

---

## **Polynomial unconstrained binary optimisation – part 2**

---

**Fred Glover\***

1OptTek Systems, Inc.,  
2241 17th Street, Boulder, CO 80302, USA  
E-mail: [glover@opttek.com](mailto:glover@opttek.com)  
\*Corresponding author

**Jin-Kao Hao**

Laboratoire d'Etude et de Recherche en Informatique (LERIA),  
Université d'Angers,  
2 Boulevard Lavoisier, 49045 Angers Cedex 01, France  
E-mail: [jin-kao.hao@univ-angers.fr](mailto:jin-kao.hao@univ-angers.fr)

**Gary Kochenberger**

School of Business Administration,  
University of Colorado at Denver,  
Denver, CO 80217, USA  
E-mail: [gary.kochenberger@cudenver.edu](mailto:gary.kochenberger@cudenver.edu)

**Abstract:** The class of problems known as quadratic zero-one (binary) unconstrained optimisation has provided access to a vast array of combinatorial optimisation problems, allowing them to be expressed within the setting of a single unifying model. A gap exists, however, in addressing polynomial problems of degree greater than 2. To bridge this gap, we provide methods for efficiently executing core search processes for the general polynomial unconstrained binary (PUB) optimisation problem. A variety of search algorithms for quadratic optimisation can take advantage of our methods to be transformed directly into algorithms for problems where the objective functions involve arbitrary polynomials.

Part 1 of this paper (Glover et al., 2010) provided fundamental results for carrying out the transformations and described coding and decoding procedures relevant for efficiently handling sparse problems, where many coefficients are 0, as typically arise in practical applications. In the present part 2 paper, we provide special algorithms and data structures for taking advantage of the basic results of part 1. We also disclose how our designs can be used to enhance existing quadratic optimisation algorithms.

**Keywords:** zero-one optimisation; unconstrained polynomial optimisation; metaheuristics; computational efficiency.

**Reference** to this paper should be made as follows: Glover, F., Hao, J-K. and Kochenberger, G. (xxxx) 'Polynomial unconstrained binary optimisation – part 2', *Int. J. Metaheuristics*, Vol. X, No. Y, pp.000–000.

**Biographical notes:** Fred Glover holds the title of Distinguished Professor at the University of Colorado and is Chief Technology Officer for OptTek Systems, Inc. He has authored or co-authored more than 400 published articles and eight books in the fields of mathematical optimisation, computer science and artificial intelligence. He is the recipient of the distinguished von Neumann Theory Prize, an elected member of the National Academy of Engineering, and has received honorary awards and fellowships from the American Association for the Advancement of Science (AAAS), the NATO Division of Scientific Affairs, the Miller Institute of Basic Research in Science and numerous other organisations.

Jin-Kao Hao holds his Full Professor position in the Computer Science Department of the University of Angers (France) and is currently the Director of the LERIA Laboratory. His research lies in the design of effective heuristic and metaheuristic algorithms for solving large-scale combinatorial search problems. He is interested in various application areas including bioinformatics, telecommunication networks and transportation. He has co-authored more than 120 peer-reviewed publications in international journals, book chapters and conference proceedings.

Gary Kochenberger is a Full Professor of Decision Science at the University of Colorado at Denver, where he is the co-Director of the Decision Science programme. His research focuses on designing and testing metaheuristics methods for large scale optimisation problems. He has co-authored more than 70 refereed papers and three books.

## 1 Introduction

### 1.1 Problem representation

The polynomial unconstrained binary (PUB) optimisation problem may be formulated as:

$$\text{PUB: Minimise } x_0 = c_0 + \sum (c_p F_p : p \in P)$$

$x$  binary

where  $x = (x_1, x_2, \dots, x_n)$  consists of binary variables,  $x_i \in \{0, 1\}$  for  $i \in N = \{1, \dots, n\}$ , the coefficients  $c_p$  for  $p \in P = \{1, \dots, p_0\}$  are non-zero scalars, and  $F_p$  is a product of components of the  $x$  vector given by:

$$F_p = \prod (x_i : i \in N_p) \text{ where } N_p \subset N.$$

Each variable  $x_i$  in the product defining  $F_p$  appears only once, noting that  $x_i^h = x_i$  for  $x_i$  binary, which renders powers  $h$  of  $x_i$  other than  $h = 1$  irrelevant.

*Remark 1* In a polynomial representation based on permutations, where two permutations  $N_p^o = (i_1, i_2, \dots, i_h)$  and  $N_q^o = (j_1, j_2, \dots, j_h)$ , are over the same set of indexes, and the associated costs  $c_p^o$  and  $c_q^o$  are both non-zero, an

equivalent problem results by redefining  $c_p^o = c_p^o + c_q^o$  and  $c_q^o = 0$ , thus eliminating the term for the vector  $N_q^o$ .

The remark is justified simply by noting that the product  $x_{i_1}x_{i_2} \dots x_{i_h}$  has the same value as the product  $x_{j_1}x_{j_2} \dots x_{j_h}$ . By summing the costs for different permutations as indicated, the remark gives a basis for a pre-processing step enabling any permutation-based formulation of PUB to be converted into the form shown. Such a pre-processing step is equivalent to producing a restricted permutation formulation where each permutation  $N_p^o = (i_1, i_2, \dots, i_h)$  has the *ascending index property*  $i_1 < \dots < i_h$ . The next remark is useful to facilitate certain operations of our method.

*Remark 2* We assume  $P_j = \{j\}$  for  $j \in N$  without regard to the value of  $c_j$ , thus providing an exception to the rule of only including terms with non-zero coefficients in the PUB representation.

### *Illustration of the PUB representation*

Consider the PUB problem whose objective function is given by:

$$x_o = 7 + 5x_1 + 2x_2 - 3x_2^2 - 4x_3x_1 + 5x_1x_2 - 2x_2x_1 + 3x_1^3x_2x_3.$$

First, since  $x_2^2 = x_2$ ,  $x_1^3 = x_1$  and  $x_1x_2 = x_2x_1$ , we can re-write  $x_o$  in ascending index notation, including only the non-zero coefficients, as:

$$x_o = 7 + 5x_1 - x_2 - 4x_1x_3 + 3x_1x_2 + 3x_1x_2x_3.$$

By Remark 2, we include the  $x_3$  term in the representation, even though it has a 0 coefficient, to give:

$$x_o = 7 + 5x_1 - x_2 + 0x_3 - 4x_1x_3 + 3x_1x_2 + 3x_1x_2x_3.$$

Assigning indexes  $p = 1$  to 6 to the terms in sequence, we identify the indexes of the variables in these terms by:

$$N_1 = \{1\}, N_2 = \{2\}, N_3 = \{3\}, N_4 = \{1, 3\}, N_5 = \{1, 2\}, N_6 = \{1, 2, 3\}$$

The cost coefficients associated with these terms, including the constant term  $c_o$ , are given by:

$$c_o = 7, c_1 = 5, c_2 = -1, c_3 = 0, c_4 = -4, c_5 = 3, c_6 = 3.$$

## *1.2 Applications and motivation*

The special case where the polynomial objective of PUB is a quadratic function (producing a polynomial of degree 2) has been widely studied. As noted in Part 1 of this paper (Glover et al., 2010), the quadratic case already encompasses a broad range of important problems, including those from social psychology, financial analysis, computer aided design, traffic management, machine scheduling, cellular radio channel allocation, and molecular conformation among others. Moreover, many combinatorial optimisation problems pertaining to graphs such as determining maximum cliques, maximum cuts,

maximum vertex packing, minimum coverings, maximum independent sets, and maximum independent weighted sets are known to be capable of being formulated by PUB in the quadratic case as documented in papers of Pardalos and Rodgers (1990), and Pardalos and Xue (1994). A review of additional applications and formulations can be found in Kochenberger et al. (2004).

The more general PUB formulation given here is of interest for its ability to encompass a significantly expanded range of problems. The cubic case, for example, permits PUB to represent the important class of satisfiability problems known as 3-SAT, and offers the advantage of providing a representation whose size does not depend on the number of logical clauses, which stands in contrast to the case for customary binary integer programming formulations of 3-SAT (see e.g., Kochenberger, 2010). The availability of procedures for efficiently handling and updating move evaluations for instances of PUB involving polynomials of degree greater than 2, as identified in the present work, provides an impetus to uncover additional problems that can be usefully given binary polynomial formulations.

Our procedures, which apply to moves that flip (complement) the values of one or more variables  $x_j$  in progressing from one solution to another, constitute a generalisation of procedures for generating 1-flip moves described in Glover et al. (1998) and extended to 2-flip and multi-flip moves Glover and Hao (2010a, 2010b). Important recent contributions of a similar nature for the quadratic problem are provided in Hanafi et al. (2010). A principle outcome of our development is that a variety of search methods which currently incorporate procedures to evaluate flip moves for the quadratic problem can replace these procedures by the methods described here, thereby producing methods capable of solving general PUB problems without any other changes in their structure.

## 2 Preliminary relationships

We briefly summarise basic observations from Part 1 of this paper without proof.

Let  $x'$  and  $x''$  represent two binary solutions and define:

$$x'_o = \sum (c_p F'_p : p \in P), \text{ where } F'_p = \Pi(x'_k : k \in N_p)$$

$$x''_o = \sum (c_p F''_p : p \in P), \text{ where } F''_p = \Pi(x''_k : k \in N_p).$$

$$\Delta x_o = x''_o - x'_o$$

The objective function change  $\Delta x_o$  thus discloses whether the transition (move) from  $x'$  to  $x''$  will cause  $x_o$  to improve or deteriorate (respectively, decrease or increase) relative to the minimisation objective.

We identify the variables  $x_i$  that are complemented in going from the solution  $x'$  to the solution  $x''$ , and the subsets for which  $x_i'' = 1$  and 0 by defining:

$$N^c = \{i \in N : x_i'' = 1 - x_i'\}$$

$$N^c(1) = \{i \in N^c : x_i'' = 1\}$$

$$N^c(0) = \{i \in N^c : x_i'' = 0\}$$

$$P(M) = \{p \in P : N_p \subset M\},$$

where  $M$  is an arbitrary subset of  $N$ .

*Observation 1:* If  $x_i' x_i'' = 0$  ( $x_i' = 0$  or  $x_i'' = 0$ ) for each  $i \in N$ , then:

$$\Delta x_0 = \sum (c_p : p \in P(N^c(1))) - \sum (c_p : p \in P(N^c(0)))$$

We augment this result by two corollaries employing the more restrictive assumption that  $x' = 0$ . Let  $N'' = \{i \in N : x_i'' = 1\}$  (hence  $P(N'') = \{p \in P : N_p \subset N''\}$ ) to identify those sets  $N_p$  such that  $x_i'' = 1$  for all  $i \in N_p$ . Thus, if  $N'' = \{i_1, \dots, i_h\}$ , then  $P(N'')$  is the index set for all those sets  $N_p$  composed of one or more of the elements  $i_1, \dots, i_h$ .

*Corollary 1.1:* If  $x' = 0$ , then:

$$\Delta x_0 = \sum (c_p : p \in P(N''))$$

To simplify the statement of the next result, it is useful to isolate the coefficients  $c_p$  of the product terms  $F_p$  that refer only to a single variable  $x_j$ . As previously noted, we suppose  $N_p = \{p\}$  for  $p \in N$ , although  $c_p = 0$  is possible for some of these indexes. Hence, the product term  $F_p$  for  $p \in N$  is the single variable term  $F_p = x_p$ . If we denote indexes  $p \in N$  instead by  $j \in N$  to conform to the practise of referring to variables  $x_j$  for  $j \in N$ , our indexing convention identifies the ' $x_j$  component' of the objective function  $x_0$  to be just  $c_j x_j$ .

*Corollary 1.2:* If  $x' = 0$ , and  $x''$  results from  $x'$  by flipping the single variable  $x_j$ , then:

$$\Delta x_0 = c_j.$$

Corollaries 1.1 and 1.2 identify the change in  $x_0$  that results from flipping a single variable  $x_j$  to be  $c_j$  while the change that results from flipping all variables  $x_j$  for  $j \in N''$  is given by  $\sum (c_p : p \in P(N''))$ . Hence, Corollary 1.2 gives an evaluation of a 1-flip move and Corollary 1.1 gives an evaluation of a  $q$ -flip move by letting the set  $N''$  represent the indexes for a selected set of  $q$  variables that are flipped from 0 to 1.

### 3 Exploiting and updating problem transformations

The preceding corollaries have an important implication: if we start from a solution  $x' = 0$ , the amount of effort to evaluate a move involving any number of flips, from 1 to  $q$ , is the same for any polynomial of degree  $d$  (defined by  $d = \max(|N_p| : p \in P)$  for which  $d \geq q$ ). Thus, by this stipulation, the work to evaluate a 1-flip is the same for all polynomials, the work to evaluate a 2-flip is the same for all polynomials of degree 2 or greater, and so on. It also follows that the work to evaluate a  $q$ -flip for a polynomial of degree  $d \geq q$  only requires a single addition operation beyond the work to evaluate a  $q$ -flip for a polynomial of degree  $d = q - 1$ . Consequently, a 3-flip in a polynomial of degree 3 or larger can be evaluated by using only one addition operation beyond that required to evaluate a 3-flip in a polynomial of degree 2.

We manipulate the formulation of PUB so that a current solution  $x'$  may always be treated as if it were the 0 solution, using the common device of transforming the  $x$  vector into another binary vector  $y$  by complementing selected components of  $x$ ; in this case, specifically complementing those components of  $x$  such that  $x'_j = 1$ , thus causing the assignment  $x = x'$  to yield a corresponding assignment  $y = y'$  for which  $y' = 0$ .

To broaden the generality of our results, we introduce a special set  $N_o$  and a corresponding ‘product term’  $F_o$  associated with the objective function variable  $x_o$ , where we stipulate that  $N_o = \emptyset$  and hence  $F_o = 1$  (applying the definition  $F_p = \Pi(x_k: k \in N_p)$  to the case where  $N_p = N_o = \emptyset$ . This yields  $c_o F_o = c_o$ , and hence  $c_o F_o$  is just the constant term associated with the objective function  $x_o$ . These conventions allow us to express changes in  $x_o$  using the same notation employed to express changes in general terms of the form  $c_p F_p$ .

For the 1-flip case we denote the variable that is flipped by  $x_j$ , hence yielding  $y_j = 1 - x_j$ . Then we define the following for each  $j \in N$ :

$$P(j) \quad \{p \in P: j \in N_p\}.$$

$$p[j] \quad \text{the unique index such that } N_{p[j]} = N_p - \{j\}. \text{ (Hence, } F_{p[j]} = \Pi(x_k: k \in N_p - \{j\}).$$

$$c_{p[j]} \quad 0 \text{ if } p[j] \notin P.$$

$$F_{p[j]} \quad y_j F_{p[j]} \text{ (Hence, } F_{p[j]} \text{ is the same as } F_p \text{ except that } y_j \text{ replaces } x_j).$$

We observe that  $P(j)$ , which identifies the index set for all sets  $N_p$  that contain  $j$ , is effectively a special case of  $P(M) = \{p \in P: N_p \subset M\}$ , by taking  $M = \{j\}$ .

*Observation 2:* Flipping  $x_j$  to replace  $x_j$  by  $y_j = 1 - x_j$  produces the following changes for each index  $p \in P(j)$ :

$$c_{p[j]} := c_{p[j]} + c_p \left( \text{changing } c_{p[j]} F_{p[j]} \text{ to become } (c_{p[j]} + c_p) F_{p[j]} \right).$$

$$c_p := -c_p$$

$$F_p := F_{p[j]} \left( \text{changing } c_p F_p \text{ to become } c_p F_{p[j]} \text{ for the new } c_p \text{ value} \right).$$

Moreover, these changes are independent, so that the change for one index  $p \in P(j)$  does not affect the change for another  $p \in P(j)$ .

It is important to observe that some or all of the indexes  $p[j]$  may not belong to  $P$  (as occurs when  $c_{p[j]} = 0$ ). If  $p[j] \notin P$ , then except for the special case where  $N_p$  contains the index of a single variable,  $p \in P$  implies  $c_p \neq 0$ , and hence the new coefficient  $c_{p[j]} := c_{p[j]} + c_p$  must be non-zero. This compels  $P$  to be enlarged for the representation PUB( $y$ ) by setting  $P := P \cup \{p[j]\}$ . On the other hand, if  $p[j] \in P$ , then it is possible that the new value  $c_{p[j]} + c_p$  of  $c_{p[j]}$  may become 0, and in this case  $P$  must be reduced by setting  $P := P - \{p[j]\}$ . Again, we later give processes for handling such operations efficiently.

The update produced by Observation 2 is completed by redefining  $x$  to be  $y$  (hence redefining  $x_j = 1 - x_j$ ) so that we may treat the current solution as  $x' = 0$ , and apply Observation 2 again to repeat the process. A record of the true solution  $x$  is kept throughout these operations, but the algorithm does not have to know this solution, and by means of the currently updated formulation only ‘sees’ a current collection of terms  $c_p F_p$  and their associated sets  $N_p$  for  $p \in P$  (referring to the current  $P$ ).

We now turn to the question of how to perform these processes efficiently, giving particular emphasis to the situation in which the polynomial is sparse; i.e., where  $P$  represents only a small subset of all possible index sets (equivalently, when many terms  $F_p$  have associated costs  $c_p = 0$  and hence do not explicitly appear in the polynomial).

To provide continuity in our present development, the next two sections likewise make reference to information contained in Part 1, though in abbreviated form.

#### 4 Overall structure of the algorithm

Drawing on Observation 2, a generic method for the PUB problem may be summarised as follows.

##### *Generic PUB method*

0. Create the initial PUB data structures and select a starting solution. Transform the problem representation so that this solution becomes the current 0 solution.

---

While a chosen termination condition is not satisfied

- 1 Select one or more variables  $x_j$  to be flipped, by employing the  $c_j$  values to identify  $\Delta x_0$  for 1-flip moves as in Corollary 1.2, or the  $c_p$  values to identify  $\Delta x_0$  for multi-flip moves as in Corollary 1.2.
- 2 Apply Observation 2 to update the current problem representation. Maintain the appropriate composition of  $P$ , and the sets  $N_p$  and  $P(i)$ , for each 1-flip replacing  $x_j$  by  $1 - x_j$ , executed as follows for  $p^* = p[j]$  where  $p^* > n$ :
  - If  $c_{p^*} = 0$  then
    - Create the set  $N_{p^*}$  and add  $p^*$  to the set  $P(i)$  for each  $i \in N_{p^*}$ .
  - ElseIf  $c_{p^*} + c_p = 0$  ( $c_{p^*} \neq 0$ ) then
    - Remove reference to the set  $N_{p^*}$  and drop  $p^*$  from  $P$  by removing  $p^*$  from the set  $P(i)$  for each  $i \in N_{p^*}$ .

EndWhile

---

The use of the  $c_j$  and  $c_p$  values as criteria for selecting a move in step 1 above may of course be accompanied by additional criteria, such as employing tabu restrictions and aspiration incentives derived from recency and frequency memory in a tabu search method.

As intimated by the algorithm's description, the operations of maintaining the problem representation reduce to the operations of updating the sets  $P(i)$ , together with maintaining a representation of the sets  $N_p$  themselves. Explicit knowledge of the set  $P$  is unnecessary, since all relevant knowledge about  $P$  is contained in the sets  $P(i)$ .

The next section gives a way to identify the elements of the sets  $N_p$  without explicitly listing them, while maintaining a list of  $c_p$  values that is dramatically smaller than would be created by allocating a multidimensional matrix to this task (by creating a cost matrix  $C(i_1, i_2, \dots, i_d)$ ). To accomplish this requires a means to code vectors of the form  $(i_1, \dots, i_h)$  as single numbers  $v$ , and to decode such numbers  $v$  back into the vectors  $(i_1, \dots, i_h)$  from which they were derived.

## 5 Coding and decoding index vectors for product terms

We reiterate the convention that the indexes of product terms are written in the form a vector of indexes  $(i_1, i_2, \dots, i_h)$  where  $i_1 < i_2 < \dots < i_h$  (hence corresponding to the product term  $\Pi(x_k: k = i_r: r = 1, \dots, h)$  where  $h$  takes a value between 1 and the degree  $d$  of the polynomial). Note that we use the symbol  $i$  in this representation because each element  $i_r$  identifies an index rather than a variable such as  $x_k$ . We first discuss the procedure for coding each such vector of indexes as a single value  $V$ .

### 5.1 Coding procedure

#### Organisation

Partition the terms into groups,  $G(1)$  to  $G(d)$ , where each group is a collection of vectors  $(i_1, \dots, i_h)$  for  $h = 1, \dots, d$ , identifying the indexes of all possible terms containing  $h$  elements. (The elements of these vectors constitute the ordered form of the sets  $N_p$  that may potentially be created, in a case where the problem is fully dense.):

$$\begin{aligned} G(1): (i_1) : i_1 = 1, \dots, n \\ G(2): (i_1, i_2) : i_1 = 1, \dots, n-1; i_2 = i_1 + 1, \dots, n \\ \dots \dots \dots \\ G(d): (i_1, \dots, i_d) : i_1 = 1, \dots, n-(d-1), i_2 = i_1 + 1, \dots, n-d; \dots; \\ i_{d-1} = i_{d-2} + 1, \dots, n-1; i_d = i_{d-1} + 1, \dots, n \end{aligned}$$

*Step 1* Create a base cardinality  $\Delta(h)$  and a cumulative cardinality  $n(h)$  for each group  $G(h)$ ,  $h = 1, \dots, d$ . (The cumulative cardinality is the number of vectors in group  $G_h$  added to the number of vectors in all preceding groups).

$$\begin{aligned} G(1): \Delta(1) = n; \quad n(1) = \Delta(1) \\ G(h): \Delta(h) = n(n-1)\dots(n-(h-1)) / h! \quad n(h) = n(h-1) + \Delta(h) \text{ for } h = 2, \dots, d \end{aligned}$$

*Step 2* Create the base coding  $v(h)$  for an arbitrary vector  $(i_1, \dots, i_h)$ , for each group  $G(h)$ :

$$\begin{aligned} G(1): v(1) = i_1 \\ G(h): v(h) = \Pi(i_h - q) : q = 1, \dots, h / h! + v(h-1) \text{ for } h = 2, \dots, d. \end{aligned}$$

*Step 3* Create the full coding  $V(h)$ , by adding the cumulative cardinality  $n(h-1)$  to  $v(h)$  for an arbitrary vector  $(i_1, \dots, i_h)$ , for each group  $G(h)$ :

$$\begin{aligned} G(1): \text{For } (i_1) : V(1) = v(1) (= i_1) \\ G(h): \text{For } (i_1, \dots, i_h) : V(h) = n(h-1) + v(h) \text{ for } h = 2, \dots, d \end{aligned}$$

The cumulative cardinality  $n(d)$  is the maximum value of  $|P|$  for a polynomial of degree  $d$ , hence the maximum number of non-zero coefficients  $c_p$  when the polynomial is

represented in ascending index order. The coding operation assigns a unique index  $p = V[M]$  to each set  $M = \{i_1, \dots, i_h\}$  for  $h = 1, \dots, d$ , where  $i_1 < \dots < i_h$ . We use the notation  $V[M]$  to distinguish the value produced by coding  $M$  from the value  $V(h)$  that represents the coding value previously defined. Thus, in particular,  $V[M] = V(h)$  for the value  $V(h)$  calculated by reference to  $M = \{i_1, \dots, i_h\}$  in step 3 above. ( $M$  may strictly speaking be considered a vector, though we continue to refer to it as a set for convenience). The indexes  $p = V[M]$  are exactly those from the set  $\{1, 2, \dots, n(d)\}$ .

### 5.2 Using the coding for inputting initial problem data

The coding procedure of Section 5.1 is implemented immediately upon inputting the initial problem data, thereby providing a compact representation to take advantage of situations where the data is sparse. Accomplishing this within the data input procedure is quite simple:

#### Input procedure

- 
- 1 Initialise for  $p = 1$  to  $n(d)$
  - 2 Read problem data and simultaneously generate the coding: let  $M$  denote the current set of indexes input from the problem data to become a set  $N_p$ , and let  $c$  denote the cost associated with  $M$  that is to become the value  $c_p$  attached to the product term  $\Pi(x_k: k \in N_p)$  for  $N_p = M$ .
  - 3 Produce the index  $p = V[M]$  by applying the Full Coding Rule to the elements of  $M$ , and let  $c_p = c$ .
- 

### 5.3 Decoding a coded value $p = V$ to obtain the index set $M$ such that $V[M] = V$

Let  $V$  denote the (full) coded value of an index set  $M = \{i_1, \dots, i_h\}$ , where  $V$  is the index  $p$  of the unknown set  $N_p = M$ . We identify each component  $i_1, \dots, i_h$  of  $M$  (hence of  $N_p$ ) by performing appropriate operations on the value  $V$ .

For an arbitrary real number  $z$ , let  $[z]$  denote the greatest integer  $\leq z$ , and let  $\langle z \rangle$  denote the least integer  $\geq z$  (hence when  $z$  is a positive non-integer value, then  $[z]$  rounds  $z$  down and  $\langle z \rangle$  rounds  $z$  up). Then for each value  $r$  from 2 to the degree  $d$  of the polynomial, we make reference to a constant  $\beta(r)$  determined by the formula:

$$\beta(r) = r + 1 - \langle (r!)^{1/r} \rangle.$$

The values  $\beta(r)$ , for  $r = 2, \dots, d$ , may be computed in advance and stored, thus avoiding the need to re-compute these values multiple times when applying the decoding algorithm. For example, if  $d = 10$ , the relevant  $\beta(r)$  values obtained by the preceding formula are as follows:  $\beta(2) = 1$ ,  $\beta(3) = 2$ ,  $\beta(4) = 2$ ,  $\beta(5) = 3$ ,  $\beta(6) = 4$ ,  $\beta(7) = 4$ ,  $\beta(8) = 5$ ,  $\beta(9) = 5$ ,  $\beta(10) = 6$ . (These values disclose that when  $r \leq 5$ ,  $\beta(r)$  can be computed from the simpler formula  $c(r) = \lceil (r + 1)/2 \rceil$ ).

*Decoding algorithm*


---

```

Step 1. /* Identify the group h associated with V. */
The group is G(1) if  $V \leq n(1)$  and is G(h) for  $h > 1$  if  $n(h-1) < V \leq n(h)$ .

Step 2. /* Convert V to a Base Code value v */
 $v = V - n(h-1)$ 

Step 3 /* Determine the vector  $(i_1, i_2, \dots, i_h)$  from the Base Code value by generating its
components  $i_r$  in reverse order, for  $r = h, h-1, \dots, 1$ . */
 $r = h$ 
while  $r > 1$ 
  If  $v = 1$  Then
     $i_r = r$ 
  Else
     $w = (v-1)r!$ 
     $u = \lfloor w^{1/r} \rfloor$ 
     $i^* = u + \beta(r)$ 
    ( $i_r = i^*$  or  $i^* + 1$ , depending on which of these gives the largest
value of  $i_r$  satisfying  $\Pi \leq w$  for  $\Pi = (i_r-1)(i_r-2)\dots(i_r-r)$ )
     $\Pi_0 = (i^*-1)(i^*-2)\dots(i^*-r+1)$ 
     $\Pi_1 = (i^*-r) \Pi_0$  (the value of  $\Pi$  if  $i_r = i^*$ )
     $\Pi_2 = i^* \Pi_0$  (the value of  $\Pi$  if  $i_r = i^* + 1$ )
    If  $\Pi_2 \leq w$  then
       $\Pi = \Pi_2$ 
       $i_r = i^* + 1$ 
    Else
       $\Pi = \Pi_1$ 
       $i_r = i^*$ 
    Endif
     $v := v - \Pi/r!$  /* giving the "residual v" to determine the next
component  $i_r$  for  $r := r-1$ . */
  Endif
   $r := r-1$ 
EndWhile
 $i_1 = v$ 

```

---

*Illustration of memory required for the cost values  $c_p$  and the associated sets  $N_p$* 

The amount of memory required to store the costs  $c_p$  and the associated sets  $N_p$  using the foregoing processes is equal to the value  $n(d)$  for a polynomial of degree  $d$  when using an ascending index order representation. To illustrate, we calculate these values for the cases  $d = 1$  to 5, assuming  $n = 100$  in each case, and compare these values to the amount of

memory required by using a cost matrix of the form  $C(i_1, \dots, i_d)$ , which entails a storage space of  $n^d$ .

Values of $d$ :	1	2	3	4	5
Cost matrix memory $n^d$ :	100	10,000	1,000,000	100,000,000	10,000,000,000
Coded memory $n(d)$ :	100	5,050	166,750	4,087,957	79,375,495

The rapid growth in the size of memory depicted by this illustration suggests that a polynomial of degree 4 or 5 may be the largest that is practical to work with. However, this memory does not account for the fact that the problem will generally be sparse, so that only a few percent of the total number of possible product terms may exist at any given time. We address the challenges of taking advantage of our results within the context of exploiting sparsity, which is a hallmark of real world problems, in the following sections.

## 6 Special memory structure and updates

### 6.1 A basic version

We first describe a basic version of the memory structures to lay a foundation for a more advanced version of these structures which gives a much faster means of accessing the  $p$  indexes than by using the coding and decoding operations.

As a starting point, we access the elements  $p$  in  $P(i)$  for  $i \in N$  by means of a location vector  $\text{Loc}(q)$ , where the indexes  $p = \text{Loc}(q)$  corresponding to  $p \in P(i)$  are in turn accessed by a doubly linked list,  $\text{Before}(q)$  and  $\text{After}(q)$ . We refer to  $\text{Loc}(q)$ ,  $\text{Before}(q)$  and  $\text{After}(q)$  collectively as the  $q$  lists. For a given set  $P(i)$ , the relevant indexes  $q$  of  $\text{Before}(q)$  and  $\text{After}(q)$  that generate the values  $p = \text{Loc}(q)$  begin at the index  $q = \text{First}(i)$ . We organise the lists so that  $\text{First}(i) = 0$  if  $P(i)$  is empty. Otherwise, the condition  $\text{After}(q) = 0$  signifies that  $q$  is the last element of the list that begins with  $q = \text{First}(i)$ . The elements  $p \in P(i)$  are then generated as follows.

---

#### Generate All $p \in P(i)$

```

q = First(i)
While q > 0
    p = Loc(q) /* identifies the "p location" of the current
                element p ∈ P(i) */
    Decode the index V[M] = p as in Section 5 to identify h and
    M = {i1, ..., ih}.
    q = After(q)
EndWhile

```

---

The list  $\text{Before}(q)$  is not required here, but is needed to perform operations of dropping elements from  $P(i)$ . We maintain the  $\text{Before}(q)$  and  $\text{After}(q)$  lists so that they satisfy the obvious relationship  $\text{Before}(q'') = q'$  if and only if  $\text{After}(q') = q''$ , with the exception that we make use of a dummy entry  $\text{Before}(0)$  whose value can be arbitrary.

A ‘pool’ array, denoted PoolQ, keeps track of available  $q$  indexes. An element  $q$  is removed from PoolQ in order to add a new index  $p = \text{Loc}(q)$  to some list  $P(i)$ , and an element  $q$  is added back to PoolQ when an index  $p = \text{Loc}(q)$  is dropped from some list  $P(i)$ . In particular, a specified element  $p^*$  as indicated in the Generic PUB Method can be added to  $P(i)$  by the following operation.

---

**Add  $p^*$  to  $P(i)$**

```

Remove an index  $q^*$  from PoolQ /* The identity of  $q^*$  is irrelevant.*/
 $\text{Loc}(q^*) = p^*$ 
 $q\text{First} = \text{First}(i)$ 
 $\text{After}(q^*) = q\text{First}$ 
 $\text{Before}(q\text{First}) = q^*$ 
 $\text{First}(i) = q^*$ 

```

---

During initialisation, when building the  $q$  lists from the input data, it suffices to set  $\text{First}(i) = 0$  for all  $i \in N$ . The element  $\text{Before}(0)$  can be assigned any value; for example,  $\text{Before}(0) = 0$ . Then the indicated operation for adding  $p^*$  to  $P(i)$  can be used to build the initial  $q$  lists for  $P(i)$  as well as to add elements to these lists on later steps.

Based on the preceding comments, the complete initialisation of the arrays is therefore achieved as follows.

---

**Initialisation step**

```

Set  $\text{First}(i) = 0$  for all  $i \in N$ .  $\text{Before}(0) = 0$ .
 $c_p = 0$  for  $p = 1$  to  $n(d)$ 
 $q = 0$ 
While problem data remain to be input
  Read in next data element pair  $(M, c)$ , where  $M = \{i_1, \dots, i_h\}$ 
  and  $c$  is the cost coefficient associated with  $M$ .
  Assume  $i_1 < \dots < i_h$  if  $h > 1$ .
  Code the set  $M = \{i_1, \dots, i_h\}$  to obtain  $p = V[M]$  and set  $c_p = c$ 
  For each  $i \in M$ 
     $q := q + 1$ 
     $q^* = \text{First}(i)$ 
     $\text{After}(q) = q^*$ 
     $\text{Before}(q^*) = q$ 
     $\text{First}(i) = q$ 
    Add  $q$  to PoolQ and set  $p = \text{Loc}(q)$ 
  End For  $i$ 
EndWhile

```

---

To drop the index  $p^*$  from a given set  $P(i)$  as indicated in the Generic PUB Method, we note that  $p^*$  refers to a set  $N_{p^*}$  that does not include the index  $j$  for the variable  $x_j$  flipped, i.e., the set  $N_{p^*}$  was not accessed by going through a linked list. Thus it is necessary for each  $i \in N_{p^*}$  to know the identity of the index  $q$  such that  $p^*$  is accessed by setting  $p^* = \text{Loc}(q)$  (where  $q$  itself is accessed from the linked list starting with  $\text{First}(i)$ ). One

way to do this is to search for this  $q$  by scanning the  $q$  list associated with  $P(i)$ . Employing such a search makes it possible to avoid referring to the linked list  $\text{Before}(q)$  but the process can be fairly expensive computationally. Consequently we employ the following approach that achieves greater efficiency.

For each set  $N_p = (i_1, \dots, i_h)$  we store a vector  $Q_p = (q_1, \dots, q_h)$  that identifies the relevant  $q$  indexes for  $N_p$ . In particular  $q_1$  is the  $q$  element on the  $q$  list for  $i_1$  that yields  $p = \text{Loc}(q_1)$ ,  $q_2$  is the  $q$  element on the  $q$  list for  $i_2$  that yields  $p = \text{Loc}(q_2)$ , and so forth. Instead of searching for the relevant  $q$  for each  $i = i_r$  in the set  $N_p = (i_1, \dots, i_h)$ , we make use of the associated value  $q^* = q_r$  and remove the index  $p$  from the set  $P(i)$  for  $i = i_r$  by the operation.

---

**Drop  $p$  from  $P(i)$**

```

If First(i) = q* then
    First(i) = After(q*)
Else
    qB = Before(q*)
    qA = After(q*)
    Before(qA) = qB
    After(qB) = qA
Endif

```

---

The preceding operation is only performed at most  $d$  times for any given set  $N_p$ , since each such set contains at most  $d$  elements. In the cubic case where  $d = 3$ , for example, the update for a given set  $N_p$  is exceedingly fast. In general, the index sets  $P(i)$  for all  $i \in N$  are maintained and updated efficiently using the indicated structure.

We now examine how our preceding basic ideas can be extended to provide advanced memory structures and updating operations to take advantage of sparsity more effectively.

## 6.2 Advanced memory structures and updates

It is advantageous to organise the memory so that different levels  $h$ , for  $h = 1$  to  $d$ , are treated separately. Thus in place of referring to index sets  $N_p$  by a single index  $p$ , we refer to these sets by means of paired indexes  $(h, p)$ , using the notation  $N_{hp} = \{i_1, \dots, i_h\}$ . The indexes  $p$  at a given level  $h$  may therefore overlap with the indexes at another level. We motivate this ‘differentiation by level’ by stating a useful consequence of Observation 2.

Define  $nz_0(h)$  to be the number of non-zero terms in the initial problem data at level  $h$ , i.e.,  $nz_0(h)$  is the number of terms over sets  $N_{hp} = \{i_1, \dots, i_h\}$  such that  $c_p \neq 0$ . Let  $z_0(h)$  denote an upper limit on the number of non-zero terms at level  $h$  that will be produced by the generic PUB algorithm throughout its entire execution. We make use of  $z_0(h)$  to identify a limit on the number of non-zero terms produced at any given point in the algorithm’s execution.

*Corollary 2.1:* A upper bound value for  $z_0(h)$  for  $h = 1$  to  $d$  is given by:

$$z_0(d) = nz_0(d)$$

$$z_0(h-1) = \text{Min}(\Delta(h-1), z_0(h)h) \text{ for } h = 2 \text{ to } d.$$

*Proof:* By Observation 2 a 1-flip can only add new terms for the sets  $N_{p[j]}$  defined by  $N_{p[j]} = N_p - \{j\}$ . Letting  $p^*$  denote  $p[j]$  as in the generic PUB algorithm, it is therefore impossible to add or drop a set  $N_{p^*}$  such that  $|N_{p^*}| = d$ , because no larger set  $N_p$  exists such that  $N_{p^*} = N_p - \{j\}$ . Consequently,  $z_o(d) = nz_o(d)$  gives a valid bound at level  $d$ . In addition, if  $z_o(h)$  is the number of terms in the union of all such terms generated throughout the progress of the algorithm, then by Observation 2 each such term can give rise to at most  $h$  other terms at level  $h - 1$  (since there are  $h$  variables in the term that can be flipped, and each variable flipped affects exactly one term  $N_{p^*}$  at level  $h - 1$ , potentially adding  $N_{p^*}$  as a new term if  $c_{p^*} = 0$ ). This establishes that  $z_o(h - 1)$  is bounded from above by  $z_o(h)h$ , noting that the limit  $\Delta(h - 1)$  as defined in Section 5 can not be exceeded.

For sparse problems, the bound on  $z_o(h - 1)$  given by  $z_o(h)h$  will generally be more limiting than the bound given by  $\Delta(h - 1)$ , except in the case for  $h = 2$ , when  $\Delta(h - 1) = \Delta(1) = n$ . A useful direct implication of the corollary is that the number of sets  $N_p = \{i_1, \dots, i_h\}$  that can receive non-zero costs  $c_p$  at any stage of the algorithm is given by the upper bound  $U(d) = \sum(z_o(h): h = 1 \text{ to } d)$ .

We next provide a way to exploit Corollary 2.1 for sparse problems by a different manner of coding the indexes  $p \in P$ , drawing on the bounds  $z_o(h)$  and storing the sets  $N_p$  explicitly.

### *Sparse problem coding – the descriptive challenge*

The stages of the advanced algorithm will be covered at a finer level of detail than normally devoted to such descriptions, in view of the fact that seemingly minor departures can produce significant repercussions affecting the accuracy as well as efficiency of the method. The goal of communicating the essential ideas poses an unconventional challenge, because the classical means of presenting pseudo code at a coarse level of detail unfortunately leaves large gaps in critical components of the algorithm, and it is essential to fill these gaps to provide a satisfactory understanding of the method. To assist in this process, we make use of supporting observations and parenthetical remarks that provide further explanation.

### *Data organisation*

The organisation employed by our method allows the data to be input in random order. We henceforth assume data entries consist of triples  $(h, M, c)$ , where  $M = \{i_1, \dots, i_h\}$  identifies a set  $N_p$  and where  $c$  denotes the cost associated with this set. In contrast to the previous assumption that the elements of  $M$  are ordered so that  $i_1 < \dots < i_h$ , we perform the ordering as data are input and simultaneously keep track of whether the same  $M$  may be referenced more than once (where the values  $i_1, \dots, i_h$  are input in different orders), in order to appropriately sum the costs for different instances of the same set as indicated in Remark 1 in Section 1.

All sets  $N_p$  for a given value of  $h$  are placed in a single group, referring to them as sets  $N_{hp}$  as the index  $p$  ranges from 1 to a value  $\text{Last}(h)$  which keeps track of the size of the group at level  $h$ . The set  $M = \{i_1, \dots, i_h\}$  is recorded as a set  $N_{hp}$  by using an array  $N(h, p, r)$ , where:

$$N(h, p, r) = i_r \text{ for } r = 1 \text{ to } h.$$

Each time a new set  $M$  is input for a given value of  $h$ , we update  $\text{Last}(h)$  by setting  $\text{Last}(h) := \text{Last}(h) + 1$ , followed by  $p = \text{Last}(h)$  and then recording  $N(h, p, r)$  as indicated.

The arrays  $N_{hp}$ ,  $p = 1$  to  $\text{Last}(h)$ , are accompanied by the following associated arrays:

$\text{Code}(h, p)$  the coded value  $v = V[N_{hp}]$

$\text{Cost}(h, p)$   $c$  (the sum of relevant  $c$  values, if more than one instance of the set  $M$  is input)

$\theta(v)$  an array to store coded indexes  $p$ , such that  $v = V[N_{hp}]$  for  $v = 1$  to  $n(d - 1)$ .  
(The fact that  $v$  only goes to  $n(d - 1)$  and not to  $n(d)$  is explained later).

### 6.3 Full organisation of the algorithm

In the following, we explicitly store all costs  $\text{Cost}(1, i)$ , referring to the variables  $x_i$ ,  $i = 1$  to  $n$ , including zero as well as non-zero costs, hence  $nz_o(1)$  is held constant at  $nz_o(1) = n$ . The method is then divided into three stages, where each of the first two stages performs preliminary set-up operations for the stage following, which consists of the main computations of the generic PUB method. This staged organisation contributes to the ability to take advantage of the coding and decoding operations in a way that saves considerable memory and yet provides substantial gains in speed during the stage devoted to the PUB algorithm.

#### Stage 1

A first pass reads the original data to determine the values  $nz_o(h)$ ,  $h = 2$  to  $d$ . Stage 1 then computes the values  $z_o(h)$  and  $U(h)$ ,  $h = 0$  to  $d$  as indicated in Corollary 2.1 and its discussion, to give  $z_o(d) = nz_o(d)$  together with  $z_o(h - 1) = \text{Min}(\Delta(h - 1), z_o(h)h)$  for  $h = 2$  to  $d$ , and  $U(d) = \sum(z_o(h): h = 1 \text{ to } d)$ . These values are passed to Stage 2 in order to dimension the arrays  $\text{Last}(h)$ ,  $N(h, p, r)$ ,  $\text{Code}(h, p)$ , and  $\text{Cost}(h, p)$  used in the next stage as indicated next.

#### Stage 2

##### Dimensioning and preliminaries

$\text{Last}(h)$  is dimensioned to satisfy  $h \leq d$ , and the upper bound  $z_o(h)$  on the value of  $\text{Last}(h)$  is used in dimensioning the other arrays. Rather than employ an aggregate  $N(h, p, r)$  array, separate arrays  $N(2, p, r)$ , ...,  $N(d, p, r)$  are employed, where a given array  $N(h, p, r)$  is dimensioned for  $p \leq z_o(h)$  and  $r \leq h$ . (No array  $N(1, p, r)$  is required). Similarly, we use separate arrays  $\text{Code}(2, p)$  ...,  $\text{Code}(d, p)$  and  $\text{Cost}(1, p)$ , ...,  $\text{Cost}(d, p)$  dimensioned so that  $p \leq z_o(h)$  in each case. (Note the  $\text{Cost}(h, p)$  arrays differ from the others by starting at  $h = 1$  instead of  $h = 2$ .) As discussed later, for speed of access, particularly in the cases where  $d = 2$  or  $3$ , it can be advantageous to use 2-D rather than 3-D arrays, e.g.,  $N2(p, r)$ ,  $N3(p, r)$ , etc.

The coding array  $\theta(v)$  (which identifies index pairs  $(h, p)$  where the sets  $N_{hp}$  are stored) is dimensioned to attain a maximum  $v$  value of  $v = n(d - 1)$ . The elements of this array for  $v = n(d - 1) + 1$  to  $n(d)$  are omitted. These refer to sets  $N_{hp} = \{i_1, \dots, i_h\}$  such that  $h = d$  (hence  $|N_{hp}| = d$ ) and we do not need to code or decode the sets for  $h = d$  because, by Observation 2, none of them will change. The purpose of using  $\theta(v)$  in the

present organisation of the method is to be able to identify the location of new sets  $N_{p^*}$  that will be created and stored (for  $p^* = p[j]$  as in Observation 2). This likewise allows substantial saving of memory space.

The following code can be implemented in a manner that does not create the  $N(h, p, r)$  arrays but only retains their coded values in Stage 2. This option defers the generation of these arrays to the beginning of Stage 3 as a means of saving additional memory space at a slightly increased computational expense at the start of Stage 3. Such an approach can be used if memory space is at a premium.

Stage 2 consists of two phases, Phase 1 and Phase 2, as follows.

---

**Phase 1 Initialisation:**

```

 $\theta(v) = v$ , for  $v = 1$  to  $n$ 
    /* corresponding to indexes for the variables  $x_i$ ,  $i = v = 1$  to  $n$  */
 $\theta(v) = 0$ ,  $v = n + 1$  to  $n(d - 1)$ 
    /*  $\theta(v) = 0$  indicates that a set  $N_{hp} = \{i_1, \dots, i_h\}$  that will receive the
    coded value  $v$  has not yet been input. */
 $\text{Cost}(h,p) = 0$  and  $\text{Code}(h,p) = p$  for  $h = 1$  and  $p = 1$  to  $n$ .
    /* Treats the variables  $x_i$  for  $i = 1$  to  $n$  as if already input
    with 0 costs. */
 $\text{Last}(1) = n$  and  $\text{Last}(h) = 0$  for  $h = 2$  to  $d$ .

```

---



---

**Read in data and set up lists**

While data remain to be read

Read in the data triple  $(h, M, c)$  where  $M = \{i_1, \dots, i_h\}$ . Re-index, if necessary, so that  $i_1 < \dots < i_h$ .

Code the set  $M$  to identify  $v = V[M]$ . (If  $h = 1$ ,  $v = i_1$ .)

If  $\theta(v) = 0$  then

$\text{Last}(h) := \text{Last}(h) + 1$

$p = \text{Last}(h)$

$\text{Code}(h,p) = v$

/\* Create the array  $N_{hp}$  to save  $\{i_1, \dots, i_h\}$  \*/

Set  $N(h,p,r) = i_r$  for  $r = 1$  to  $h$ .

$\text{Cost}(h,p) = c$

$\theta(v) = p$

Else

/\*  $\theta(v) \neq 0$ , hence a record has already been created for the set

$\{i_1, \dots, i_h\}$ . Identify the location  $p$  where this record exists,

establishing  $N_{hp} = \{i_1, \dots, i_h\}$ . \*/

$p = \theta(v)$  (Implicit: if  $h = 1$ , then  $p = v = i_1$ .)

$\text{Cost}(h,p) := \text{Cost}(h,p) + c$

Endif

EndWhile

---

At the end of the foregoing process, if  $\text{Cost}(h, p) = 0$  for  $h > 1$  and for some  $p$  in the range from 1 to  $\text{Last}(h)$ , then reference to this pair  $(h, p)$  can be dropped. This situation can arise if different instances of the same set  $\{i_1, \dots, i_h\}$  are input (with their indexes in different order) and if the costs of these sets sum to 0, hence cancelling each other. Presumably this will be a rare occurrence, and the method will still function accurately by not bothering to drop such zero-cost elements. Consequently, checking for zero-cost elements may be considered as optional. If the checking is done, however, the operation proceeds as follows:

---

```

For h = 2 to d
  hLast = Last(h)
  For p = Last(h) to 1 /* in reverse order */
    If Cost(h,p) = 0 then
      /* write the hLast entry over the present one */
      Cost(h,p) = Cost(h,hLast)
      Nhp = Nh,hLast /* N(h,p,r) = N(h,hLast,r) for r = 1 to h */
      v = Code(h,hLast)
      Code(h,p) = v
       $\theta(v) = p$ 
      hLast = hLast - 1
    Endif
  Endfor p
  Last(h) = hLast
Endfor h

```

---

*Expand the records to include sets  $N_{hp}$  that may eventually receive non-zero costs*

We refer to temporary vectors  $T_r$  for  $r = 1$  to  $h$  where  $T_r = T(r, k)$   $k = 1$  to  $h - 1$ , which are only used for values of  $h$  between 2 and  $d - 1$ .

---

### Phase 2 Initialisation

LastNZ(h) = Last(h),  $h = 1$  to  $d$  /\* This records the index  $p = \text{Last}(h)$  for the last non-zero cost  $\text{Cost}(h,p)$  from Stage 1 at each level  $h$ . \*/

/\* Next examine the levels in reverse order. \*/

For  $h = d, d - 1, \dots, 3$

/\* The case for  $h = 2$  is not needed, because  $\{i_1, i_2\}$  contains the simple  $h - 1$  level subsets  $\{i_1\}$  and  $\{i_2\}$  and we don't need to look up the  $p$  indexes for these single element subsets since they are given directly by  $p = i_1$  and  $p = i_2$ . \*/

For  $p = 1$  to Last(h)

/\* Access the set  $N_{hp} = \{i_1, \dots, i_h\}$ ; i.e., where  $i_r = N(h,p,r)$  for  $r = 1$  to  $h$  \*/

/\* Identify the  $h$  different vectors  $T_r = N_{hp} - \{i_r\}$  for  $r = 1$  to  $h$ . Hence  $T_r = N_p - \{j\}$  for  $j = i_r$ . Each such vector, which contains  $h - 1$  of the  $h$  elements of  $N_{hp}$ , identifies one of the sets  $N_{p^*}$  of Observation 2, i.e., a set that can potentially be created from

```

flipping some variable  $x_j$ . */

For j = 1 to h
  /* Here j denotes the index that is dropped from  $N_{hp}$  to
  create  $T_r$ . */
  k = 0
  For r = 1 to h
    /* Store all indexes of  $N(h,p,r)$  in  $T_r$  except the
    index j. */
    If  $r \neq j$  then
      k := k + 1
       $T(r,k) = N(h,p,r)$ 
    Endif
  Endfor r
Endfor j

/* The temporary  $T_r$  sets are now created. Store each one as a level  $h-1$  set  $N_{h-1,p^*}$ ,
for an appropriate  $p^*$ , unless the  $T_r$  set is already stored as some set  $N_{h-1,p^*}$ .
In addition, for each r we set  $\text{Find}(h,p,r) = p^*$ , to be able to find the index  $p^*$ 
where the  $h^* = h-1$  level set associated with the element  $N(h,p,r)$  of  $N_{hp}$  is
located. We will need  $\text{Find}(h,p,r)$  in order to quickly locate  $N_{h^*p^*}$  ( $= N_{h-1,p^*}$ ) when
the variable  $x_j$  is flipped for  $j = N(h,p,r)$  (hence  $j = i_r$ , for  $N_{hp} = \{i_1, \dots, i_r, \dots, i_h\}$ ).
Then  $N_{h^*p^*}$  for  $p^* = \text{Find}(h,p,r)$  will be the set  $N_p - \{j\}$  of Observation 2. Since
the Find array is associated element for element with the  $N_{hp}$  array, we denote it
by  $\text{Find}_{hp}$ . */

 $h^* = h - 1$ 
For r = 1 to h
   $v^* = V[T_r]$  /* Obtain  $v^*$  by coding  $T_r = (T(r,1), \dots, T(r,h^*))$  */
   $p^* = \theta(v^*)$ 
  If  $p^* = 0$  then
    /* There is no record yet for the vector  $T_r$ . Hence it will become a new
     $N_{hp^*}$  which currently has a 0 cost. Update  $\text{Last}(h^*)$  ( $= \text{Last}(h-1)$ ) and
    make it the new  $p^*$ . */
     $\text{Last}(h^*) := \text{Last}(h^*) + 1$ 
     $p^* = \text{Last}(h^*)$ 
     $\text{Cost}(h^*, p^*) = 0$ 
    /* Record  $N_{h^*p^*} = T_r$  */
    For k = 1 to  $h^*$ 
       $N(h^*, p^*, k) = T(r,k)$ 
    Endfor k
    /* Record the location  $p^*$  of  $N_{h^*p^*}$  in the  $\theta(v)$  array */
     $\theta(v^*) = p^*$ 
  Endif

  /* If the preceding "if-then" statement does not apply, i.e., if  $p^* \neq 0$ , then
   $T_r$  and its appropriate record have already been created and stored as

```

```

        Nh*p* for p* =  $\theta(v^*)$ . Nothing needs to be done in this case. Finally, for
        both cases, record the entry Find(h,p,r) of the Findhp array. */
        Find(h,p,r) = p*
    Endfor r
Endwhile p
Endfor h

```

---

The temporary  $T(r, k)$  vectors have been introduced to make the foregoing process more understandable. These vectors can be instead generated without storing them at all, in an initial part of the loop for  $r = 1$  to  $h$ . This results in eliminating the loop for  $j = 1$  to  $h$ .

### Concluding step of stage 2

The arrays  $Last(h)$ ,  $LastNZ(h)$ , together with  $Code(h, p)$ ,  $Cost(h, p)$ , and the larger arrays  $N_{hp} = N(h, p, r)$ , and  $Find_{hp} = Find(h, p, r)$ , are placed a data file to be read in by the Generic PUB Method.  $Code(h, p)$  is not needed hereafter.

#### 6.3.1 Analysis for dimensioning the generic PUB algorithm

The array  $\theta(v)$ ,  $v = 1$  to  $n(d - 1)$  created in Stage 2 is not passed to the Generic PUB Algorithm, hence substantially reducing its overall memory requirements. (Likewise, a number of arrays used by the generic PUB algorithm were not required in the initial pre-processing operations of Stage 2, saving space in these routines as well).

The sets  $N_{hp}$  for  $h = 1$  to  $d$  and  $p = 1$  to  $Last(h)$  at the conclusion of Stage 2 encompass all sets  $\{i_1, \dots, i_h\}$  that may possibly receive non-zero costs. By Corollary 2.1 and its analysis, we know the number of these sets is at most  $U(d) = \sum(z_o(h): h = 1 \text{ to } d)$ . Each  $N_{hp}$  array and each associated  $Find_{hp}$  array will contribute an additional  $h \times z_o(h)$  entries to the allocated array space, since there are at most  $z_o(h)$  of these  $h$ -element arrays, and hence in total they contribute  $2\sum h \times z_o(h): h = 1 \text{ to } d$  to the array space.

The array space required by the generic PUB method to read in the data generated by Stage 2 is therefore as follows. Let  $N^h$  and  $Find^h$  denote the sets consisting of all arrays  $N_{hp}$  and  $Find_{hp}$  for  $p = 1$  to  $Last(h)$ .  $N^h$  and  $Find^h$  each contribute  $h \times Last(h)$  elements to array space that will be passed to the Generic PUB Algorithm. Actually, the  $Find_{hp}$  array is not stored for  $h = 1$ . Consequently, the arrays within  $N^h$  and  $Find^h$  consume a space of  $2N_{max} + n$ , where  $N_{max} = \sum(h \times Last(h): h = 2 \text{ to } d)$ . Finally, let  $Cost^h$  denote the set of arrays whose elements are given by  $Cost(h, p)$  for  $p = 1$  to  $Last(h)$ , which collectively consume an amount of array space equal to  $N_{sum} = \sum(Last(h): h = 1 \text{ to } d)$ .

In addition the maximum NQ value for dimensioning PoolQ and the  $q$  lists  $Loc(q)$ ,  $Before(q)$  and  $After(q)$  will equal  $N_{max}$ , by the following observation. We are interested in counting, for each  $i$ , the number of sets  $N_{hp}$  for  $h \geq 2$  that contain  $i$ , and then summing this number over all  $i$ . This is the same as summing the number of elements in each set  $N_{hp}$  such that  $h \geq 2$ , thereby equalling the indicated value for  $N_{max}$ .

One further array,  $Q_{hp}$  is generated inside of the generic PUB algorithm, and is dimensioned exactly as  $N_{hp}$ . Hence the collection  $Q^h$  has the same number of elements,  $h \times Last(h)$ , as  $N^h$  (and  $Find^h$ ). However, the sets  $Q^h$  are only created for  $h = 2$  to  $d - 1$ , and hence the collection of the  $Q^h$  sets consume an array space of  $N_Q = \sum(h \times Last(h): h = 2 \text{ to } d - 1)$ .

To summarise, let  $N^{\text{All}}$ ,  $\text{Find}^{\text{All}}$ ,  $Q^{\text{All}}$  and  $\text{Cost}^{\text{All}}$  respectively denote the collection of all  $N^h$ ,  $\text{Find}^h$ ,  $Q^h$  and  $\text{Cost}^h$  arrays. The collection of all arrays for the generic PUB method can then be dimensioned  $\text{Last}(d)$ ,  $\text{LastNZ}(d)$ ,  $\text{First}(n)$ ,  $\text{Before}(1 + N_{\max})$ ,  $\text{After}(N_{\max})$ ,  $\text{PoolQ}(N_{\max})$ ,  $\text{Loc}(N_{\max})$ ,  $N^{\text{All}}(N_{\max})$ ,  $\text{Find}^{\text{All}}(N_{\max})$ ,  $Q^{\text{All}}(N_Q)$  and  $\text{Cost}^{\text{All}}(N_{\text{sum}})$ .

The total array space consumed by the generic PUB algorithm, whose details are given next, is therefore  $1 + 2d + 6N_{\max} + N_Q + N_{\text{sum}}$ . By comparison, if we define  $N_{\max}^0$  and  $N_{\text{sum}}^0$  to be the larger quantities that replace  $\text{Last}(h)$  in the definitions of  $N_{\max}$  and  $N_{\text{sum}}$  by the bounds  $z_c(h)$ , then the total array space consumed by stage 2 is  $2d + n(d-1) + 2N_{\max}^0 + 2N_{\text{sum}}^0$ .

### 6.3.2 Generic PUB algorithm – completed organisation

#### Preliminaries

The array  $Q_{hp}$  ( $= Q(h, p, 1), \dots, Q(h, p, h)$ ) created in this algorithm accompanies the array  $N_{hp}$  by indicating the value  $q = Q(h, p, r)$  for each  $r = 1$  to  $h$  such that the index  $p$  is stored on the  $q$ -list, by the relationship  $p\text{Loc}(q) = p$ . [The same  $p$  index may be accessed by more than one  $q$ , but the level  $h$  will differ, so that the pair  $(h, p)$  given by  $h = h\text{Loc}(q)$  and  $p = p\text{Loc}(q)$  will be unique for each  $q$ , i.e., no two  $q$  indexes will access the same pair  $(h, p)$ ]. The array  $Q_{hp}$  is used to identify the index  $q$  and hence the index  $p$  that is to be dropped, as explained below. For clarity, the relationships between the arrays  $N_{hp}$ ,  $\text{Find}_{hp}$  and  $Q_{hp}$  are illustrated in Appendix 1.

We also introduce an Option E (where E denotes ‘Evaluation’), which provides for an enhanced way to evaluate candidate moves for selection. Option E is divided into four parts E(1), ..., E(4), whose components are described in Section 7, to exploit an additional property of profitable moves that derives from Observations 1 and 2. We enclose reference to Option E in square brackets ‘[ ]’ to indicate that its details are described externally.

#### Generic PUB method completed

---

##### Initialisation

Read in  $\text{Last}(h)$ ,  $\text{LastNZ}(h)$ ,  $h = 1$  to  $d$ , and  $\text{Cost}(h,p)$  for  $h = 1$  to  $d$  and  $p = 1$  to  $\text{Last}(h)$ .

Read in  $N(h,p,r)$  and  $\text{Find}(h,p,r)$  for  $h = 1$  to  $d$ ,  $p = 1$  to  $\text{Last}(h)$  and  $r = 1$  to  $h$ .

---

##### Generate the sets P(i)

Set  $\text{First}(i) = 0$  for all  $i \in N$ .  $\text{Before}(0) = 0$ .

[Option E(1): Initialise supplemental evaluation.]

$q\text{Now} = 0$

/\* Next, set up  $P(i)$  to refer to those indexes  $p$  such that  $N_{hp}$  contains  $i$  and such that  $\text{Cost}(h,p)$  is non-zero. Each  $P(i)$  is recorded by means of the lists  $\text{Before}(q)$  and  $\text{After}(q)$ , starting with  $q = \text{First}(i)$ , while simultaneously recording  $Q_{hp} = Q(h,p,r)$ ,  $r = 1$  to  $h$ . To restrict  $P(i)$  to refer to non-zero cost terms, only look at indexes  $p$  that are accessed for  $p = 1$  to  $\text{LastNZ}(h)$ , limited to levels  $h = 2$  to  $d$  since the level  $h = 1$  is treated separately.\*/

For  $h = 2$  to  $d$

    For  $p = 1$  to  $\text{LastNZ}(h)$

```

/* Nhp has already been created in Stage 2, and read in to the
Generic PUB routine during its initialisation. */
[Option E(2): Complete initialisation of supplemental evaluation.]
For r = 1 to h
  i = N(h,p,r) /* i = ir for Nhp = {i1, ..., ih} */
  qNow := qNow + 1
  qPrevious = First(i)
  After(qNow) = qPrevious
  Before(qPrevious) = qNow
  First(i) = qNow
  hLoc(qNow) = h
  pLoc(qNow) = p
  rLoc(qNow) = r
  Q(h,p,r) = qNow
EndFor r
EndWhile p
Endfor h

/* Set up the original PoolQ, in the form PoolQ(1) to PoolQ(NQ), to store q indexes not yet used
that may potentially be needed. */
qLast = qNow
For q = qLast to Nmax
  PoolQ(q - qLast + 1) = q
EndFor
NQ = Nmax - qLast + 1

```

---

### 6.3.2.1 Updating by observation 2 when $x_j$ is flipped

In the next segment, given the index  $j$  of the variable  $x_j$  that has been flipped, each  $p \in P(j)$  is examined via the lists Before(q) and After(q), identifying  $p = pLoc(q)$  (starting from  $q = First(j)$ ). The operations first update the cost for set  $N_{hp}$  by reversing the sign of  $Cost(h, p)$ ; i.e., setting  $c = Cost(h, p)$  and then setting  $Cost(h, p) := -c$ . Updates are then performed on the set  $N_{hp} - \{j\}$  by identifying  $p^* = p[j]$  as in the Generic PUB Algorithm and setting  $Cost(h, p^*) := c + Cost(h, p^*)$ . In our organisation,  $p^*$  is obtained by  $p^* = Find(h, p, r)$  where  $r$  is identified by  $j = N(h, p, r)$ , i.e.,  $N_{hp} = \{i_1, \dots, i_r, \dots, i_h\}$  and  $i_r = j$ . Note that the index  $p^*$  refers to a set at level  $h - 1$ , where for  $h^* = h - 1$  the set  $N_{h^*p^*}$  has the same components as  $N_{hp}$  except for  $i_r = j$ .

---

```

q = First(j)
While q > 0
  p = pLoc(q)
  h = hLoc(q)
  r = rLoc(q)

```

```

c = Cost(h,p)
Cost(h,p) = - c
  [Option E(3)]
p* = Find(h,p,r)
  /* Given p* for level h* = h - 1, check to see if Nh*,p* currently exists, i.e., if
  Cost(h*,p*) is non-zero. If costs are not integers, then Cost(h*,p*) = 0 may be
  established by |Cost(h*,p*)| < ε where ε is a small positive value. */
h* = h - 1
If h* = 1 then skip the following "If-then" sequence, picking up q :=
After(q) for the next iteration of the loop.
  [Option E(4)]
If Cost(h*,p*) = 0 then
  /* Create Nh*,p*: this entails adding a new q* to access Nh*,p* so that the path P(i) can
  include reference to p* by finding it stored in Loc(q*), i.e., by accessing the pair
  (h*,p*) = Loc(q*), or the triple (h*,p*,r) = Loc(q*). */
  For r* = 1 to h*
    i = N(h*,p*,r*)
    /* Examine each entry i = ir* of Nh*,p* = (i1, ..., ir*, ..., ih*) as r* goes from 1 to h*,
    and add a new q, denoted by q*, to the list P(i). Likewise, add p* to P(i): Start by
    getting an available index q* from PoolQ. */
    q* = PoolQ(NQ)
    NQ := NQ - 1
    qFirst = First(i)
    After(q*) = qFirst
    Before(qFirst) = q*
    First(i) = q*
    hLoc(q*) = h*
    pLoc(q*) = p*
    rLoc(q*) = r*
    /* Create entry r* of the Q(h*,p*,r*) array */
    Q(h*,p*,r*) = q*
  Endfor r*
  /* Give Nh*,p* a cost = Cost(h*,p*) + c = 0 + c. */
  Cost(h*,p*) = c
Else
  /* Here the cost for Nh*,p* is non-zero, and Nh*,p* is already recorded. */
  Cost(h*,p*) := Cost(h*,p*) + c
  /* Check the changed value of Cost(h*,p*) */
  If Cost(h*,p*) = 0 then
    /* Since the cost for Nh*,p* changes from non-zero to zero, drop p* from P(i) for
    each i in Nh*,p* so that p* and Nh*,p* will not be accessed at level h* = h - 1.
    Exploit the list Qh*,p* to do this. */
    For r* = 1 to h*

```

```

/* Examine each  $q^* = q_{r^*}$  of  $Q_{h^*,p^*} = (q_1, \dots, q_{r^*}, \dots, q_{h^*})$  as  $r^*$  goes from 1 to
 $h^*$ , where  $q^*$  is the value of  $q$  such that  $p^*$  is accessed by  $p^* = pLoc(q^*)$ ,
locating  $p^*$  on the list  $P(j)$ . */
i = N(h*,p*,r*)
q* = Q(h*,p*,r*)
If First(i) = q* then
    First(i) = After(q*)
Else
    qB = Before(q*)
    qA = After(q*)
    Before(qA) = qB
    After(qB) = qA
Endif
/* Put q* back on PoolQ */
NQ := NQ + 1
PoolQ(NQ) = q*
EndFor r*
Endif
q := After(q)
Endwhile

```

A way to additionally streamline the foregoing operations and employ reduced memory for the case of quadratic and cubic polynomials is given in Appendix 2.

### 6.3.2.2 Memory implications

To illustrate the memory requirements of this organisation, we take two examples from 3-SAT problems in Kochenberger (2010) whose PUB formulations give  $n = 200$  and  $n = 600$ , respectively. For this purpose we have selected the instances for  $n = 200$  and  $n = 600$  that contain the largest number of initial non-zero costs for each of these two values of  $n$ . From this initial non-zero cost data we are able to compute the bounds  $z_o(h)$  on the number of non-zero costs at each level  $h$  by Corollary 2.1, and then use the observations of Section 6.3.1 to identify the total memory requirements. We report these requirements separately for Stage 2 and for the generic PUB stage of the algorithm, since these requirements are independent in the sense that only the larger of the two requirements determines the memory consumed at any point by the algorithm.

In the case of the generic PUB stage, without more extensive processing we do not know the values  $Last(h)$  for  $h = 1$  to  $d$  that give the values  $N_{max}$ ,  $N_{sum}$  and  $N_Q$  to precisely identify the total memory employed, but we can substitute the upper bounds  $z_o(h)$  for the values  $Last(h)$  to obtain an over-estimate of the total memory. Consequently we use the same values  $N_{max}^0$  and  $N_{sum}^0$  used to identify the memory requirements for Stage 2, together with an associated value  $N_Q^0$  that over-estimates  $N_Q$ . We note for 3-SAT problems that  $d = 3$ .

*Illustration for n = 200*

The initial number of non-zero costs are given by  $nz_0(1) = 157$ ,  $nz_0(2) = 1,553$  and  $nz_0(3) = 1,122$ . Applying Corollary 2.1, we have the following upper bounds on the number of non-zero costs that the algorithm can generate at levels 3 and 2:

$$z_0(3) = nz_0(3) = 1,122,$$

$$z_0(2) = 1,553 + \text{Min}(\Delta(2), 1,553 + 3(1,122)) = \text{Min}(19,900, 4,919) = 4,919.$$

The value  $z_0(1)$  is limited by  $\Delta(1) = n = 200$ . (Information about level 1 always explicitly refers to all of the indexes 1 to  $n$  in any case.)

By the definitions at the end of Section 6.3.1 we have  $N_{\max}^0 = 13,204$ ,  $N_Q^0 = 9,838$ , and the value  $n(d-1) = 200 + 200 \times 199/2 = 20,100$ . The associated formulas for total memory from Section 6.3.1 are:

$$\text{Stage 2} \quad \text{Memory} = 2d + n(d-1) + 2N_{\max}^0 + 2N_{\text{sum}}^0$$

$$\text{Generic PUB stage} \quad \text{Memory} = 1 + 2d + 6N_{\max}^0 + N_{\text{sum}}^0 + N_Q$$

Consequently, we obtain

$$\text{Stage 2 memory} \quad 58,996$$

$$\text{Generic PUB memory} \quad 95,310.$$

These values may be compared to the values  $n^3 = 8,000,000$  and  $n(d) = 258,900$ , for the classical and full coded memory representations. This shows that even a full reliance on coded memory throughout the algorithm, which results in less efficient processing, consumes somewhat more memory than our approach for exploiting sparsity. In fact, the value  $n(d) = 258,900$  underestimates the amount of memory used by the full coding by a factor of 2 or more, because additional arrays are needed to execute the method.

*Illustration for n = 600*

The initial number of non-zero costs in this instance are given by  $nz_0(1) = 453$ ,  $nz_0(2) = 3,768$  and  $nz_0(3) = 2,550$ . Upper bounds on the number of non-zero costs at levels 3 and 2 are thus:

$$z_0(3) = nz_0(3) = 2,550$$

$$z_0(2) = 11,418$$

The value  $z_0(1)$  is limited by  $\Delta(1) = n = 600$ .

We again apply the formulas:

$$\text{Stage 2 : Memory} = 2d + n(d-1) + 2N_{\max}^0 + 2N_{\text{sum}}^0$$

$$\text{Generic PUB stage : Memory} = 1 + 2d + 6N_{\max}^0 + N_{\text{sum}}^0 + N_Q$$

where in this case  $N_{\max}^0 = 30,486$ ,  $N_{\text{sum}}^0 = 13,968$ ,  $N_Q^0 = 22,836$ , and the value  $n(d-1) = 600 + 600 \times 599/2 = 180,300$ . Then we obtain:

$$\text{Stage 2 Memory} = 269,214$$

$$\text{Generic PUB Memory} = 219,727$$

These values may be compared to the values  $n^3 = 216,000,000$  and  $n(d) = 36,000,500$  for the classical and full coded memory representations. We see that the memory savings provided by our method for exploiting sparse data become more dramatic as the size of the problem increases.

## 7 Multi-flip moves and enhanced evaluations

The preceding analyses suggest that the total number of potential moves grows rapidly enough so that 2-flip moves will usually be the practical limit for examining all moves of a given class. However, candidate list strategies as described in Glover and Laguna (1997) can enable moves involving somewhat larger numbers of flips to be evaluated and used. This is particularly true in the case of polynomials of degree 3 and especially of degree 2, where methods subsequently described make it possible to perform exceedingly efficient updates. In these cases, the filter-and-fan strategy of Glover (1997), which builds on basic candidate list ideas, provides an additional opportunity for exploiting multi-flip moves.

We begin by examining special ways to take advantage of 2-flip moves for polynomials of arbitrary degree, and then show how to build on these processes to provide useful methods for generating 3-flip and higher order moves by means of natural candidate list ideas, accounting for the fact that the evaluation of higher order moves involves some sacrifice of the efficiencies available in the 2-flip case.

The Generic PUB Algorithm has the useful property that the effort required to evaluate 2-flip moves is independent of the degree  $d$  of the polynomial. The only difference entailed by higher degree polynomials is that more work is required in updating the PUB representation after a move is made, but the work to evaluate a move involving a given number of flips is independent of this degree.

Specifically, for a polynomial of any degree, a 2-flip move requires examining only the sets  $N_p$  of the form  $N_p = \{i_1, i_2\}$  together with the two associated sets  $N_{i_1} = \{i_1\}$  and  $N_{i_2} = \{i_2\}$ . If 1-flip moves are evaluated separately, then an option for evaluating 2-flips is to restrict attention to sets  $N_p$  for  $p \in P$ ; i.e., for which  $c_p \neq 0$ . The organisation that differentiates the sets  $N_p$  by level, hence creating sets  $N_{hp}$  for  $h = 1$  to  $d$ , makes it possible to carry out these evaluations efficiently.

In general, if we restrict attention to evaluating only 1-flip and 2-flip moves, we can obtain enhanced evaluations by taking advantage of an additional consequence of Corollaries 1.1 and 1.2 in Section 2.

*Corollary 1.3:* If  $x' = 0$  constitutes a local optimum relative to:

- 1 1-flip moves, then it is also a local optimum relative to 2-flip moves unless  $c_p < 0$  for some  $p$  such that  $|N_p| = 2$ .

- 2 both 1-flip and 2-flip moves, then it is a local optimum for 3-flip moves unless  $c_p < 0$  for some  $p$  such that  $|N_p| = 3$ , or unless there exists some  $i \in N$  such that  $i$  belongs to 2 sets  $N_p$  having  $|N_p| = 2$  and  $c_p < 0$ .

*Proof:* Represent the coefficient  $c_p$  of a set  $N_p = \{i_1, \dots, i_h\}$  by  $c(i_1, \dots, i_h)$  and identify the cost of flipping all elements in  $N_p$ , whether or not  $p \in P$  (i.e.,  $c_p \neq 0$ ) by  $\text{FlipCost}(i_1, \dots, i_h) = \sum(c(k_1, \dots, k_r) : \{k_1, \dots, k_r\} \subset \{i_1, \dots, i_h\})$ . By Corollaries 1.1 and 1.2, if  $x' = 0$  is a local optimum relative to 1-flip moves, then  $c_i \geq 0$  for all  $i \in N$ , and the only relevant sets  $N''$  for 2-flip moves consist of those  $N_p$  of the form  $N_p = N_{2p} = \{i_1, i_2\}$  together with their subsets  $\{i_1\}$  and  $\{i_2\}$ . The solution  $x' = 0$  fails to be a local optimum for 2-flip moves only if for some such  $N_p$  we have  $c(i_1, i_2) + c(i_1) + c(i_2) < 0$  which immediately implies  $c_p < 0$ . On the other hand, if  $x' = 0$  is a local optimum relative to both 1-flip and 2-flip moves, we additionally have  $c(i_1, i_2) + c(i_1) + c(i_2) \geq 0$  for all  $p$  such that  $N_p$  has the form  $N_p = \{i_1, i_2\}$ . A 3-flip involving a set  $N'' = \{i_1, i_2, i_3\}$  has  $\text{FlipCost}(i_1, i_2, i_3) = c(i_1, i_2, i_3) + c(i_1, i_2) + c(i_2, i_3) + c(i_1, i_3) + c(i_1) + c(i_2) + c(i_3)$  (understanding a component costs that does not correspond to some  $c_p$  for  $p \in P$  to be 0, including  $c(i_1, i_2, i_3)$  if  $d < 3$ ). Denying the assertion (2) of the corollary, we have  $c(i_1, i_2, i_3) \geq 0$  and no index  $i$  belongs to two of the three sets  $\{i_1, i_2\}$ ,  $\{i_2, i_3\}$ ,  $\{i_1, i_3\}$  whose associated costs  $c(i_1, i_2)$ ,  $c(i_2, i_3)$  and  $c(i_1, i_3)$  are negative. Equivalently, at least two of the three costs  $c(i_1, i_2)$ ,  $c(i_2, i_3)$  and  $c(i_1, i_3)$  must be non-negative. Suppose these costs are  $c(i_1, i_2)$  and  $c(i_2, i_3)$ . Group the components of  $\text{FlipCost}$  to give  $\text{FlipCost}(i_1, i_2, i_3) = c(i_1, i_2, i_3) + c(i_1, i_2) + c(i_2, i_3) + c(i_2) + (c(i_1, i_3) + c(i_1) + c(i_3))$ . Each of the first three terms is non-negative by denying the conclusion of (2), while the two terms of the sum  $c(i_2) + (c(i_1, i_3) + c(i_1) + c(i_3))$  are both non-negative by the assumption that  $x' = 0$  is locally optimal relative to 1-flips and 2-flips. This completes the proof by contradiction.

The foregoing result has implications for choice rules in algorithms that fall within the domain of the Generic PUB Algorithm by creating a bias toward selecting moves that are components of profitable  $q$ -flip moves, where  $q$  is larger than we can consider for a direct evaluation. Define:

$$P^-(j) = \{p \in P(j) : c_p < 0\}$$

and determine an evaluation for flipping  $x_j$  given by  $\text{Eval}(j) = \text{Eval}_1(j)$  or  $\text{Eval}_2(j)$ , where:

$$\text{Eval}_1(j) = |P^-(j)|$$

$$\text{Eval}_2(j) = -\sum(c_p : p \in P^-(j))$$

Alternatively,  $\text{Eval}(j)$  may be determined as a linear combination of  $\text{Eval}_1(j)$  and  $\text{Eval}_2(j)$ . Note the index  $j$  will automatically be excluded from  $P^-(j)$  under the local optimality assumption. Then, in the situation where  $x' = 0$  is locally optimal for 1-flip moves, or for both 1-flip and 2-flip moves, we choose to flip a variable  $x_j$  such that  $j = \arg \max(\text{Eval}(i) : i \in N)$ . If we further restrict the definition of  $P^-(j)$  to be given by  $P^-(j) = \{p \in P(j) : c_p < 0 \text{ and } |N_p| \leq 3\}$ , then  $\text{Eval}(j)$  is designed to favour the selection of a variable  $x_j$  that may be a component move within an improving 3-flip move. By extension, the foregoing rule is a way of selecting  $x_j$  in a manner that increases the chance that it will be a component of some higher level flip move.

*Method to exploit corollary 1.3: option E*

Corollary 1.3 gives rise to an Option E as previously referred to in the Generic PUB code. The supplemental evaluation based on  $\text{Eval}_1(j)$  and  $\text{Eval}_2(j)$  proceeds as follows, where the indicated code can be inserted directly in the locations previously identified in Section 6.3.2. We note that we do not have to include reference to  $c_p$  for  $p \in N$  because  $\text{Eval}_1(j)$  and  $\text{Eval}_2(j)$  are only referenced when a 1-flip is not improving, i.e., when  $c_p \geq 0$ .

---

Option E(1): *Initialise supplemental evaluation*

$\text{Eval}_1(i) = 0$  and  $\text{Eval}_2(i) = 0$  for all  $i \in N$ .

Option E(2): *Completed initialisation of supplemental evaluation*

$c = \text{Cost}(h, p)$

If  $c < 0$  then

    For  $r = 1$  to  $h$

$i = N(h, p, r)$

$\text{Eval}_1(i) = \text{Eval}_1(i) + 1$

$\text{Eval}_2(i) = \text{Eval}_2(i) - c$

    Endfor  $r$

Endif

Option E(3)

If  $c > 0$  then

$\text{Eval}_1(j) := \text{Eval}_1(j) + 1$

$\text{Eval}_2(j) := \text{Eval}_2(j) - c$

Elseif  $c < 0$  then

$\text{Eval}_1(j) := \text{Eval}_1(j) - 1$

$\text{Eval}_2(j) := \text{Eval}_2(j) + c$

Endif

Option E(4)

$c^* = \text{Cost}(h^*, p^*)$

If  $c^* \geq 0$  and  $c + c^* < 0$  then

    For  $r^* = 1$  to  $h^*$

$i = N(h^*, p^*, r^*)$

$\text{Eval}_1(i) := \text{Eval}_1(i) + 1$

$\text{Eval}_2(i) := \text{Eval}_2(i) - (c + c^*)$

    Endfor  $r^*$

Elseif  $c^* < 0$  and  $c + c^* \geq 0$  then

    For  $r^* = 1$  to  $h^*$

$\text{Eval}_1(i) := \text{Eval}_1(i) - 1$

$\text{Eval}_2(i) := \text{Eval}_2(i) + (c + c^*)$

    Endfor  $r^*$

Endif

---

The justification of the foregoing code segments is provided by Corollary 1.3, in conjunction with Observation 2 and the general structure of the Generic PUB Algorithm.

## 8 Concluding remarks

The PUB optimisation problem affords a model that encompasses problems significantly more general than those captured by the widely studied quadratic model. Our results for the PUB problem enable 1-flip, 2-flip and higher level flip moves (within limits of computational complexity) to be executed efficiently. These results give a framework for a generic PUB algorithm whose evaluation routines can be embedded in a variety of existing search methods. Special data structures and streamlined updating methods are introduced that offer additional efficiencies.

We particularly focus on organising our Generic PUB Algorithm to take advantage of large and sparse problems, which typically arise in practical applications. By incorporating a coding procedure in a preliminary stage and employing special data structures and updating algorithms to perform the main computations of the PUB method, our approach offers the potential to solve such practical problems with greater efficacy and reduced computational effort.

## References

- Glover, F. (1997) ‘A template for scatter search and path relinking’, in Hao, J-K., Lutton, E., Ronald, E., Schoenauer, M. and Snyers, D. (Eds.): *Artificial Evolution, Lecture Notes in Computer Science, 1363*, pp.13–54, Springer, Berlin.
- Glover, F. and Hao, J.K. (2010a) ‘Efficient evaluations for solving large 0-1 unconstrained quadratic optimization problems’, *International Journal of Metaheuristics*, Vol. 1, No 1, pp.3–10.
- Glover, F. and Hao, J.K. (2010b) ‘Fast 2-flip move evaluations for binary unconstrained quadratic optimisation problems’, *International Journal of Metaheuristics*, Vol. 1, No. 2, pp.100–107.
- Glover, F. and Laguna, M. (1997) *Tabu Search*, Kluwer Academic Publishers, Boston.
- Glover, F., Hao, J.K. and Kochenberger, G. (2010) ‘Polynomial unconstrained binary optimization – part 1’, *International Journal of Metaheuristics*, Vol. 1, No. 3, pp.232–256.
- Glover, F., Kochenberger, G., Alidaee, B. and Amini, M. (1998) ‘Tabu search with critical event memory: an enhanced application for binary quadratic programs’, in Voss, S., Martello, S., Osman, I.H. and Roucairol, C. (Eds.): *Meta-Heuristics – Advances and Trends in Local Search Paradigms for Optimization*, pp.83–109, Kluwer Academic Publishers, Boston/Dordrecht/London.
- Hanafi, S., Rebai, A-R. and Vasquez, M. (2010) ‘Several versions of the devour digest tidy-up heuristic for unconstrained binary quadratic problems’, Working paper, LAMIH, Université de Valenciennes.
- Kochenberger, G. (2010) ‘Notes on 3-SAT and max 3-SAT’, Working paper, University of Colorado, Denver.
- Kochenberger, G., Glover, F., Alidaee, B. and Rego, C. (2004) ‘A unified modeling and solution framework for combinatorial optimization problems’, *OR Spectrum*, Vol. 26, No. 2, pp.237–250.
- Pardalos, F. and Xue, J. (1994) ‘The maximum clique problem’, *The Journal of Global Optimization*, Vol. 4, No. 3, pp.301–328.
- Pardalos, P. and Rodgers, G.P. (1990) ‘Computational aspects of a branch and bound algorithm for quadratic zero-one programming’, *Computing*, Vol. 45, No. 2, pp.131–144.

## Appendix 1

### *Illustrations of relationships between component*

#### *Data structures*

##### *A.1.1 Relationship between $N_{hp}$ and $Find_{hp}$*

For purposes of illustration we represent  $N_{hp}$  and  $Find_{hp}$  by  $N(h, p)$  and  $Find(h, p)$ . Consider these two associated vectors for level  $h = 3$  and  $p = 23$ , where  $p$  is arbitrarily given the value 23, to yield the two vectors  $N(3, 23)$  and  $Find(3, 23)$  as follows. (Note the index  $h = 3$  identifying the third level is included at the start of each vector so that all vectors at this level can be conveniently accessed by holding  $h$  constant at 3).

$$N(3, 23) = (4, 7, 13) \text{ (hence, } N(3, 23, 1) = 4; N(3, 23, 2) = 7; N(3, 23, 3) = 13)$$

$$Find(3, 23) = (18, 34, 28) \text{ (hence, } Find(3, 23, 1) = 18; Find(3, 23, 2) = 34; Find(3, 23, 3) = 28)$$

Each component of the  $Find(3, 23)$  vector identifies a  $p^*$  value at level  $h - 1 = 2$  that yields a reduced level 2 form of the level 3 vector  $N(3, 23)$ . Specifically,  $Find(3, 23)$  identifies 3 different level 2 vectors derived from  $(4, 7, 13)$ , consisting of  $(7, 13)$ ,  $(4, 13)$  and  $(4, 7)$ . Each of these level 2 vectors drops one element of  $(4, 7, 13)$ ; i.e.,  $(7, 13)$  drops the first element,  $(4, 13)$  drops the second element and  $(4, 7)$  drops the third element. (Consequently, these three vectors correspond to  $N_{hp} - \{j\}$  as  $j$  successively equals the components 4, 7 and 13 of  $N_{hp}$ .) In short, the three reduced level 2 vectors are identified by the three  $p^*$  values  $(18, 34, 28)$  so that  $N(2, 18) = (7, 13)$ ,  $N(2, 34) = (4, 13)$  and  $N(2, 28) = (4, 7)$ . As in the case of the  $p$  value of 23, the  $p^*$  values 18, 34 and 28 are chosen arbitrarily for concreteness. (These values will in fact be determined by the sequence in which data elements are read into the problem.) An image of how the two vectors relate positionally can be gained by again writing the  $Find(3, 23)$  vector below the  $N(3, 23)$  vector as follows.

$$N(3, 23) = (4, 7, 13)$$

$$Find(3, 23) = (18, 34, 28)$$

We use the symbol # to indicate that the corresponding component of a vector is dropped, and portray the relationship between  $N(3, 23)$  and  $Find(3, 23)$  by writing:

$$N(2, 18) = (\#4, 7, 13); N(2, 34) = (4, \#7, 13); N(2, 28) = (4, 7, \#13); \text{ thus giving}$$

$$N(2, 18) = (7, 13); N(2, 34) = (4, 13); N(2, 28) = (4, 7)$$

Level 2 vectors constitute a special case that is handled differently from level  $h$  vectors for  $h > 2$ . We illustrate this special case starting with  $N_{hp}$  and  $Find_{hp}$  for  $h = 2$ . We again choose a value  $p$  arbitrarily (here  $p = 13$ ) to give the two associated vectors  $N(2, 13)$  and  $Find(2, 13)$ :

$$N(2, 13) = (5, 7)$$

$$Find(2, 13) = (7, 5)$$

In contrast to the arbitrary designation  $p = 13$ , the  $p^*$  values of the Find vector; i.e., the ‘7’ and the ‘5’ of  $\text{Find}(2, 13) = (7, 5)$ , are not arbitrary, but determined by the entries of  $N(2, 13) = (5, 7)$ , which  $\text{Find}(2, 13) = (7, 5)$  duplicates in reverse order. To see why this is so, recall that the  $p^*$  entries of  $\text{Find}(2, 13) = (p_1, p_2)$  respectively identify the vectors  $N(2, 13) - \{5\}$  and  $N(2, 13) - \{7\}$ , corresponding to  $N_{hp} - \{j\}$  as  $j$  ranges over the entries of  $N_{hp}$ . (We take the liberty of mixing set notation and vector notation, since the interpretation is obvious.) Using the # symbol as introduced above, we can write:

$$N(1, p_1) = (\#5, 7); N(1, p_2) = (5, \#7), \text{ thus giving}$$

$$N(1, p_1) = (7); N(1, p_2) = (5)$$

These two expressions show that the  $p^*$  values  $p_1$  and  $p_2$  are compelled to be  $p_1 = 7$  and  $p_2 = 5$ , hence giving  $N(1, 7) = (7)$  and  $N(1, 5) = (5)$ . This accords with the convention that the  $p$  value in the situation where  $N_{hp} = \{i\}$  is given by  $p = i$ . Here we have applied this convention in generating the vector  $N_{h-1, p^*}$ .

This special case illustrates that when  $h = 2$  we don’t need to use the vector  $\text{Find}_{hp}$  to generate the vectors  $N_{h-1, p^*}$  from  $N_{hp}$ , in contrast to cases where  $h > 2$ . Specifically, knowledge of the contents of any level 2 vector  $N_{2p} = (i_1, i_2)$  immediately gives the two associated  $N_{h-1, p^*}$  vectors, which are just  $(i_2)$  in the case where  $j = i_1$  and  $(i_1)$  in the case where  $j = i_2$ . Moreover, equally important, we also know where these vectors are stored because of the convention that  $N(1, p) = p$ ; i.e., the ‘vector’  $(i_2)$  is accessed by  $N_{hp}$  for  $p = i_2$  and  $(i_1)$  is accessed by  $N_{hp}$  for  $p = i_1$ .

### A.1.2 Relationship between $N_{hp}$ and $Q_{hp}$ as used in generating elements of a list $P(j)$

We next consider the relationship between the arrays  $N_{hp}$  and  $Q_{hp}$  and then combine this information with information involving the array  $\text{Find}_{hp}$  to update problem arrays when a variable  $x_j$  is flipped. Suppose  $d = 3$  and we want to generate elements of the list  $P(j)$  for  $j = 7$ . Assume all of the sets  $N_{hp}$  for  $h = 1$  to  $h = 3 (= d)$  are given in Table A1. We also identify associated  $Q_{hp}$  vectors for  $h = 2$  and  $h = 3$  (the case for  $h = 1$  is superfluous) of the form  $Q_{2p} = (q_1, q_2)$  and  $Q_{3p} = (q_1, q_2, q_3)$ . The connections between the  $N_{hp}$  and  $Q_{hp}$  vectors are elaborated Table A1.

**Table A1** Connections

		$P(7)$	
$h$	$p$	$N_{hp}$	$Q_{hp}$
1	7	7	
2	13	5, 7	28, 67
	18	7, 8	45, 33
3	23	4, 7, 13	48, 79, 63
	27	5, 7, 33	62, 91, 88
	18	7, 8, 52	35, 37, 26
	19	7, 15, 30	49, 28, 58



Examination of Table A1 verifies the contents of the Before and After arrays. For example, for  $h = 3$ , the vectors  $\text{Loc}(q) = (h, p, r)$  that yield  $N(h, p, r) = 7$  are given for  $q = 79, 91, 35$  and  $49$ . (The  $q$  values do not have to appear in exactly this sequence in the trace of the Before( $q$ ) and After( $q$ ) lists. Variation will occur according to the sequence in which the generic PUB algorithm adds and deletes elements from these lists.) The validity of the Before and After arrays for  $h = 2$  may be verified similarly.

### A.1.3 Consequences of flipping a variable

Now, we illustrate the process of updating the arrays when the variable  $x_7$  is flipped. First, consider the effect of flipping  $x_7$  on the level 3 set  $N_{hp} = N_{3,23}$ , which is the first element we encounter by tracing the  $P(7)$  list at level 3; i.e., we obtain  $q = \text{First}(3,7) = 79$ , and as previously noted,  $\text{Loc}(79) = (3, 23, 2)$  giving  $h = 3, p = 23$  and  $r = 2$ . [We already know  $h = 3$  as a result of tracing the  $q$  lists at level 3, hence we only pick up  $p = p\text{Loc}(79) = 23$  and  $r = r\text{Loc}(79) = 2$ ]. From Table A1 the information for  $h = 3$  and  $p = 23$  is given by:

$h$	$p$	$N_{hp}$	$Q_{hp}$
3	23	4, 7, 13	48, 79, 63

To determine the effect of flipping  $x_7$  we now make use of the  $\text{Find}_{hp}$  array, which was illustrated earlier in association with  $N_{hp} = N(3, 23) = (4, 7, 13)$ , giving  $\text{Find}(3, 23) = (18, 34, 28)$ , where:

$$N(2, 18) = (7, 13); N(2, 34) = (4, 13); N(2, 28) = (4, 7)$$

The flip of  $x_7$  affects  $N_{hp}$  by uniquely generating the array  $N_{h^*p^*} = N(2, 34) = (4, 13)$ . We know the coordinates  $h^* = 2$  and  $p^* = 34$  of  $N(2, 34)$  because  $h^* = h - 1$  and  $34$  is in the second position of  $\text{Find}(3, 23) = (18, 34, 28)$  just as  $j = 7$  is in the second position of  $N(3, 23) = (4, 7, 13)$ . Moreover, we know this position  $r$  because this is precisely the value given by  $r\text{Loc}(q)$ , which we found in the beginning by accessing  $q = 79$  and obtaining  $r\text{Loc}(79) = 2$ .

In sum, then, we have determined that flipping  $x_7$  directly affects the set level 3 set  $N_{hp} = N(3, 23)$  because this set is encountered on  $P(7)$  for  $q = 79$ , and the flip also affects the set  $N_{h^*p^*} = N(2, 34)$  whose coordinate  $p^* = 34$  we were able to identify using the  $\text{Find}_{hp}$  array and the position  $r\text{Loc}(79) = 2$ , yielding  $p^* = \text{Find}(h, p, 2)$ . Finally, by Observation 2, the new costs for the two affected sets are given by setting  $c = \text{Cost}(h, p) = \text{Cost}(3, 23)$ , and then setting  $\text{Cost}(3, 23) = -c$  and  $\text{Cost}(2, 34) := \text{Cost}(2, 34) + c$ .

### Illustration for level 2

To complete the illustration of the consequences of flipping a variable, we show the special case involving Level 2 sets. Consider the stage of tracing the  $P(7)$  list that examines elements on this list at Level 2. (The order in which the levels are examined is immaterial.) We illustrate for the set  $N_{hp} = N_{2,45}$  where is the second element encountered by tracing the  $P(7)$  list at Level 2; i.e., where the first element is accessed by  $q = \text{First}(2, 7) = 67$  and the second is accessed by  $q = \text{After}(67) = 45$ . As noted in the listing of the  $\text{Loc}(q)$  vectors above,  $\text{Loc}(45) = (2, 18, 1)$ , giving  $h = 2, p = 18$  and  $r = 1$ .

Table A1 gives the information for  $h = 2$  and  $p = 18$  by:

$h$	$p$	$N_{hp}$	$Q_{hp}$
2	18	7, 8	45, 33

To determine the effect of flipping  $x_7$  we could proceed as earlier by making use of the  $\text{Find}_{hp}$  array in association with the  $N_{hp}$  array, noting that  $N_{hp} = N(2, 18) = (7, 8)$  and  $\text{Find}(2, 18) = (8, 7)$ , where:

$$N(1, 8) = (8); \quad N(1, 7) = (7)$$

The flip of  $x_7$  affects  $N_{hp}$  by uniquely generating the array  $N_{h^*p^*} = N(1, 8) = (8)$ . We know the coordinates  $h^* = 1$  and  $p^* = 8$  of  $N(1, 8)$  because  $h^* = h - 1$  and 8 is in the first position of  $\text{Find}(2, 18) = (8, 7)$  just as  $j = 7$  is in the first position of  $N(2, 18) = (7, 8)$ . We also know this position  $r$  because this is precisely the value given by  $r\text{Loc}(q)$ , which we found in the beginning by accessing  $q = 45$  and obtaining  $r\text{Loc}(45) = 1$ .

However, in this special case for Level 2, we can also obtain this same value  $p^* = 8$  (and hence also know  $N(1, p^*) = 8$ ) without referring to the  $\text{Find}_{hp}$  array for  $h = 2$ . [We also only need to access the value  $p = p\text{Loc}(q)$  and not the values  $h = h\text{Loc}(q)$  and  $r = r\text{Loc}(q)$ ] for the value  $q$  found on the  $P(7)$  list].

Specifically, we  $h = 2$  is already known because we use this  $h$  value to access  $\text{First}(2, 7)$  and then  $\text{After}(q)$  to obtain  $q = 45$ . From this we obtain  $p = p\text{Loc}(45) = 18$ , and access  $N(2, 18) = (7, 8)$ . Finally, for this special  $h = 2$  case we obtain  $p^*$  by the operation:

---

```

If  $N(h, p, 1) = j$  then
     $p^* = N(h, p, 2)$ 
Else
     $p^* = N(h, p, 1)$ 
Endif

```

---

This immediately yields  $p^* = 8$  without bothering to access or store the  $\text{Find}_{hp}$  vector for  $h = 2$ . Once  $p^* = 8$  is identified, the new costs for the two affected sets are given by setting  $c = \text{Cost}(h, p) = \text{Cost}(2, 18)$ , and then setting  $\text{Cost}(2, 18) = -c$  and  $\text{Cost}(1, 8) := \text{Cost}(1, 8) + c$ .

#### A.1.4 Exploiting the $Q_{hp}$ sets

We finally illustrate how the information provided by the  $Q_{hp}$  sets is used to handle the case when the cost  $\text{Cost}(h, p)$  of a particular term  $N_{hp}$  changes from zero to non-zero, or vice versa. Such a change means that reference to  $N_{hp}$  must be added or deleted from the various lists  $P(j)$  that access  $N_{hp}$ , to continue to exploit sparsity by only accessing terms with non-zero costs.

The  $Q_{hp}$  array, represented by  $Q_{hp} = (q_1, \dots, q_h)$ , facilitates these operations by recording the 'q index'  $q_r$  where the set  $N_{hp}$  is accessed by a trace of the list  $P(i_r)$ , where  $i_r$  is the ' $r^{\text{th}}$  element' of  $N_{hr} = (i_1, \dots, i_r, \dots, i_h)$ . In reality, when examining a given level  $h$ , a cost can change from zero to non-zero, or back, only at level  $h^* = h - 1$ . Hence, we are concerned with arrays of the form  $Q_{h^*p^*}$ , where  $p^*$  is derived from the pair  $(h, p)$  as in the preceding illustrations.

We continue to assume  $d = 3$ , which implies the largest value  $h^* = h - 1$  is limited to 2, and hence the  $Q_{hp}$  values in Table A1 for  $h = 3$  are irrelevant. (Their inclusion is useful for giving information that verifies the contents of the  $\text{Loc}(q)$  vectors in our example, but the  $Q_{hp}$  arrays in general need not be generated for  $h = d$ .)

We extend the illustrations of Section A2.3, which concerns the effects of flipping  $x_7$ . We first consider these effects for the level 3 set  $N_{hp} = N_{3,23}$ . As noted at the conclusion of the illustration for  $h = 3$ , new costs  $\text{Cost}(h, p)$  and  $\text{Cost}(h^*, p^*)$  are obtained for the two affected terms  $N_{hp}$  and  $N_{h^*p^*}$ , for  $(h, p) = (3, 23)$  and  $(h^*, p^*) = (2, 34)$ , by setting  $c = \text{Cost}(h, p) = \text{Cost}(3, 23)$ , and then setting  $\text{Cost}(3, 23) = -c$  and  $\text{Cost}(2, 34) := \text{Cost}(2, 34) + c$ . We know  $c \neq 0$  because  $N_{hp}$  was accessed by tracing non-zero cost terms. We are thus concerned with whether  $\text{Cost}(h^*, p^*) = \text{Cost}(2, 34)$  starts or ends with a 0 value.

#### *Cost(h\*, p\*) starts at 0*

The term  $N_{h^*p^*}$  is not on any of the  $P(i)$  lists and hence must be added to the lists that are relevant. Here,  $N_{h^*p^*} = N_{2,34} = (i_1, i_2) = (4, 13)$ . Consequently, we need to find add  $(h^*, p^*) = (2, 34)$  to the lists  $P(i_1) = P(4)$  and  $P(i_2) = P(13)$ . We let  $r^*$  range from 1 to 2, to access  $i_{r^*} = i_1$  and  $i_2$ , and in each case, pick up a new  $q$  value,  $q = q^*$ , and then put  $q^*$  on the front of the list for  $P(i_{r^*})$ . Moreover, we simultaneously create the associated array  $Q_{h^*p^*}$ . A fuller understanding of this array will be provided upon examining the case where  $\text{Cost}(h^*, p^*)$  starts non-zero but becomes zero.

---

```

For r* = 1, 2
  /* Get a new q index */
  q* = PoolQ(NQ)
  NQ := NQ - 1
  /* Add q* to P(i) for i in N_{h^*p^*} */
  i = 4 if r* = 1 and i = 13 if r* = 2
  /* i = N(h*, p*, r*) */
  qFirst = First(i)
  After(q*) = qFirst
  Before(qFirst) = q*
  First(i) = q*
  hLoc(q*) = 2 (= h*)
  pLoc(q*) = 34 (= p*)
  rLoc(q*) = r*
  /* Create entry r* of the Q(h*, p*, r*) array */
  Q(h*, p*, r*) = q*
Endfor r*
Cost(2, 34) (= Cost(h*, p*)) = c

```

---

$Cost(h^*, p^*)$  starts non-zero and becomes 0.

In this case, the result of updating  $Cost(2, 34) := Cost(2, 34) + c$  yields  $Cost(2, 34) = 0$ . Consequently  $(h^*, p^*) = (2, 34)$  must be dropped from all relevant  $P(i)$  lists. As before,  $N_{2,34} = (i_1, i_2) = (4, 13)$ , and now we must find  $(h^*, p^*)$  on the lists  $P(i_1) = P(4)$  and  $P(i_2) = P(13)$  in order to drop them, and this is where the list  $Q_{h^*, p^*}$  enters into the picture.

Table A1 does not show  $Q_{2,34}$  because it only contains information related to terms  $N_{hp}$  that lie on the list for  $P(7)$ , and at present we are dealing with a term  $N_{h^*, p^*} = (4, 13)$  that does not contain the index  $j = 7$ , and hence does not lie on  $P(7)$ . Since  $N_{h^*, p^*}$  exists (has a non-zero cost before making changes), the associated  $Q_{h^*, p^*}$  also exists and we suppose for concreteness  $Q_{2,34} = (q_1, q_2) = (62, 29)$ , meaning that  $(h^*, p^*) = (2, 34)$  is accessed on the list for  $P(i_1) = P(4)$  by  $q_1 = 62$ , and is accessed on the list for  $P(i_2) = P(13)$  by  $q_2 = 29$ . For this, we just add  $(h^*, p^*)$  to the first of these two lists by the operation of picking up a new  $q$  value,  $q = q^* \dots$

---

```

Cost(2,34) := Cost(2,34) + c
If Cost(2,34) = 0 (|Cost(2,34)| < ε) then
  For r* = 1 to 2
    i = 4 if r* = 1 and i = 13 if r* = 2
    /* i = N(h*,p*,r*) */
    q* = 62 if r* = 1 and q* = 29 if r* = 2
    /* q* = Q(h*,p*,r*) */
    If First(i) = q* then
      First(i) = After(q*)
    Else
      qB = Before(q*)
      qA = After(q*)
      Before(qA) = qB
      After(qB) = qA
    Endif
    /* Put q* back on PoolQ */
    NQ := NQ + 1
    PoolQ(NQ) = q*
  EndFor r*
Endif

```

---

### Special case for level 2

Extending the previous illustration for Level 2, we consider the situation where  $N_{hp} = N_{2,45}$ ; i.e.,  $(h, p) = (2, 45)$  and we obtained  $(h^*, p^*) = (1, 8)$ . By starting at Level 2, we now deal with  $h^* = 1$  at Level 1. Since by convention we maintain  $N(1, i) = i$  for all Level 1 sets, and access the Level 1 sets without concern for whether they have zero or non-zero costs, there is no need to use the machinery illustrated for  $h > 2$  (and hence  $h^* > 1$ ). For this same reason, no  $Q_{hp}$  sets are recorded when  $h = 1$ , as illustrated in Table A1.

**Appendix 2***Special implications for the cubic and quadratic cases*

First we note that the decoding process can be accelerated when  $h = 2$ . In this instance, the fact that  $N_p = \{i_1, i_2\}$  allows the identity of  $i_1$  and  $i_2$  to be determined from  $p = \text{Loc}(q)$  by the following simplified version of the decoding method of Section 5.

*Simplified decoding algorithm for  $|N_p| = 2$* 


---

```

v = p - n
If v = 1 then
    i2 = 2
Else
    w = (v - 1)2
    u = [w1/2]
    i* = u + 1
    Π0 = u
    Π1 = (i* - 2) Π0
    Π2 = i* Π0
    If Π2 ≤ w then
        Π = Π2
        i2 = i* + 1
    Else
        Π = Π1
        ir = i*
    Endif
    v := v - Π/2
Endif
i1 = v

```

---

The preceding specialised algorithm can also be embedded in a specialised algorithm for the case where  $|N_p| = 3$ , and thus a general decoding algorithm that applies to cubic polynomials (with degree  $d = 3$ ) can be made more efficient by the device of including this specialisation.

Additional specialisations for both the cubic and quadratic cases arise in reference to the arrays  $r\text{Loc}(q)$  and  $\text{Find}(h, p, r)$  in Section 6.3.2.1, where these arrays are used to identify  $p^* = \text{Find}(h, p, r)$  for  $r = r\text{Loc}(q)$ . When  $h = 3$ , the index  $p^*$  can be found fairly quickly even without storing  $r\text{Loc}(q)$ , by the following simplified search:

---

```

If j = N(h,p,1) then
  p* = Find(h,p,1)
Elseif j = N(h,p,2) then
  p* = Find(h,p,2)
Else
  p* = Find(h,p,3)
Endif

```

---

Finally, Find(h, p, r) need not be stored or accessed for h = 2, since in this case p\* = Find(h, p, r) would result in identifying p\* = i where either N(h,p,1) = {i, j} or {j, i}. Hence, for h = 2 we can find p\* directly by:

---

```

If j = N(h,p,1) then
  p* = N(h,p,2)
Else
  p* = N(h,p,1)
Endif

```

---

Further simplification for h = 2 occurs by not bothering to refer to h in N(h, p, r), i.e., storing N(2, p, r) as just N(p, r). Then the preceding check is:

---

```

If j = N(p,1) then
  p* = N(p,2)
Else
  p* = N(p,1)
Endif

```

---

These simplifications can be introduced by checking for the value of h in a polynomial of any degree d, but can be incorporated directly for the cases where d = 2 and 3 to provide more economical alternative.

### *Quadratic problems*

The analysis based on implications of Observation 2 also discloses that the quadratic case occupies a special position that permits sparsity to be exploited with particular efficiency and with simplified data structures when d = 2. For quadratic polynomials,  $z_0(2)$  remains unchanged at the value  $nz_0(2)$ , the number of non-zero quadratic terms in the initial input data. The form of P(i) simplifies to  $P(i) = \{p \in P: i = i_1 \text{ or } i_2 \text{ for } N_p = \{i_1, i_2\}, i_1, i_2 \in N\}$ , understanding that i itself implicitly belongs to P(i). Further, no set  $N_p$  of the form  $N_p = \{i_1, i_2\}$  will be added or dropped as a result of making any number of flip moves when d = 2, and consequently the initial q lists Loc(q), Before(q) and After(q) which are created when the problem data is read in will remain invariant. In turn, this means that there is no need to maintain the list PoolQ, since no q indexes will be added to or dropped from this list [and PoolQ implicitly is an unchanging list  $\{1, \dots, NQ\}$  where each entry q of PoolQ always identifies the same set  $N_p$  identified by  $p = pLoc(q)$ ]. Under this circumstance, the After(q) list is also irrelevant, as is the list  $Q_p$ , since the only function of this latter list is to provide a means to drop elements from PoolQ efficiently. In sum,

the full set of updating operations for the quadratic case simplifies to precisely the following.

*Initial generation of the elements  $p \in P(i)$  during data input*

---

```

Set First(i) = 0 for all  $i \in N$ . After(0) = 0.
 $c_p = 0$  for  $p = 1$  to  $n(d)$ 
 $q = 0$ 
While problem data remain to be input
  Read in next data element ( $h, M, c$ ), where  $M = \{i_1, \dots, i_h\}$  and  $c = c(i_1, \dots, i_h)$  for  $h = 1$  or
  2. Assume  $i_1 < i_2$  (if  $h = 2$ )
  Code the set  $M = \{i_1, \dots, i_h\}$  to obtain  $p = V[M]$  and set  $c_p = c$ .
  If  $h = 2$  then
    For  $i = i_1$  and  $i = i_2$ 
       $q := q + 1$ 
      After(q) = First(i)
      First(i) = q
       $p = \text{Loc}(q)$ 
    EndFor i
  Endif
EndWhile

```

---

Then the elements  $p \in P(i)$  are identified as indicated earlier.

*Accessing the elements  $p \in P(i)$*

---

```

 $q = \text{First}(i)$ 
While  $q > 0$ 
   $p = \text{pLoc}(q)$  /* identifies the current element  $p \in P(i)$  */
  /* Access  $N_{2p} = \{i_1, i_2\}$  */
   $q = \text{After}(q)$ 
EndWhile

```

---

Again it is understood that the index  $i$  itself is to be included among these indexes accessed, though we do not bother to store it in  $P(i)$ . These enhancements for quadratic problems can be used to improve existing quadratic unconstrained binary optimisation algorithms based on flip moves when applied to problems with sparse data.