - \max\{z^*[S,T-k], v_r^+[S+k,T]\} \leq \max\{d_k^+(I(k)), 0\}.$$

Since $w_{rk}^+[S,T]$ is the maximum of $d_k^+(I)$ for $I \in N_r^+[S+k,T]$, we have the required result.

In the second case $z^*[S+k,T] = v_r^+[S+k,T]$ which implies that $z^*[S+k,T] - v_r^+[S+k,T]\} = 0$ or $z^*[S+k,T] - \max\{z^*[S,T-k], v_r^+[S+k,T]\} \leq 0$. Adding a maximum operation with $w_{rk}^+[S,T]$ completes the proof. ■

Corollary 5.5: (Preservation rules of order r). Let z be a submodular function on $[S,T] \subseteq [\emptyset, N]$ and let $k \in T \backslash S$. Then the following assertions hold.

1. First Preservation Rule of Order r: If $w^+_{rk}[S,T] \leq 0$, then $z^*[S,T] = \max\{z^*[S,T-k], v^+_r[S+k,T]\} \geq z^*[S+k,T]$.
2. Second Preservation Rule of Order r: If $w^-_{rk}[S,T] \leq 0$, then $z^*[S,T] = \max\{z^*[S+k,T], v^-_r[S,T-k]\} \geq z^*[S,T-k]$

Notice that when we apply Corollary 5.3 to an interval, we get a reduced interval, however, when we apply Corollary 5.5, we get a value v_r in addition to a reduced interval.

It can be proved by induction that the portion of the Hasse diagram eliminated by preservation rules of order $r-1$ while searching for a maximum of the submodular function will certainly be eliminated by preservation rules of order r. In this sense, preservation rules of order r are not weaker than preservation rules of order $r-1$. (A detailed proof for the result that preservation rules of order 1 are not weaker than preservation rules of order 0, refer to Goldengorin [17]).

In order to apply Corollary 5.5, we need functions that compute the value of $w^+_{rk}[S,T]$, $w^-_{rk}[S,T]$ $v^+_r[S+k,T]$, and $v^-_r[S,T-k]$. To that end, we define two recursive functions, *PPArplus* to compute $w^+_{rk}[S,T]$ and $v^+_r[S+k,T]$, and *PPArminus* to compute $w^-_{rk}[S,T]$ and $v^-_r[S,T-k]$. The pseudocode for *PPArplus* is shown below. Its output is a 3-tuple, containing, in order, $w^+_{rk}[S,T]$ and $v^+_r[S+k,T]$, and a solution in $M^+_r[S+k,T]$ whose objective function value is $v^+_r[S+k,T]$. The pseudocode for *PPArminus* can be constructed in a similar manner.

---

function PPArplus$([S,T], r, k)$

1. begin
2. $w := -\infty$;
3. $v := -\infty$;
4. $vset := \emptyset$;
5. $(w, v, vset) :=$ IntPPArPlus$([S+k,T], r, w, v, vset)$;
6. return $(w, v, vset)$;
7. end;

function IntPPArplus$([X,Y], r, w, v, vset)$

1. begin
2. for each $t \in Y \setminus X$ do begin

```
3.          if z(X + t) > v then begin
4.              v := z(X + t);
5.              vset := (X + t);
6.          end;
7.          if d_t^+(X + t) > w then w := d_t^+(X + t);
8.          if d_t^+(X + t) > 0 and r > 1 then
9.              (w, v, vset) := IntPPArPlus([X + t, Y], r − 1, w, v, vset);
10.     end;
11.     return (w, v, vset);
12. end;
```

Note that *PPArplus* and *PPArminus* are both $\mathcal{O}(n\binom{n}{r})$, i.e. polynomial for a fixed value of r. However, in general, they are not polynomial in r.

We now use *PPArplus* and *PPArminus* to describe the Preliminary Preservation Algorithm of order r (PPAr(r)). Given a submodular function z on $[X, Y] \subseteq [\emptyset, N]$, PPAr outputs a subinterval $[S, T]$ of $[X, Y]$ and a set B such that $z^*[X, Y] = \max\{z^*[S, T], z(B)\}$ and $\min\{w^+_{rk}[S, T], w^-_{rk}[S, T]\} > 0$ for all $k \in T \backslash S$. At iteration i of the algorithm when the search has been restricted to $[S_i, T_i]$, starts by applying the PP algorithm (from Goldengorin *et al.* [14]) to this interval and reducing it to $[S'_i, T'_i]$. If $|T'_i \setminus S'_i| > 0$, an element $k \in T'_i \backslash S'_i$ is chosen, and the algorithm tries to apply Corollary 5.5.1 to decide whether it belongs to the set that maximizes $z(\cdot)$ over $[S_i, T_i]$ or not. If it does, then the search is restricted to the interval $[S'_i + k, T'_i]$. Otherwise, the search tries to apply Corollary 5.5.2 to decide whether the interval can be reduced to $[S'_i, T'_i - k]$.

Algorithm PPAr$([S, T], r)$
Output: $x^\alpha \in [S, T]$ such that $z(x^\alpha) \geq z^*[S, T] - \alpha$.

Code:

```
1. begin
2.     X := S, Y := T; B := arg max{z(S), z(T)};
3.     while Y ≠ X do begin
4.         [S_i, T_i] := PP([X, Y]);
5.         d^+ := max{d_k^+(S) | k ∈ T\S};
6.         d^- := max{d_k^-(T) | k ∈ T\S};
7.         if d^+ > d^- then begin
8.             k := arg max{d_t^+(S) | t ∈ T\S};
```

```
9.                    (w, v, vset) := PPArplus([S_i, T_i], r, k);
10.                   if v > z(B) then B := vset;
11.                   if w ≤ 0 then Y := T_i − k;
12.                   else return ([S_i, T_i], B);
13.             else begin
14.                   k := arg max{d_t^-(S) | t ∈ T\S};
15.                   (w, v, vset) := PPArminus([S_i, T_i], r, k);
16.                   if v > z(B) then B := vset;
17.                   if w ≤ 0 then X := S_i + k;
18.                   else return ([S_i, T_i], {w_ri^+[S_i, T_i]}, {w_ri^-[S_i, T_i]}, B);
19.             end;
20.       end;
21. end.
```

It is clear that if $r = |T \setminus S|$, PPAr will always find an optimal solution to our problem. However, PPAr is not a polynomial in r, and so PPAr with a large r is not practically useful.

We can embed PPAr in a branch and bound framework to describe DCA-MSFr, a data correcting algorithm based on PPAr. It is similar to the DCA-MSF proposed in Goldengorin *et al.* [14]. For DCA-MSFr we are given a submodular function z to be maximized over an interval $[S, T]$, and an accuracy parameter α, and we need to find a solution such that the difference between the objective function values of the solution output by DCA-MSFr and the optimal solution will not exceed α.

Notice that for a submodular function z, PPAr with a fixed r may terminate with $T \neq S$ and $\min\{w_{ri}^+[S,T], w_{ri}^-[S,T] \mid i \in T \setminus S\} = \omega > 0$. The basic idea behind DCA-MSFr is that if this situation occurs, then the data of the current problem is corrected in such a way that ω is non-positive for the corrected function and PPAr can continue. Moreover, each correction of z needs to be carried out in such a way that the corrected function remains submodular. The attempted correction is carried out implicitly, in a manner similar to the one in Goldengorin *et al.* [14] but using Corollary 5.5 instead of Corollary 5.3. Thus, for example, if $w_{rj}^+[S,T] = \omega \leq \alpha$, then PPAr is allowed to continue, but the accuracy parameter reduced to $\alpha - \omega$.

If such a correction is not possible, i.e. if ω exceeds the accuracy parameter, then we branch on a variable $k \in \arg\max\{d_i^+(S), d_i^-(T) | i \in T \setminus S\}$ to partition the interval $[S,T]$ into two intervals $[S+k, T]$ and $[S, T-k]$. This branching rule was proposed in Goldengorin [10]. An upper bound for the

value of z for each of the two intervals is then computed to see if either of the two can be pruned. We use an upper bound due to Khachaturov [23] described as follows. Let $d^+(S,T) = \{d_i^+(S) | d_i^+(S) > 0, i \in T \backslash S\}$ and $d^-(S,T) = \{d_i^-(T) | d_i^-(T) > 0, i \in T \backslash S\}$. Further let $d^+[i]$ (respectively $d^-[i]$) denote the ith largest element of $d^+(S,T)$ (respectively $d^-(S,T)$). Then ub described below is an upper bound to $z^*[S,T]$.

$$ub[S,T] = \max\{\min_{i=1,\ldots,|T\backslash S|}\{z(S) + \sum_{j=1}^{i} d^+[j], z(T) + \sum_{j=1}^{i} d^-[j]\}\}.$$

The following pseudocode describes DCA-MSFr formally.

Algorithm DCA-MSFr$([S,T], \alpha, r)$

```
1. begin
2.       best_set := arg max{z(S), z(T)};
3.       best := z(best_set);
4.       (best_set, best) := IntDCA-MSFr([S,T], α, r, best_set, best);
5.       return best_set;
6. end.
```

function IntDCA-MSFr$([S,T], \alpha, r, best\_set, best)$

```
1. begin
2.       ([S,T], {w+_rk}, {w-_rk}, B] :=PPAr([S,T], r);
3.       if z(B) > best then begin
4.             best_set := B;
5.             best := z(B);
6.       end;
7.       if S = T return (best_set, best);
8.       ω+ := max{w+_rk[S,T] | k ∈ T \ S};
9.       choose j+ from min{k | w+_rk[S,T] = ω+, k ∈ T \ S};
10.      ω- := max{w-_rk[S,T] | k ∈ T \ S};
11.      choose j+ from j- := min{k | w-_rk[S,T] = ω-, k ∈ T \ S};
12.      if ω+ ≤ α then                                   (* Correction *)
13.            IntDCA-MSFr([S + j+, T], α − ω+, r, best_set, best);
14.      else if ω- ≤ α then                              (* Correction *)
15.            IntDCA-MSFr([S, T − j-], α − ω-, r, best_set, best);
16.      else begin                  (* Branch [S,T] → [S + k, T], [S, T − k] *)
17.            choose k from arg max{d+_i(S), d-_i(T) | i ∈ T \ S};
18.            if ub[S + k, T] > best then begin          (* Bound *)
19.                  (bs1, b1) :=IntDCA-MSFr([S + k, T], α, r, best_set, best);
```

```
20.                     if b_1 > best then begin
21.                         best_set := bs_1;
22.                         best := b_1;
23.                     end;
24.                 end;
25.                 if ub[S, T − k] > best then begin                (* Bound *)
26.                     (bs_2, b_2) :=IntDCA-MSFr([S, T − k], α, r, best_set, best);
27.                     if b_2 > best then begin
28.                         best_set := bs_2;
29.                         best := b_2;
30.                     end;
31.                 end;
32.         end;
33. end;
```

5.3 Computational Experience with Quadratic Cost Partition Instances

In this section we report our computational experience with DCA-MSFr. We choose the quadratic cost partition problem as a test bed. The quadratic cost partition (QCP) problem can be described as follows (see e.g., Lee *et al.* [27]). Given nonnegative real numbers q_{ij} and real numbers p_i with $i, j \in N = \{1, 2, \ldots, n\}$, the QCP is the problem of finding a subset $S \subseteq N$ such that the function $z(S) = \sum_{j \in S} p_i - \frac{1}{2} \sum_{i,j \in S} q_{ij}$ will be maximized. The density d of a QCP instance is the ratio of the number of finite q_{ij} values to $n(n-1)/2$, and is expressed as a percentage. It is proved in Theorem 2.2 of Lee *et al.* [27] that $z(\cdot)$ is submodular.

In Goldengorin *et al.* [14] computational experiments with QCP have been restricted to instances of size not more than 80, because instances of that size have been considered in Lee *et al.* [27]. For these instances, it was shown that the average calculation times grow exponentially when the number of vertices increases and reduce exponentially with increasing density.

Herein we report the performance of DCA-MSFr on QCP instances of varying size and densities. The maximum time that we allow for an instance is 10 CPU minutes on an personal computer running on a 300MHz Pentium processor with 64 MB memory. The algorithms have been implemented in Delphi 3.

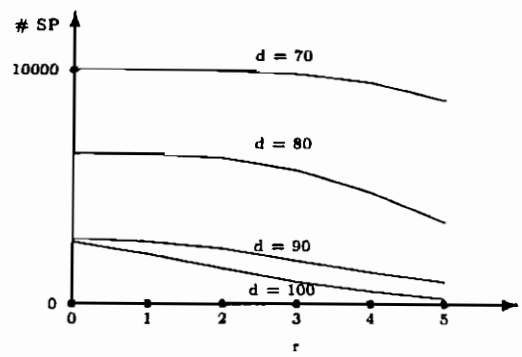


Fig. 8: Average number of subproblems generated against r for QCP instances with $n = 100$ and varying d values

The instances we test our algorithms on are statistically similar to the instances in Lee *et al.*[27]. Instances of size n and density $d\%$ are generated as follows. A graph with n nodes and $\frac{d}{100} \times \frac{n(n-1)}{2}$ random edges is generated. The edges are assigned costs from a $\mathcal{U}[1, 100]$ distribution. n edges connect each node to itself, and these edges are assigned costs from a $\mathcal{U}[0, 100]$ distribution. The distance matrix of this graph forms a QCP instance.

We first report the effect of varying the value of r on the performance of DCA-MSFr(r). It is intuitive that DCA-MSFr(r) will require more execution times when the value of r increases. Our computation experience with 10 QCP instances of size 100 and different densities is shown in Figures 8-10. Figure 8 shows the number of subproblems generated when r is increased from a value of 0 (i.e. DCA-MSF) to 5. As is intuitive, the number of subproblems reduce with increasing r for all density values. Figure 9 shows the execution times of DCA-MSFr(r) with varying d and r values. Recall that when the value of r increases, the time required at each subproblem increases, since PPAr requires more computations for larger r values. The decrease in the number of subproblems approximately balance the increase in the time at each subproblem for r values in the range 0 through 4. When $r = 5$, the computation times for DCA-MSFr(r) increase significantly for all densities. From Figure 9 it seems that for dense graphs, r values of 3 or 4 are most favorable. This effect also holds for larger instances — Figure 10 shows the execution times for instances with $n = 200$ and $d = 100$.

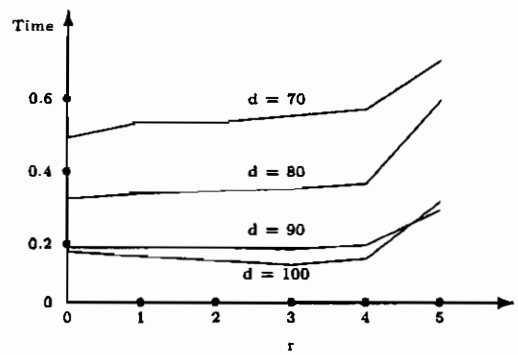


Fig. 9: Average execution time (in seconds) against r for QCP instances with $n = 100$ and varying d values

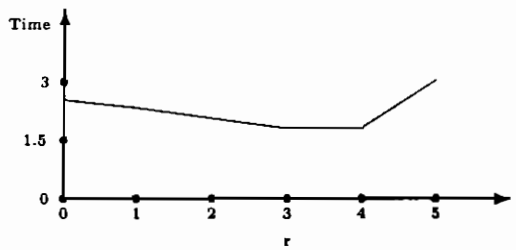


Fig. 10: Average execution time (in seconds) against r for QCP instances with $n = 200$ and $d = 100$

We next report the results of our experiments to solve large sized QCP instances with DCA-MSFr(r). Using results obtained from the previous part of our study, we choose to use DCA-MSFr(3) as our algorithm of choice. We consider instances of the QCP with size n ranging from 100 to 500 and densities varying between 10% and 100%. We try to solve these instances exactly ($\alpha_0 = 0\%$), and with a prescribed accuracy $\alpha_0 = 5\%$ within 10 minutes. We report in Tables 1 and 2 the average execution times in seconds for exact and approximate solutions with DCA-MSFr(3) and DCA-MSF. The entries marked '*' could not be solved within 10 minutes. From the table, we note that the execution times increase exponentially with increasing problem size and decreasing problem densities. Therefore QCP instances with 500 vertices and densities between 90% and 100% are the largest instances which can be solved by the DCA-MSFr(3) within 10 minutes on a standard personal computer. We also see that on an average DCA-MSFr(3) takes roughly 11% of the time taken by DCA-MSF for the exact solutions, and roughly 13% of the time taken by DCA-MSF for the approximate solutions. The reduction in time is more pronounced for problems with higher size and higher densities.

Tab. 1: Average execution times on QCP instances when $\alpha = 0\%$

		Problem size (n)				
Density(%)	Algorithm	100	200	300	400	500
100	DCA-MSFr(3)	0.098	2.63	18.316	85.827	229.408
	DCA-MSF	0.831	64.372	340.681	1894.811	*
90	DCA-MSFr(3)	0.138	3.824	37.931	173.063	624.925
	DCA-MSF	1.027	78.614	794.037	3505.892	*
80	DCA-MSFr(3)	0.28	9.506	98.69	679.914	*
	DCA-MSF	1.784	217.898	2681.973	*	*
70	DCA-MSFr(3)	0.393	17.643	413.585	*	*
	DCA-MSF	2.498	631.492	*	*	*
60	DCA-MSFr(3)	0.731	86.33	*	*	*
	DCA-MSF	3.509	1414.103	*	*	*
50	DCA-MSFr(3)	1.752	345.723	*	*	*
	DCA-MSF	9.382	*	*	*	*
40	DCA-MSFr(3)	3.457	*	*	*	*
	DCA-MSF	17.245	*	*	*	*
30	DCA-MSFr(3)	11.032	*	*	*	*
	DCA-MSF	48.013	*	*	*	*
20	DCA-MSFr(3)	47.162	*	*	*	*
	DCA-MSF	195.82	*	*	*	*
10	DCA-MSFr(3)	70.081	*	*	*	*
	DCA-MSF	446.293	*	*	*	*

Tab. 2: Average execution times on QCP instances when $\alpha = 5\%$

		Problem size (n)				
Density(%)	Algorithm	100	200	300	400	500
100	DCA-MSFr(3)	0.094	2.444	17.179	85.096	222.883
	DCA-MSF	0.752	38.351	229.396	1162.396	*
90	DCA-MSFr(3)	0.118	3.607	34.972	166.996	608.755
	DCA-MSF	0.916	49.926	583.754	1996.544	*
80	DCA-MSFr(3)	0.228	8.186	89.685	580.789	*
	DCA-MSF	1.108	162.455	1875.603	3604.715	*
70	DCA-MSFr(3)	0.304	15.693	364.48	*	*
	DCA-MSF	1.593	376.629	3165.384	*	*
60	DCA-MSFr(3)	0.517	72.931	*	*	*
	DCA-MSF	2.874	895.426	*	*	*
50	DCA-MSFr(3)	1.298	267.445	*	*	*
	DCA-MSF	5.931	1937.673	*	*	*
40	DCA-MSFr(3)	2.179	*	*	*	*
	DCA-MSF	10.327	*	*	*	*
30	DCA-MSFr(3)	5.88	*	*	*	*
	DCA-MSF	22.209	*	*	*	*
20	DCA-MSFr(3)	17.477	*	*	*	*
	DCA-MSF	74.841	*	*	*	*
10	DCA-MSFr(3)	12.196	*	*	*	*
	DCA-MSF	95.122	*	*	*	*

6 The Simple Plant Location Problem

The Simple Plant Location Problem (SPLP) takes a set $I = \{1, 2, \ldots, m\}$ of sites in which plants can be located, a set $J = \{1, 2, \ldots, n\}$ of clients, each having a unit demand, a vector $F = (f_i)$ of fixed costs for setting up plants at sites $i \in I$, and a matrix $C = [c_{ij}]$ of transportation costs from $i \in I$ to $j \in J$ as input. It computes a set P^*, $\emptyset \subset P^* \subseteq I$, at which plants can be located so that the total cost of satisfying all client demands is minimal. The costs involved in meeting the client demands include the fixed costs of setting up plants, and the transportation cost of supplying clients from the plants that are set up. A detailed introduction to this problem has appeared in Cornuejols *et al.* [6], which also classifies the problem as NP-hard. The objective function of the SPLP is supermodular, but we do not use the results of the previous section explicitly in this section.

In applying data correcting to the SPLP, we work with a pseudo-Boolean formulation of the problem. We show how data correcting can be used to preprocess SPLP instances efficiently, and then to solve the problem.

6.1 A Pseudo-Boolean Formulation of the SPLP

The pseudo-Boolean approach to solving the SPLP (Hammer [21], Beresnev [4]) is a penalty-based approach that relies on the fact that any instance of the SPLP has an optimal solution in which each client is supplied by exactly one plant. This implies, that in an optimal solution, each client will be served fully by the plant located closest to it. Therefore, it is sufficient to determine the sites where plants are to be located, and then use a minimum cost assignment of clients to plants.

An instance of the SPLP can be described by a m-vector $F = (f_i)$, and a $m \times n$ matrix $C = [c_{ij}]$; $m, n \geq 1$. We will use the $m \times (n+1)$ *augmented matrix* $[F|C]$ as a shorthand for describing an instance of the SPLP. The total cost $f_{[F|C]}(P)$ associated with a subset P of I consists of two components, namely the fixed costs $\sum_{i \in P} f_i$ and the transportation costs $\sum_{j \in J} \min\{c_{ij} | i \in P\}$; i.e.

$$f_{[F|C]}(P) = \sum_{i \in P} f_i + \sum_{j \in J} \min\{c_{ij} | i \in P\},$$

and the SPLP is the problem of finding

$$P^* \in \arg\min\{f_{[F|C]}(P) | \emptyset \subset P \subseteq I\}. \tag{4}$$

In the remainder of this subsection we describe the pseudo-Boolean formulation of the SPLP due to Hammer [21].

A $m \times n$ *ordering matrix* $\Pi = [\pi_{ij}]$ is a matrix each of whose columns $\Pi_j = (\pi_{1j}, \ldots, \pi_{mj})^T$ define a permutation of $1, \ldots, m$. Given a transportation matrix C, the set of all ordering matrices Π such that $c_{\pi_{1j}j} \leq c_{\pi_{2j}j} \leq \cdots \leq c_{\pi_{mj}j}$ for $j = 1, \ldots, n$, is denoted by $perm(C)$.

Defining for each $i = 1, \ldots, m$

$$y_i = \begin{cases} 0 & \text{if } i \in P \\ 1 & \text{otherwise,} \end{cases} \tag{5}$$

we can indicate any solution P by a vector $\overline{y} = (y_1, y_2, \ldots, y_m)$. The fixed cost component of the total cost can be written as

$$F_F(\overline{y}) = \sum_{i=1}^{m} f_i(1 - y_i). \tag{6}$$

Given a transportation cost matrix C, and an ordering matrix $\Pi \in perm(C)$, we can denote differences between the transportation costs for each $j \in J$ as

$$\begin{aligned}\Delta c[0,j] &= c_{\pi_{1j}j}, \quad \text{and} \\ \Delta c[l,j] &= c_{\pi_{(l+1)j}j} - c_{\pi_{lj}j}, \quad l = 1,\ldots,m-1.\end{aligned}$$

Note that $\Delta c[l,j] \geq 0$, even if the transportation cost matrix C contains negative entries. The transportation costs of supplying any client $j \in J$ from any open plant can be expressed in terms of the $\Delta c[\cdot,j]$ values. It is clear that we have to spend at least $\Delta c[0,j]$ in order to satisfy j's demand, since this is the cheapest cost of satisfying j. If no plant is located at the site closest to j, i.e. $y_{\pi_{1j}} = 1$, we try to satisfy the demand from the next closest site. In that case, we spend an additional $\Delta c[1,j]$. Continuing in this manner, the transportation cost of supplying $j \in J$ is

$$\begin{aligned}\min\{c_{ij} | i \in P\} &= \Delta c[0,j] + \Delta c[1,j] \cdot y_{\pi_{1j}} + \Delta c[2,j] \cdot y_{\pi_{1j}} \cdot y_{\pi_{2j}} \\ &\quad + \cdots + \Delta c[m-1,j] \cdot y_{\pi_{1j}} \cdots y_{\pi_{(m-1)j}} \\ &= \Delta c[0,j] + \sum_{k=1}^{m-1} \Delta c[k,j] \cdot \prod_{r=1}^{k} y_{\pi_{rj}},\end{aligned}$$

so that the transportation cost component of the cost of a solution $\overline{y}$ corresponding to an ordering matrix $\Pi \in perm(C)$ is

$$T_{C,\Pi}(\overline{y}) = \sum_{j=1}^{n} \Big\{ \Delta c[0,j] + \sum_{k=1}^{m-1} \Delta c[k,j] \cdot \prod_{r=1}^{k} y_{\pi_{rj}} \Big\}. \tag{7}$$

Combining (6) and (7), the total cost of a solution $\overline{y}$ to the instance $[F|C]$ corresponding to an ordering matrix $\Pi \in perm(C)$ is given by the pseudo-Boolean polynomial

$$\begin{aligned}f_{[F|C],\Pi}(\overline{y}) &= F_F(\overline{y}) + T_{C,\Pi}(\overline{y}) \\ &= \sum_{i=1}^{m} f_i(1-y_i) + \\ &\quad \sum_{j=1}^{n} \Big\{ \Delta c[0,j] + \sum_{k=1}^{m-1} \Delta c[k,j] \cdot \prod_{r=1}^{k} y_{\pi_{rj}} \Big\}. \end{aligned} \tag{8}$$

It can be shown (see Goldengorin *et al.* [15]) that the total cost function $f_{[F|C],\Pi}(\cdot)$ is identical for all $\Pi \in perm(C)$. We call this pseudo-Boolean

polynomial the *Hammer function* $H_{[F|C]}(\overline{y})$ corresponding to the SPLP instance $[F|C]$ and $\Pi \in perm(C)$. In other words

$$H_{[F|C]}(\overline{y}) = f_{[F|C],\Pi}(\overline{y}) \text{ where } \Pi \in perm(C). \tag{9}$$

We can formulate (4) in terms of Hammer functions as

$$\overline{y^*} \in \arg\min\{H_{[F|C]}(\overline{y}) | \overline{y} \in \{0,1\}^m, \overline{y} \neq \overline{1}\}. \tag{10}$$

As an example, consider the SPLP instance:

$$[F|C] = \left[\begin{array}{c|cccc} 9 & 7 & 12 & 22 & 13 \\ 4 & 8 & 9 & 18 & 17 \\ 3 & 16 & 17 & 10 & 27 \\ 6 & 9 & 13 & 10 & 11 \end{array}\right]. \tag{11}$$

Two possible ordering matrices corresponding to C are

$$\Pi_1 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 1 & 4 & 1 \\ 4 & 4 & 2 & 2 \\ 3 & 3 & 1 & 3 \end{bmatrix} \text{ and } \Pi_2 = \begin{bmatrix} 1 & 2 & 4 & 4 \\ 2 & 1 & 3 & 1 \\ 4 & 4 & 2 & 2 \\ 3 & 3 & 1 & 3 \end{bmatrix}. \tag{12}$$

The Hammer function is $H_{[F|C]}(\overline{y}) = \{9(1-y_1)+4(1-y_2)+3(1-y_3)+6(1-y_4)\} + \{7+1y_1+1y_1y_2+7y_1y_2y_4\} + \{9+3y_2+1y_1y_2+4y_1y_2y_4\} + \{10+0y_3+8y_3y_4+4y_2y_3y_4\} + \{11+2y_4+4y_1y_4+10y_1y_2y_4\} = 59-8y_1-y_2-3y_3-4y_4+2y_1y_2+4y_1y_4+8y_3y_4+21y_1y_2y_4+4y_2y_3y_4$.

6.2 Preprocessing SPLP instances

The first preprocessing rules for the SPLP involving both fixed costs and transportation costs appeared in Khumawala [24]. In terms of Hammer functions, these rules are stated in the following theorem. We assume (without loss of generality) that we cannot partition I into sets I_1 and I_2, and J into sets J_1 and J_2, such that the transportation costs from sites in I_1 to clients in J_2, and from sites in I_2 to clients in J_1 are not finite. We assume too, that the site indices are arranged in non-increasing order of $f_i + \sum_{j \in J} c_{ij}$ values.

Theorem 6.1: Let $H_{[F|C]}(\overline{y})$ be the Hammer function corresponding to the SPLP instance $[F|C]$ in which like terms have been aggregated. For each site index k, let a_k be the coefficient of the linear term corresponding to y_k and let be the sum of the coefficients of all non-linear terms containing y_k. Then the following assertion holds.

RO: If $a_k \geq 0$, then there is an optimal solution $\overline{y^*}$ in which $y_k^* = 0$, else

RC: If $a_k + t_k \leq 0$, then there is an optimal solution $\overline{y^*}$ in which $y_k^* = 1$, provided that $y_i^* \neq 1$ for some $i \neq k$.

Notice that RO and RC primarily try to either open or close sites. If it succeeds, it also changes the Hammer function for the instance, reducing the number of non-linear terms therein. In the remaining portion of this subsection, we describe a completely new reduction procedure (RP), whose primary aim is to reduce the coefficients of terms in the Hammer function, and if we can reduce it to zero, to eliminate the term from the Hammer function. This procedure is based on fathoming rules of branch and bound algorithms and data correcting principles.

Let us assume that we have an upper bound (UB) on the cost of an optimal solution for the given SPLP instance. This can be obtained by running a heuristic on the problem data. Now consider a non-linear term $s \cdot \prod_{r=1}^{k} y_{\pi_{rj}}$ in the Hammer function. This term will contribute to the cost of a solution, only if plants are *not* located in any of the sites $\pi_{1j}, \ldots, \pi_{kj}$. Let LB be a lower bound on the cost of solutions in which facilities are not located in sites $\pi_{1j}, \ldots, \pi_{kj}$. If $LB \leq UB$, then we cannot make any judgement about this term. On the other hand, if $LB > UB$, then we know that there cannot be an optimal solution with $y_{\pi_{1j}} = \ldots = y_{\pi_{kj}} = 1$. In this case, if we reduce the coefficient s by $LB - UB - \varepsilon$, ($\varepsilon > 0$, small), then the new Hammer function and the original one have identical sets of optimal solutions. If after the reduction, s is non-positive, then the term can be removed from the Hammer function. Such changes in the Hammer function alter the values of t_k, and can possibly allow us to use Khumawala's rules to close certain sites. Once some sites are closed, some of the linear terms in the Hammer function change into constant terms, and some of the quadratic terms change into linear ones. These changes cause changes in both the a_k and the t_k values, and can make further application of Khumawala's rules possible, thus preprocessing some other sites, and making further changes in the Hammer function. A pseudocode of the reduction procedure (RP) is provided below.

Procedure RP($H_{[F|C]}(\overline{y})$)
Output: A preprocessed instance of the SPLP, i.e. an equivalent instance of reduced size, and decisions to either locate or not locate plants in some of the sites.

Code:

```
1. begin
2.     repeat
3.         compute an upper bound UB for the instance;
4.         for each nonlinear term s · ∏_{r=1}^{k} y_{π_{rj}} in H_{[F|C]}(ȳ) do
5.         begin
6.             compute lower bound LB on the cost of solutions in
7.             which plants are not located in sites π_{1j}, ..., π_{kj};
8.             if LB > UB then
9.                 reduce the coefficient of the term by
10.                max{s, LB − UB − ε};
11.        apply Khumawala's rules until no further preprocessing is possible;
12.        recompute the Hammer function H_{[F|C]}(ȳ);
13.    until no further preprocessing of sites was achieved in the current iteration;
14. end;
```

Let us consider the application of all preprocessing rules to the example with the Hammer function $H_{[F|C]}(\overline{y}) = 59 - 8y_1 - y_2 - 3y_3 - 4y_4 + 2y_1y_2 + 4y_1y_4 + 8y_3y_4 + 21y_1y_2y_4 + 4y_2y_3y_4$. The values of a_k, t_k and $a_k + t_k$ are as follows:

k :	1	2	3	4
a_k :	−8	−1	−3	−4
t_k :	27	27	12	37
$a_k + t_k$:	19	26	9	33

It is clear that neither RO nor RC is applicable here, since the coefficient of the term $21y_1y_2y_4$ is too large. Therefore, we try to reduce this coefficient by applying the RP.

An upper bound of $UB = 51$ to the original problem can be obtained by setting $y_1 = y_4 = 1$ and $y_2 = y_3 = 0$. A lower bound to the problem under the restriction $y_1 = y_2 = y_4 = 1$ is 73, since $H_{[F|C]}(1, 1, 0, 1) = 73$. Using RP therefore, we can reduce the coefficient of $21y_1y_2y_4$ by $73 - 51 - \varepsilon = 20$, so that the new Hammer function with the same set of optimal solutions as the original function becomes $H'(\overline{y}) = 59 - 8y_1 - y_2 - 3y_3 - 4y_4 + 2y_1y_2 + 4y_1y_4 + 8y_3y_4 + 1y_1y_2y_4 + 4y_2y_3y_4$. The updated values of a_k, t_k, and $a_k + t_k$ are presented below.

k :	1	2	3	4
a_k :	-8	-1	-3	-4
t_k :	7	7	12	17
$a_k + t_k$:	-1	6	9	13

RC can immediately be applied in this situation to set $y_3 = 1$. Updating $H'(\overline{y})$, we can apply RO and set $y_2 = y_4 = 0$. This allows us to apply RC again to set $y_3 = 1$, thus giving us an optimal solution (i.e. $(1, 0, 1, 0)$) to the instance, with a cost of 48.

6.3 The Data Correcting Algorithm

The basic idea behind the data correcting algorithm is to modify the Hammer function in a way, such that the RO and RC rules can be applied to the modified instance. While modifying the instance, care is taken so that an optimal solution to the modified instance is not too sub-optimal for the original instance. We make use of the following suitably modified version of Lemma 3.1 for this problem.

Lemma 6.2: Consider two Hammer functions $H_1(\overline{y})$ and $H_2(\overline{y})$. Let $\overline{y_1^*}$ and $\overline{y_2^*}$ be the optimal solutions to $H_1(\overline{y})$ and $H_2(\overline{y})$ respectively. Then

$$H_1(\overline{y_1^*}) - H_1(\overline{y_2^*}) \leq \sum_{i=0}^{m-1} \sum_{j=1}^{n} |\Delta c^1[i,j] - \Delta c^2[i,j]|.$$

Consider a SPLP instance with accuracy parameter α in which RO and RC cannot be applied. Clearly, in the Hammer function for this instance, $a_k < 0$ and $a_k + t_k > 0$ for all k. Let $k_0 = \arg\min\{|a_k|, a_k + t_k\}$. Also let $|a_{k_0}| \leq a_{k_0} + t_{k_0}$. In this case, if we change (correct) the Hammer function of the instance by increasing the coefficient of y_{k_0} to zero, then RO can be applied to the corrected instance and preprocessing can continue. However, this is allowed only if $\min\{|a_{k_0}|, a_{k_0} + t_{k_0}\} \leq \alpha$. In such a situation, if $|a_{k_0}| > a_{k_0} + t_{k_0}$, then the instance can be corrected by decreasing the coefficient of y_{k_0} by $a_{k_0} + t_{k_0}$; then RC can be applied to the corrected instance and preprocessing can continue. Notice that while correcting the instance, we allow for suboptimality to the extent of $|a_{k_0}|$ in the first case, and $a_{k_0} + t_{k_0}$ in the second case. Thus, the accuracy parameter for the corrected instance is reduced appropriately.

It may happen however, that $\min\{|a_{k_0}|, a_{k_0} + t_{k_0}\} > \alpha$. In that case, correction is not possible at this stage and the problem has to be broken down into subproblems. This is done by a branching operation. Goldengorin *et al.* [16] suggest that the algorithm branches on an index from $\arg\max\{t_k\}$.

The logic behind this rule is the following. A plant would have been located in this site in an optimal solution if the coefficient of linear term involving y_k in the Hammer function would have been increased by $-a_k$. We could have predicted that a plant would not be located there if the same coefficient would have been decreased by $t_k + a_k$. Therefore we could use the average of $-a_k$ and $a_k + t_k$ as a measure of the chance that we will *not* be able to predict the fate of site k in any subproblem of the current subproblem. If we want to reduce the size of the branch and bound tree by assigning values to such variables, then we can think of a branching function that branches on the index k_0 with the largest average value, i.e. the largest value of $-a_k + (a_k + t_k)$, i.e. the largest value of t_k.

On the basis of the discussion above, the pseudocode for a data correcting algorithm for SPLP is given below. It works by maintaining three sets, Ω containing the sites where facilities are to be located, Λ containing the sites where facilities are not to be located, and Ψ containing the rest of the sites. RO and RC rules are assumed to be able to manipulate these three sets.

Algorithm DCA-SPLP$(H_{[F|C]}(\overline{y}), \alpha)$
Output: A solution $\overline{y^\alpha}$ to the SPLP such that $H_{[F|C]}(\overline{y^\alpha}) \leq H_{[F|C]}(\overline{y^*}) + \alpha$.

Code:

```
1. begin
2.       apply RP to initialize Ω, Λ and Ψ;
3.       y^α := Int-DCA-SPLP(Ω, Λ, Ψ, H_[F|C](·), α);
4.       return y^α;
5. end.
```

Function Int-DCA-SPLP$(\Omega, \Lambda, \Psi, H_{[F|C]}(\cdot), \alpha)$

```
1. begin
2.       while RO or RC is applicable
3.             update Ω, Λ, Ψ, and H_[F|C](·) by applying RO and RC;
4.       k* ∈ arg{k ∈ Ψ | min{|a_k|, a_k + t_k} =
...            min{min{|a_s|, a_s + t_s} | s ∈ Ψ}};
5.       if |a_k*| ≤ a_k* + t_k* then begin
6.             if |a_k*| ≤ α then                          (* Correct *)
7.                   return Int-DCA-SPLP(Ω + k*, Λ, Ψ − k*, H_[F|C](·), α − |a_k*|);
```

```
8.              else begin                                          (* Branch *)
9.                    k^b := arg max{t_k | k ∈ Ψ};
10.                   y_1 :=Int-DCA-SPLP(Ω + k^b, Λ, Ψ − k^b, H_[F|C](·), α);
11.                   y_2 :=Int-DCA-SPLP(Ω, Λ + k^b, Ψ − k^b, H_[F|C](·), α);
12.                   return arg max{H_[F|C](y_1), H_[F|C](y_2)};
13.             end;
14.       else begin
15.             if a_k* + t_k* ≤ α then                             (* Correct *)
16.       return Int-DCA-SPLP(Ω, Λ + k*, Ψ − k*, H_[F|C](·), α − a_k* − t_k*);
17.             else begin                                          (* Branch *)
18.                   k^b := arg max{t_k | k ∈ Ψ};
19.                   y_1 :=Int-DCA-SPLP(Ω + k^b, Λ, Ψ − k^b, H_[F|C](·), α);
20.                   y_2 :=Int-DCA-SPLP(Ω, Λ + k^b, Ψ − k^b, H_[F|C](·), α);
21.                   return arg max{H_[F|C](y_1), H_[F|C](y_2)};
22.             end;
23.       end;
24. end;
```

6.4 Computational Experience with SPLP Instances

We report our computational experience with the DCA-SPLP on several benchmark instances of the SPLP in the remainder of this section. The performance of the algorithm is compared with that of the algorithms described in the papers that suggested these instances. We used one of two bounds in the implementations of RP and DCA-SPLP: a combinatorial Khachaturov-Minoux bound (Khachaturov [22] and Minoux [29]); and a much stronger Erlenkotter bound based on a LP dual-ascent algorithm (Erlenkotter [7]). We implemented the DCA-SPLP in PASCAL, compiled it using Prospero Pascal, and ran it on a 733 MHz Pentium III machine. The computation times we report are in seconds on our machine.

6.4.1 Testing the Effectiveness of the Reduction Procedure RP

Given an instance of the SPLP, the reduction procedure RP reduces it to a smaller core instance by making decisions to locate or not locate plants in several sites. The effectiveness of the RP can thus be measured either by computing the number of free locations in the core instance, or by computing the number of non-zero nonlinear terms present in the Hammer function of the core instance. Tables 3 and 4 shows how the various methods of

reduction perform on the benchmark SPLP instances in the OR-Library (Beasley [3]). In the tables, procedure (a) refers to the use of the "delta" and "omega" rules from Khumawala [24], procedure (b) to the RP with the Khachaturov-Minoux combinatorial bound to obtain a lower bound, and procedure (c) to the RP with the Erlenkotter bound to obtain a lower bound.

Tab. 3: Number of free locations after preprocessing SPLP instances in the OR-Library

Problem	m	n	Procedure		
			a	b	c
cap71	16	50	4	0	0
cap72	16	50	6	0	0
cap73	16	50	6	3	3
cap74	16	50	2	0	0
cap101	25	50	9	0	0
cap102	25	50	13	3	0
cap103	25	50	14	0	0
cap104	25	50	12	0	0
cap131	50	50	34	32	8
cap132	50	50	27	25	5
cap133	50	50	25	19	10
cap134	50	50	19	0	0

The existing preprocessing rules due to Khumawala [24] and Goldengorin *et al.* [15] (i.e. procedure (a), which was used in the SPLP example in Goldengorin *et al.* [14]) cannot solve any of the OR-Library instances to optimality. However, the variants of the new reduction procedure (i.e. procedures (b) and (c)) solve a large number of these instances to optimality. Procedure (c), based on the Erlenkotter bound is marginally better than procedure (b) in terms of the number of free locations (Table 3), but substantially better in terms of the number of non-zero nonlinear terms in the Hammer function (Table 4).

Tables 3 and 4 also demonstrate the superiority of the new preprocessing rule over the "delta" and "omega" rules. Consider for example the problem cap132. The "delta" and "omega" rules reduce the problem size from $m = 50$ and 2389 non-zero nonlinear variables to $m = 27$ and 112 non-zero nonlinear variables. However, the new preprocessing rule reduces the same problem to one having $m = 5$ and 3 non-zero nonlinear variables!

Tab. 4: Number of non-zero nonlinear terms in the Beresnev function after preprocessing SPLP instances in the OR-Library

Problem	Non-zero terms before preprocessing	Procedure		
		a	b	c
cap71	699	6	0	0
cap72	699	12	0	0
cap73	699	13	2	2
cap74	699	1	0	0
cap101	1147	24	0	0
cap102	1147	33	2	0
cap103	1147	38	0	0
cap104	1147	29	0	0
cap131	2389	163	135	8
cap132	2389	112	92	3
cap133	2389	101	60	11
cap134	2389	62	0	0

6.4.2 Bilde and Krarup-type Instances

These are the earliest benchmark problems that we consider here. The exact instance data is not available, but the process of generating the problem instances is described in Bilde and Krarup [5]. There are 22 different classes of instances and in this subsection we use the nomenclature used in Bilde and Krarup [5]. In our experiments we generated 10 instances for each of the types of problems, and used the mean values of our solutions to evaluate the performance of our algorithm with the one used in Bilde and Krarup [5]. In our implementation, we used the Khachaturov-Minoux combinatorial bound in the reduction procedure RP as well as in the DCA-SPLP.

The reduction procedure was not useful for these instances, but the DCA-SPLP could solve all the instances in reasonable time. The results of our experiments are presented in Table 5. The performance of the algorithm implemented in Bilde and Krarup [5] was measured in terms of the number of branching operations performed by the algorithm and its execution time in CPU seconds on a IBM 7094 machine. We estimate the number of branching operations by our algorithm as the logarithm (to the base 2) of the number of subproblems it generated. From the table we see that the DCA-SPLP reduces the number of subproblems generated by the algorithm in Bilde and Krarup [5] by a factor of 1000. This is especially interesting because Bilde and Krarup use a bound (discovered in 1967) identical to the Erlenkotter bound in their algorithm (see Körkel [25]) and we use the Khachaturov-

Tab. 5: Results from Bilde and Krarup-type instances

Problem	DCA		Bilde and Krarup	
Type	Branching	CPU time	Branching	CPU Time†
B	11.72	0.67	43.3	4.33
C	17.17	14.81	⋆	>250
D1	13.80	0.65	216	11
D2	12.13	0.38	218	24
D3	10.87	0.19	169	19
D4	10.25	0.15	141	17
D5	9.24	0.07	106	14
D6	8.99	0.09	101	15
D7	8.79	0.09	83	13
D8	8.60	0.09	55	11
D9	8.15	0.07	47	11
D10	7.29	0.03	43	11
E1	18.66	35.28	1271	202
E2	16.14	8.64	1112	172
E3	14.59	3.81	384	82
E4	13.65	2.74	258	65
E5	12.73	2.01	193	53
E6	11.82	0.90	136	43
E7	10.82	0.53	131	42
E8	10.79	0.68	143	48
E9	10.62	0.76	117	44
E10	10.36	0.69	79	37

† IBM7094 seconds.
⋆ could not be solved in 250 seconds.

Minoux combinatorial bound. The CPU time required by the DCA-SPLP to solve these problems were too low to warrant the use of any $\alpha > 0$.

6.4.3 Galvão and Raggi-type Instances

Galvão and Raggi [8] developed a general 0-1 formulation of the SPLP and presented a 3-stage method to solve it. The benchmark instances suggested in this work are unique, in that the fixed costs are assumed to come from a Normal distribution rather than the more commonly used Uniform distribution. The reader is referred to Galvão and Raggi [8] for a detailed description of the problem data.

As with the data in Bilde and Krarup [5], the exact data for the instances are not known. So we generated 10 instances for each problem size, and used the mean values of the solutions for comparison purposes. In our DCA-SPLP implementation, we used the Khachaturov-Minoux combinatorial bound in

the reduction procedure RP and in the DCA-SPLP. The comparative results are given in Table 6. Since the computers used are different, we cannot make any comments on the relative performance of the solution procedures. However, since the average number of subproblems generated by the DCA-SPLP is always less than 10 for each of these instances, we can conclude that these problems are easy for our algorithm. In fact they are too easy for the DCA-SPLP to warrant $\alpha > 0$.

Tab. 6: Results from Galvão and Raggi-type instances

Problem	DCA				Galvão and Raggi	
Size ($m = n$)	# solved by pre-processing	# of sub-problems†	CPU time†	# of open plants†	CPU time*	# of open plants
10	6	2.3	<0.001	4.7	<1	3
20	5	2.4	<0.001	9.0	<1	8
30	7	1.8	0.002	13.6	1	11
50	7	2.6	0.002	20.3	2	20
70	2	3.8	0.004	28.8	6	31
100	3	3.5	0.011	41.1	6	44
150	1	7.8	0.010	64.4	25	74
200	4	2.9	0.158	81.8	63	84

† Average over 10 instances.
* IBM 4331 seconds.

Notice that the average number of opened plants in the optimal solutions to the instances we generated is quite close to the number of opened plants in the optimal solutions reported in Galvão and Raggi [8]. Also notice that the reduction procedure was quite effective — it solved 35 of the 80 instances generated.

6.4.4 Instances from the OR-Library

The OR-Library [3] has a set of instances of the SPLP. These instances were solved in Beasley [2] using an algorithm based on the Lagrangian heuristic for the SPLP. Here too, we used the Khachaturov-Minoux combinatorial bound in the reduction procedure RP as well as in the DCA-SPLP. We solved the problems to optimality using the DCA. The results of the computations are provided in Table 7. The execution times suggest that the DCA-SPLP is faster than the Lagrangian heuristic described in Beasley [2]. The reduction procedure was also quite effective for these instances, solving 4 of the 16 instances to optimality, and reducing the number of free sites appreciably in the other instances. Once again the use of $\alpha > 0$ cannot be justified,

considering the execution times of the DCA.

Tab. 7: Results from OR-Library instances

Problem name	m	n	DCA: m after preprocessing	DCA: # of subproblems	DCA: CPU time	CPU time (Beasley [2])†	# of open plants
cap71	16	50	⋆	0	<0.01	0.11	11
cap72	16	50	⋆	0	<0.01	0.08	9
cap73	16	50	⋆	0	<0.01	0.11	5
cap74	16	50	⋆	0	<0.01	0.05	4
cap101	25	50	9	6	<0.01	0.18	15
cap102	25	50	13	16	<0.01	0.16	11
cap103	25	50	14	16	<0.01	0.14	8
cap104	25	50	12	7	0.01	0.11	4
cap131	50	50	34	196	0.01	0.31	15
cap132	50	50	27	183	0.02	0.28	11
cap133	50	50	25	71	<0.01	0.29	8
cap134	50	50	19	25	<0.01	0.15	4

⋆ instance solved by preprocessing only.
† Cray-X-MP/28 seconds.

6.4.5 Körkel-type Instances with 65 Sites

Körkel [25] described several relatively large Euclidean SPLP instances ($m = n = 100$, and $m = n = 400$) and used a branch and bound algorithm to solve these problems. The bound used in that work is an improvement on a bound based on the dual of the linear programming relaxation of the SPLP due to Erlenkotter [7] and is extremely effective. In this subsection, we use instances that have the same cost structure as the ones in Körkel [25] but for which $m = n = 65$. Instances of this size were not dealt with in Körkel [25]. We implemented the Khachaturov-Minoux combinatorial bound both for the reduction procedure RP and the DCA-SPLP.

In Körkel [25], 120 instances of each problem size are described. These can be divided into 28 sets (the first 18 sets contain 5 instances each, and the rest contain 3 instances each). We solved all the 120 instances we generated, and found out that the instances in Sets 1, 2, 3, 4, 10, 11, and 12 are more difficult to solve than others. We therefore used these instances in the experiments in this section. The transportation cost matrix for a Körkel instance of size $n \times n$ is generated by distributing n points in random within a rectangular area of size 700×1300 and calculating the Euclidean distances between them. The fixed cost are computed as in Table 8.

Tab. 8: Description of the fixed costs for instances in Körkel (1989)

Problem Set	# of instances	Fixed cost for i^{th} instance
Set 1	5	Identical, set at $141 + 6.6i$
Set 2	5	Identical, set at $174 + 6.6i$
Set 3	5	Identical, set at $207 + 6.6i$
Set 4	5	Identical, set at $174 + 66i$
Set10	5	Identical, set at $7170 + 660i$
Set11	5	Identical, set at $7120.5 + 333.3i$
Set12	5	Identical, set at $8787 + 333.3i$

The values of the results that we present for each set is the average of the values obtained for all the instances in that set. Interestingly, the pre-processing rules were found to be totally ineffective for all of these problems. Since the fixed costs are identical for all the sites, the sites are distributed randomly over a region, and the variable cost matrix is symmetric, no site presents a distinct advantage over any other. This prevents our reduction procedure to open or close any site. Table 9 shows the variation in the costs of the solution output by the DCA-SPLP with changes in α, and Table 10 shows the corresponding decrease in execution times.

Tab. 9: Costs of solutions output by the DCA-SPLP on Körkel-type instances with 65 sites

Problem	Optimal	Acceptable accuracy*				
Set		1%	2%	3%	5%	10%
Set 1	6370.0	6404.8	6450.6	6480.6	6569.2	6781.0
Set 2	6920.6	6952.2	6971.4	7028.4	7123.8	7320.2
Set 3	7707.4	7738.0	7770.2	7797.6	7854.6	8053.8
Set 4	9601.2	9642.4	9680.2	9698.4	9786.6	9932.0
Set10	146691.2	146896.6	146909.6	147543.6	148062.0	151542.2
Set11	168598.4	168858.2	169655.0	170341.6	170597.0	173913.8
Set12	186386.3	186729.7	187112.0	188002.7	188854.2	192528.7

⋆ As a percentage of the optimal cost.

The effect of varying the acceptable accuracy α on the cost of the solutions output by the DCA-SPLP is also presented graphically in Figure 11. We define the *achieved accuracy* β as

$$\beta = \frac{\text{cost of the DCA-SPLP output} - \text{cost of an optimal solution}}{\text{cost of optimal solution}}$$

Tab. 10: Execution times for the DCA-SPLP on Körkel-type instances with 65 sites

Problem Set	Optimal	Acceptable accuracy* 1%	2%	3%	5%	10%
Set 1	119.078	90.948	70.758	55.494	43.200	20.426
Set 2	290.388	225.108	172.422	145.828	96.240	36.966
Set 3	458.370	339.420	259.022	203.036	150.216	50.378
Set 4	158.386	129.694	109.754	89.666	65.548	30.058
Set10	428.598	370.120	319.804	283.832	230.078	142.090
Set11	542.530	476.350	418.628	408.594	290.338	160.744
Set12	479.092	416.472	370.832	326.572	261.835	149.038

⋆ As a percentage of the optimal cost.

and the *relative time* τ as

$$\tau = \frac{\text{execution time for the DCA-SPLP for acceptable accuracy } \alpha}{\text{execution time for the DCA-SPLP to compute an optimal solution}}$$

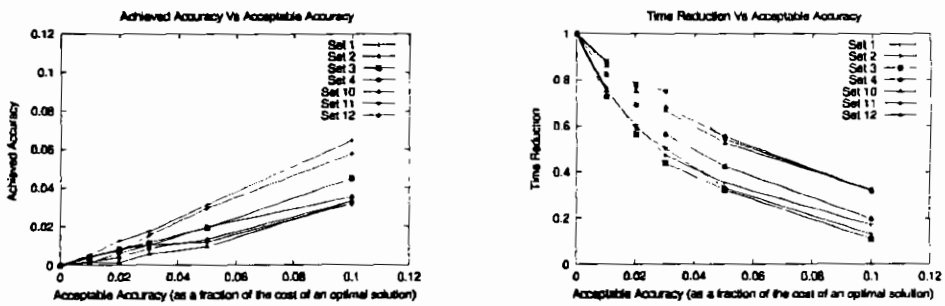


Fig. 11: Performance of the DCA-SPLP for Körkel-type instances with 65 sites

Note that the achieved accuracy β varies almost linearly with α, with a slope close to 0.5. Also note that the relative time τ of the DCA-SPLP reduces with increasing α. The reduction is slightly better than linear, with an average slope of -8.

6.4.6 Körkel-type Instances with 100 Sites

We solved the benchmark instances in Körkel [25] with $m = n = 100$ to optimality and observed that the instances in Sets 10, 11, and 12 required relatively longer execution times. So we restricted further computations to

instances in those sets. The fixed and transportation costs for these problems are computed in the procedure described in Subsection 6.4.5. Tables 11 and 12 show the results obtained by running the DCA-SPLP on these problem instances. In our DCA-SPLP implementation for solving these instances, we used the Erlenkotter bound in both the reduction procedure RP and the DCA-SPLP.

Tab. 11: Costs of solutions output by the DCA-SPLP on Körkel-type instances with 100 sites

Problem	Optimal	Acceptable accuracy*				
Set		1%	2%	3%	5%	10%
Set10	190782.0	191550.8	192755.4	192080.6	195983.2	203934.2
Set11	219583.4	220438.8	222393.6	221947.2	228467.2	235963.4
Set12	240402.4	241609.6	243336.8	244209.4	247417.6	259168.6

⋆ As a percentage of the optimal cost.

Tab. 12: Execution times for the DCA-SPLP on Körkel-type instances with 100 sites

Problem	Optimal	Acceptable accuracy*				
Set		1%	2%	3%	5%	10%
Set10	133.746	91.774	65.99	65.908	44.2	32.074
Set11	81.564	55.356	39.554	38.348	33.628	17.598
Set12	111.272	85.858	65.608	55.928	61.758	33.014

⋆ As a percentage of the optimal cost.

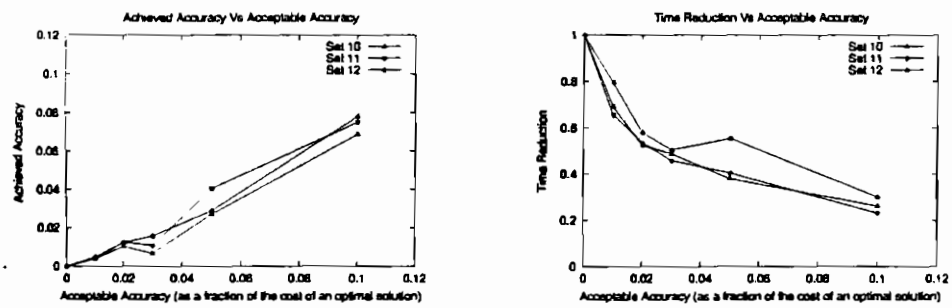


Fig. 12: Performance of the DCA-SPLP for Körkel-type instances with 100 sites

Figure 12 illustrates the effect of varying the acceptable accuracy α on the cost of the solutions output by the DCA-SPLP for the instances mentioned above. The nature of the graphs is similar to those in Figure 11. However, in several of the instances we noticed that β reduced when α is increased, and in some other instances τ increased when α was increased.

References

[1] E. Balas and P. Toth, Branch and Bound Methods, Chapter 10 in Lawler *et al.* [26].

[2] J.E. Beasley, Lagrangian Heuristics for Location Problems, *European Journal of Operational Research* Vol.65 (1993) pp. 383-399.

[3] J.E. Beasley, OR-Library, `http://mscmga.ms.ic.ac.uk/info.html`

[4] V.L. Beresnev, On a Problem of Mathematical Standardization Theory, *Upravliajemyje Sistemy* Vol.11 (1973) pp. 43-54 (in Russian).

[5] O. Bilde and J. Krarup, Sharp Lower Bounds and Efficient Algorithms for the Simple Plant Location Problem, *Annals of Discrete Mathematics* Vol.1 (1977) pp. 79-97.

[6] G. Cornuejols, G.L. Nemhauser, and L.A. Wolsey, The Uncapacitated Facility Location Problem, in P.B. Mirchandani and R.L. Francis (eds.) *Discrete Location Theory*, (New York:Wiley-Interscience, 1990) pp. 119-171.

[7] D. Erlenkotter, A Dual-Based Procedure for Uncapacitated Facility Location, *Operations Research* Vol.26 (1978) pp. 992-1009.

[8] R.D. Galvão and L.A. Raggi, A Method for Solving to Optimality Uncapacitated Location Problems, *Annals of Operations Research* Vol.18 (1989) pp.225-244.

[9] B. Goldengorin, Methods of Solving Multidimensional Unification Problems, *Upravljaemye Sistemy* Vol.16 (1977) pp. 63-72.

[10] B. Goldengorin, A Correcting Algorithm for Solving Some Discrete Optimization Problems, *Soviet Mathametical Doklady* Vol.27 (1983) pp. 620-623.

[11] B. Goldengorin, A Correcting Algorithm for Solving Allocation Type Problems, *Automated Remote Control* Vol.45 (1984) pp. 590-598.

[12] B. Goldengorin, Correcting Algorithms for Solving Multivariate Unification Problems, *Soviet Journal of Computer Systems Science* Vol.1 (1985) pp. 99-103.

[13] B. Goldengorin, On the Exact Solution of Problems of Unification by Correcting Algorithms, *Doklady Akademii, Nauk, SSSR* Vol.294 (1987) pp. 803-807.

[14] B. Goldengorin, G. Sierksma, G.A. Tijssen, and M. Tso, The Data-Correcting Algorithm for Minimization of Supermodular Functions. *Management Science* Vol.45 (1999) pp. 1539-1551.

[15] B. Goldengorin, D. Ghosh, and G. Sierksma, *Equivalent Instances of the Simple Plant Location Problem*, (SOM Research Report-00A54, University of Groningen, The Netherlands 2000).

[16] B. Goldengorin, G.A. Tijssen, D. Ghosh, G. Sierksma, Solving the Simple Plant Location Problem Using a Data Correcting Approach, *Journal of Global Optimization.* Vol.25 (2003) pp. 377-406.

[17] B. Goldengorin, *Data Correcting Algorithms in Combinatorial Optimization*, (Ph.D. Thesis, SOM Research Institute, University of Groningen, Groningen, The Netherlands, 2002).

[18] B. Goldengorin and D. Ghosh, A Multilevel Search Algorithm for the Maximization of Submodular Functions, (to appear in *Journal of Global Optimization*).

[19] P.C. Gilmore, E.L. Lawler, and D.B. Shmoys, Well-Solved Special Cases, Chapter 4 in Lawler *et al.* [26].

[20] G. Gutin and A.P. Punnen (eds.) *The Traveling Salesman Problem and its Variations*, (Kluwer Academic Publishers, The Netherlands, 2002).

[21] P.L. Hammer, Plant Location — A Pseudo-Boolean Approach. *Israel Journal of Technology* Vol.6 (1968) pp. 330-332.

[22] V.R. Khachaturov, *Some Problems of the Consecutive Calculation Method and its Applications to Location Problems*, (Ph.D. Thesis, Central Economics and Mathematics Institute, Russian Academy of Sciences, Moscow, 1968), (in Russian).

[23] V.R. Khachaturov, *Mathematical Methods of Regional Programming*, (Moscow, Nauka, 1989), (in Russian).

[24] B.M. Khumawala, An Efficient Branch and Bound Algorithm for the Warehouse Location Problem, *Management Science* Vol.18 (1975) pp. B718-B731.

[25] M. Körkel, On the Exact Solution of Large-Scale Simple Plant Location Problems. *European Journal of Operational Research* Vol.39 (1989) pp. 157-173.

[26] E.L. Lawler, J.K. Lenstra, A.H.G. Rinooy Kan, and D.B. Shmoys, (eds.) *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*, (Wiley-Interscience, 1985).

[27] H. Lee, G.L. Nemhauser, and Y. Wang, Maximizing a Submodular Function by Integer Programming: Polyhedral Results for the Quadratic Case, *European Journal of Operational Research* Vol.94 (1996) pp. 154-166.

[28] L. Lovasz, Submodular Functions and Convexity, in A. Bachem, M. Grötschel, B. Korte (eds.) *Mathematical Programming: The State of the Art*, (Springer-Verlag, Berlin, 1983) pp. 235-257.

[29] M. Minoux, Accelerated Greedy Algorithms for Maximizing Submodular Set Functions, in J. Stoer (ed.) *Actes Congres IFIP*, (Springer, Berlin, 1977) pp. 234-243.

[30] G. Reinelt, TSPLIB 95, `http://www.iwr.uni-heidelberg.de/iwr/comopt/soft/TSPLIB95/TSPLIB.html`, 1995.