THE UNIVERSITY OF TEXAS AT SAN ANTONIO, COLLEGE OF BUSINESS

Working Paper SERIES

February 15, 2007

WP # 0010MSS-061-2007

A Primogenitary Linked Quad Tree Approach for Solution Storage and Retrieval in Heuristic Binary Optimization

Minghe Sun

Department of Management Science and Statistics
College of Business
The University of Texas at San Antonio
San Antonio, Texas 78249

(210)458-5777 (phone) (210)458-6350 (fax)
minghe.sun@utsa.edu

http://faculty.business.utsa.edu/msun

A Primogenitary Linked Quad Tree Approach for Solution Storage and Retrieval in Heuristic Binary Optimization

Minghe Sun

Department of Management Science and Statistics
College of Business
The University of Texas at San Antonio
San Antonio, TX 78249

(210) 458-5777 (phone) (210) 458-6350 (fax)
minghe.sun@utsa.edu
http://faculty.business.utsa.edu/msun

A Primogenitary Linked Quad Tree Approach for Solution Storage and Retrieval in Heuristic Binary Optimization

Abstract

A data structure, called the primogenitary linked quad tree (PLQT), is used to store and retrieve solutions in heuristic solution procedures for binary optimization problems. Solutions represented by vectors of binary variables are encoded into vectors of integers in a much lower dimension. The vectors of integers are used as composite keys to store and retrieve solutions in the PLQT. An algorithm processing trial solutions for possible insertion into or retrieval from the PLQT is developed. Examples are provided to demonstrate the way the algorithm works. Another algorithm traversing the PLQT is also developed. Computational results show that the PLQT approach takes only a very tiny portion of the CPU time taken by a linear list approach for the same purpose for any reasonable application. The CPU time taken by the PLQT managing the solutions is negligible as compared to that taken by a heuristic procedure for any reasonably hard to solve binary optimization problems, as shown in a tabu search heuristic procedure for the capacitated facility location problem. Compared to the hashing approach, the PLQT approach takes about the same amount of CPU time but much less memory space while completely eliminating collision.

Keywords: Primogenitary Quad Tree, Data Structure, Heuristic Procedures, Binary Optimization, Combinatorial Optimization.

A Primogenitary Linked Quad Tree Approach for Solution Storage and Retrieval in Heuristic Binary Optimization

Many combinatorial optimization problems are NP hard and, therefore, are very difficult to solve. Exact algorithms can solve small problems and heuristic procedures are usually employed for large problems. Researchers have developed many metaheuristic methods, such as simulated annealing [Kirkpatrick, Gelatt and Vecchi, 1983], tabu search [Glover, 1989, 1990a, 1990b; Glover and Laguna, 1997], scatter search (Glover, Laguna and Martí, 2000), genetic algorithms [Holland, 1992], and ant colony optimization (Deneubourg, 1983; Deneubourg and Goss, 1989), for hard combinatorial optimization problems. These methods provide frameworks or guidelines in forming a strategy for solving a problem. To solve a specific hard combinatorial optimization problem, a metaheuristic method has to be tailored to form a specific heuristic procedure to take advantage of the problem structure. Many heuristic procedures using these metaheuristic methods have been developed for many hard combinatorial optimization problems. For problems with realistic sizes, heuristic procedures usually take a very long computation time to find good, but not necessarily optimal, solutions.

The focus of this study is to develop a data structure approach to store and retrieve trial solutions for binary optimization problems, a specific type of combinatorial optimization problems, in heuristic procedures. The data structure is called the primogenitary linked quad tree (PLQT), a quad tree with special structures. This approach is so efficient that the computation time it takes is unnoticeable as compared to that a heuristic procedure takes. Therefore, it is a handy tool for researchers to use in developing their heuristic procedures for binary optimization problems.

In Section 1, the binary optimization problem and the necessity for trial solution storage and retrieval are discussed. In Section 2, alternative ways of representing solutions are discussed and integer vectors are proposed to encode solutions of binary optimization problems. The PLQT and algorithms managing it are described in Section 3. Examples demonstrating the insertion of integer vectors into and the retrieval of integer vectors from a PLQT are given in Section 4. Computational results are presented in Section 5. Finally concluding remarks and summaries are given in Section 6.

1. Introduction

The most common combinatorial optimization problems are binary optimization problems. A binary optimization problem may be considered to involve a set of objects and a subset of the objects meeting certain restrictions needs to be selected based on one or more criteria or objectives.

In a project or investment portfolio selection problem [Stummer and Sun, 2005], for example, the objects are the projects or investment instruments. A portfolio is a selected subset. The major objective is the maximization of the total expected return on investment although there are other objectives in a multiple criteria problem [Stummer and Sun, 2005]. The restrictions include the limited budgets, diversification requirements and balancing requirements among others.

In a facility location problem [Delmaire, Diaz, Fernandez and Ortega, 1998; Ducati, Armentano, and Sun, 2004; Sun, 2005, 2006a], as another example, the objects are the potential sites to locate facilities. A subset of these sites is selected to actually have the facilities established. Each site has a fixed cost to establish and to operate the facility and has fixed costs or variable costs to serve the clients, such as transporting the products to the customers.

The major objective is to minimize the total costs. The restrictions are to meet the client demands possibly within the capacities of the selected facilities.

Common to all binary optimization problems is that the status of an object i can be represented by a binary variable y_i, *i.e.*,

$$y_i = \begin{cases} 0, & \text{if object } i \text{ is not selected} \\ 1, & \text{if object } i \text{ is selected.} \end{cases} \quad (1)$$

Any selected subset, *i.e.*, a solution, for a problem with n objects can be represented by a binary vector with n elements $\mathbf{y} = (y_0, y_1, \ldots, y_i, \ldots, y_{n-2}, y_{n-1})$. Each combination of the n binary variables is a possible solution. Hence, a problem with n binary variables has 2^n possible solutions. However, usually a small portion of the 2^n possible solutions is feasible, *i.e.*, satisfying all restrictions. In a heuristic procedure, usually a very small portion of the 2^n possible solutions is evaluated. In addition to binary variables, most problems also involve real, *i.e.*, continuous, variables.

Usually a binary optimization problem can be formulated as a binary mathematical programming model [Nemhauser and Wosley, 1988; Wosley, 1988] where a restriction is represented by a constraint and a criterion is represented by an objective function. A binary minimization problem with one objective function may be written as

$$\min \quad f(\mathbf{x}, \mathbf{y}) \quad (2)$$

$$\text{s.t.} \quad g_i(\mathbf{x}, \mathbf{y}) \geq 0 \qquad \text{for } i = 1, \cdots, \eta \quad (3)$$

$$\mathbf{x} \in \Re^{n'}, \ \mathbf{y} \in B^n. \quad (4)$$

Where $\mathbf{x}$ represents the vector of real variables and B^n is the collection of all ordered n-tuples of 0s and 1s. The model is a pure binary programming model if $n' = 0$ and is a mixed binary programming model otherwise. In this study, $n > 0$ is assumed.

Each $\mathbf{y} \in B^n$ determines a trial solution. Once $\mathbf{y}$ is determined, the trial solution can be evaluated to determine $f(\mathbf{x}, \mathbf{y})$. Evaluating a trial solution is usually very time consuming in the solution process. In a portfolio selection problem, for example, a linear programming problem needs to be solved [Stummer and Sun, 2005]; in a capacitated facility location problem [Ducati, Armentano, and Sun, 2004], a transportation problem needs to be solved; and in a single source facility location problem [Delmaire, Diaz, Fernandez and Ortega, 1998], a generalized assignment problem, which is itself a binary optimization problem, needs to be solved, to evaluate a solution. Therefore, each solution needs to be evaluated only once in an efficient heuristic solution procedure. Once evaluated, the solution may be saved or stored and may be retrieved later if needed. In the solution process using a heuristic procedure, many solutions are evaluated.

When such a problem is solved with a heuristic procedure, the solution process usually follows a selective trajectory in the solution space. At a visited solution on the trajectory, a neighborhood is created by evaluating or retrieving one or more trial solutions. A trial solution may be obtained by changing the value of one binary variable, called a simple move, or by changing one binary variable from 0 to 1 and another from 1 to 0, called an exchange or swap move. For each trial solution, the saved solutions may be searched to find out if it has already been evaluated and saved. If found, the trial solution is retrieved and does not need to be evaluated again; otherwise, the trial solution is evaluated and saved. Based on some predefined rules, the procedure selects one of

these evaluated trial solutions as the next visited solution to move to. A move is the transition from the current solution to one in the neighborhood. Once a solution is visited, it does not need to be visited again. If a solution is visited the second time, the same trajectory may be followed again if the predefined rules do not change and, hence, repetition or cycling may occur. Therefore, measures are usually taken by heuristic procedures to prohibit the visit of solutions which have been visited already. For this purpose, the visited solutions need to be memorized in some way.

Hence, there are two purposes for storing the evaluated solutions. One purpose is to save computation time. Once a trial solution is found to be evaluated already, it does not need to be evaluated again and only needs to be retrieved. The other purpose is to prevent repetition or cycling. Once a solution is found to be visited, it may not be selected to visit again.

The PLQT approach developed in this study can be used by any heuristic procedure for any binary optimization problem. The PLQT is a new data structure recently developed by Sun [2006b] for fast access of data, or data with composite keys, in $\Re^K$. It is an enhancement of the more traditional quad tree data structure [Finkel and Bentley, 1974; Habenicht, 1982, 1991; Sun and Steuer, 1996a, 1996b]. Compared to the traditional quad tree, the PLQT uses substantially less computation time and takes considerably less memory and storage space [Sun, 2006b]. Among others, quad trees have been employed in discrete multiple criteria optimization, geometric information systems, image processing, and computer aided design and computer aided manufacturing. Sun [2006b] showed through computational experiment that the PLQT is much faster than the traditional quad tree, which is in turn much faster than the linear list, in identifying, storing and retrieving nondominated solutions in discrete multiple criteria optimization. The PLQT is even much faster in storing and retrieving trial solutions in heuristic procedures because the PLQT does not need to be reconstructed as in the application reported by Sun [2006b].

2. Integer Representation of Binary Solutions

Let b represent the number of bits used to represent an integer. Most computer languages use 2 or 4 bytes of memory to represent an integer. If 2 bytes are used, $b = 16$, and if 4 bytes are used, $b = 32$. Both positive and negative integers that can be represented by b bits can be used for this application. However, for easy description only unsigned integers are used in the following discussion. Therefore, the range of integers a computer can represent is between 0 and $2^b - 1$ and the maximum number of different integers a computer can represent is 2^b.

2.1 Integer Coding of Binary Solutions

The binary vector $\mathbf{y} = (y_0, y_1, \ldots, y_i, \ldots, y_{n-2}, y_{n-1})$ representing a trial solution can also be written as a binary number with n digits, *i.e.*, $(y_{n-1}\ y_{n-2} \ldots y_i \ldots y_1\ y_0)$. A decimal integer z equivalent to this binary number is determined by

$$z = \sum_{i=0}^{n-1} 2^i y_i \,. \tag{5}$$

The decimal integer z is unique for each binary vector $\mathbf{y} = (y_0, y_1, \ldots, y_i, \ldots, y_{n-2}, y_{n-1})$ and ranges from 0 to $2^n - 1$. The 2^n possible solutions of a binary optimization problem with n binary variables can be naturally ordered from 0 to $2^n - 1$. Therefore, the decimal integer z can be used naturally as the index to store the evaluated

solutions in an array with 2^n elements, *i.e.*, the solution with an index z is stored at position z. There are two difficulties with this approach. One is representability because managing the trial solutions this way becomes impossible when $n > b$. Unfortunately, most hard to solve problems have $n > b$. When $n \le b$, the binary optimization problem is relatively easy to solve and sophisticated heuristic procedures may not be needed. The other difficulty is the waste of memory space. Because only a small portion of the 2^n possible solutions is feasible and only a small portion of the feasible solutions is evaluated in the solution process, it is very inefficient to store these solutions in an array with 2^n elements and the memory spaces for the infeasible solutions and for solutions not evaluated are wasted.

Naturally the evaluated solutions may be stored in two different ways if an array or a list with fewer than 2^n elements is used. One way is to arrange them in the order of increasing value of z. However, storing them in this order is a very time consuming process. Another way is to store the evaluated solutions in the order that they are found. The problem with this approach is that searching the array for a specific solution is a very time consuming process. Managing these solutions using these approaches takes most, *e.g.*, up to 99% [Stummer and Sun, 2005], of the computation time in some heuristic solution procedures.

2.2 Hashing

Researchers have proposed and used hashing for storage and retrieval of evaluated solutions in heuristic procedures [Woodruff and Zemel, 1993; Carlton and Barnes, 1996; Klein, 2000; Ducati, Armentano, and Sun, 2004]. Woodruff and Zemel [1993] proposed four hash functions for different neighborhood structures used in tabu search. The original purpose was to use the saved solutions to implement tabu conditions. The first two hash functions are given in (6) and (8) in the following.

The first hash function is

$$h = \sum_{i=0}^{n-1} w_i y_i \,, \tag{6}$$

where w_i for $i = 0, \cdots, n-1$ are a set of randomly generated integers. Then h determines the position where the corresponding solution (or its objective function value) is stored in a 2^b array. When y_i changes its value in a trial solution, h can be easily updated from its current value. For binary optimization, h in (6) can be updated using (7) in the following,

$$h \leftarrow \begin{cases} h - w_i & \text{if } y_i \text{ changes from 1 to 0} \\ h + w_i & \text{if } y_i \text{ changes from 0 to 1.} \end{cases} \tag{7}$$

The second hash function is

$$h = \left(\sum_{i=0}^{n-1} w_i y_i \right) \bmod \Phi \,. \tag{8}$$

Then h is used as the index to store the corresponding solution in an array with Φ elements. The two hash functions in (6) and (8) are in fact the same when $\Phi = 2^b$ as originally proposed. However, the one in (8) explicitly handles memory overflow, *i.e.*, what to do when $\sum_{i=0}^{n-1} w_i y_i \ge \Phi$. In (6), $\Phi = 2^b$ is assumed and the "mod" operation is performed through memory overflow. When $\Phi < 2^b$, *e.g.*, as used by Klein [2000], an array with fewer than 2^b elements is needed to store the data. In this case, a "mod" operation, which is much more time

consuming than integer addition or subtraction, has to be performed. When $\Phi < 2^b$, h in (8) cannot be updated using (7). Therefore, the computation of h in (8) is much more time consuming than that in (6).

Hashing somewhat overcomes the two difficulties mentioned above but has its own disadvantages [Carlton and Barnes, 1996]. The major disadvantage is collision which imposes another difficulty. Collision occurs when two or more solutions have the same hash value h. As Carlton and Barnes [1996] pointed out, collision may cause problems in the heuristic procedure if extra care is not taken in the implementation. To avoid excessive rate of collision, an array much larger than that actually needed must be used. Therefore, the difficulty of memory waste is not completely overcome. Auxiliary data structures may have to be used to handle collisions, as discussed in any data structures textbooks. Because of collision, when the position that a new solution hashed to is occupied, a comparison has to be made to check if the saved solution and the new solution are the same solution or not. Therefore, extra computation time may be needed.

If m solutions are to be stored in an array with Φ elements, assuming h is uniformly distributed between 0 and $\Phi - 1$, the probability of collision, denoted by $P(\text{collision})$, is given by

$$P(\text{collision}) = 1 - \frac{\Phi!}{(\Phi - m)!\Phi^m}. \tag{9}$$

In (9), $\Phi!/(\Phi - m)!$ is the number of ways to put each of the m solutions in a unique position, Φ^m is the number of total ways to put the m solutions in Φ positions, and $\Phi!\Big/\left[(\Phi - m)!\Phi^m\right]$ is the probability of no collision. Carlton and Barnes [1996] listed $P(\text{collision})$ for different values of m and Φ. Even with $\Phi = 2^b$, $P(\text{collision})$ becomes pretty large even for reasonably small values of m. If h is not approximately uniformly distributed between 0 and $\Phi - 1$, $P(\text{collision})$ may be higher [Carlton and Barnes, 1996]. Very likely, h is not appoximately uniformly distributed if w_i for $i = 0, \cdots, n-1$ are not carefully chosen. Unfortunately, choosing a good set of w_i may not be an easy task.

Woodruff and Zemel [1993] listed the following three goals for a hash function. (1) Computation and update of h should be as easy as possible. This means that the structure of h should reflect the structure of the neighborhood sets. (2) The integer generated should be in a range that results in a reasonable storage requirement and comparison effort (*e.g.*, an integer requiring two or four bytes). (3) The probability of collision should be low. A collision occurs when two different vectors are encountered with the same hash function value. These goals may also be used as criteria to measure other solution storage and retrieval approaches.

2.3 Integer Vector Representation of Trial Solutions

In this study, an integer vector $\mathbf{z} \in Z^K$ with $K = \lceil n/b \rceil$ is used to encode a trial solution. The notation $\lceil x \rceil$ represents the smallest integer greater than x and Z^K is the collection of all K-tuples of integers. In the following, $\mathbf{z} \in Z^K$ is used to represent a generic vector, and $\mathbf{z}^i \in Z^K$, $\mathbf{z}^{m'} \in Z^K$ and so on are used to represent specific vectors in Z^K.

The binary vector $\mathbf{y} = (y_0, y_1, \ldots, y_i, \ldots, y_{n-2}, y_{n-1})$ is divided into K sections with b elements in each section except for the last section. For convenience, the left most section is designated the first section, *i.e.*, section 0, and the right most section is designated the last, *i.e.*, section $K-1$. The elements in section k are

(y_{kb} , y_{kb+1} ,…, $y_{kb+i'}$,…, $y_{(k+1)b-2}$, $y_{(k+1)b-1}$) for $k = 0,\cdots, K-2$, and (y_{kb} , y_{kb+1} ,…, $y_{kb+i'}$,…, y_{n-2} , y_{n-1}) for $k = K-1$. For example, for a problem with $n = 200$ binary variables when $b = 32$, the integer vector $\mathbf{z}$ has $K = \lceil n/b \rceil = \lceil 200/32 \rceil = 7$ elements. The elements in section 0 are (y_0 , y_1 ,…, y_{31}), those in section 1 are (y_{32} , y_{33} ,…, y_{63}), and those in section 6 are (y_{192} , y_{193} ,…, y_{199}).

As z in (5), an integer z_k for section k is defined as

$$z_k = \sum_{i=kb}^{u_k} 2^{i-kb} y_i \ . \tag{10}$$

where $u_k = (k+1)b-1$ for $k = 0,\cdots, K-2$ and $u_{K-1} = n-1$. There is a one to one correspondence between $\mathbf{y} \in B^n$ and $\mathbf{z} \in Z^K$ but K is much smaller than n . Then $\mathbf{z} = (z_0, z_1, \cdots z_{K-1})$ is used as the composite key to store and retrieve the trial solution represented by $\mathbf{y}$.

Although it is very efficient to compute the vector $\mathbf{z}$ by using the definition in (10), the components of $\mathbf{z}$ need to be computed using (10) only for the first trial solution while those for subsequent trial solutions only need to be updated from their current values. Suppose the binary variable y_i is going to change its value in the next trial solution in a simple move. The section k' where y_i is located in the vector $\mathbf{y}$ is determined by

$$k' = \lfloor i/b \rfloor . \tag{11}$$

The notation $\lfloor x \rfloor$ represents the largest integer smaller than x . Then only the value of $z_{k'}$ will be changed from its current value and the values of z_k will stay the same for all $k \neq k'$. Let

$$i' = i - k'b . \tag{12}$$

Then i' is in the position where y_i is in section k' . The value of $z_{k'}$ is then updated using (13) in the following

$$z_{k'} \leftarrow \begin{cases} z_{k'} - 2^{i'}, & \text{if } y_i \text{ changes from 1 to 0} \\ z_{k'} + 2^{i'}, & \text{if } y_i \text{ changes from 0 to 1.} \end{cases} \tag{13}$$

In the implementation, the values of $2^{i'}$ for $0 \le i' \le b-1$ are computed only once and stored in an array with b elements. Hence, they do not need to be computed each time when needed in (10) or (13). In fact in some computer languages, they may not need to be computed and stored but can be obtained through the "bitwise shift" operation when needed because only the i' th bit is on and all others are off in $2^{i'}$ for any $0 \le i' \le b-1$. In some computer languages, $2^{i'}$ for $0 \le i' \le b-1$ can also be treated as masks and (10) and (13) can be implemented through bitwise manipulations. Because (13) just simply toggles one bit of $z_{k'}$ on or off, it can be implemented through the "bitwise exclusive or" operation. The values of k' and i' may be tracked and may not have to be computed through (11) and (12) in the implementation. More complicated, such as swap or exchange, moves can be decomposed into multiple simple moves and (11)-(13) can be used multiple times to update the relevant elements of $\mathbf{z}$.

The difficulty of representability mentioned above is completely overcome with the integer vector representation. The difficulty of memory waste is also resolved when the evaluated solutions are stored in a specific order. Because of the one to one correspondence between $\mathbf{y} \in B^n$ and $\mathbf{z} \in Z^K$, the difficulty caused by collision is completely eliminated.

2.4 An Example

Assume there are $n = 150$ binary variables in a binary optimization problem. Also assume 4 bytes are used to represent an integer, *i.e.*, $b = 32$. Then an integer vector of $K = \lceil n/b \rceil = \lceil 150/32 \rceil = 5$ elements is needed to represent one solution. In the current solution, assume

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

i.e., the value of every fifth binary variable starting from y_0 is 1 and all others are 0. Using (10), the current solution is encoded as $\mathbf{z} = (z_0, z_1, z_2, z_3, z_4)$ = (1108378657, 277094664, 2216757314, 554189328, 135300). Suppose in the next trial solution to be checked, y_{49} is going to change from 0 to 1. According to (11), $k' = \lfloor i/b \rfloor = \lfloor 49/32 \rfloor = 1$, and according to (12), $i' = i - k'b = 49 - 32 = 17$. Hence, only z_1 needs to be updated from its current value. The new value of z_1 will become 277094664+131072=277225736 according to (13). Suppose in the following trial solution to be checked, y_{100} is going to change from 1 to 0. According to (11), $k' = \lfloor i/b \rfloor = \lfloor 100/32 \rfloor = 3$, and according to (12), $i' = i - k'b = 100 - 3(32) = 4$. As a result, only z_3 needs to be updated. The new value of z_3 will become 554189328–16 = 554189316 according to (13). If both y_{49} and y_{100} are changed in the next trial solution in an exchange move, then both z_1 and z_3 are updated at the same time.

2.5 The Linear List Approach

A natural approach to manage the evaluated solutions encoded with integer vectors is to keep them in a list or an array. In addition to the integer vector $\mathbf{z}$, other relevant information may also be saved for each evaluated solution. Suppose m' evaluated solutions represented by the integer vectors $\mathbf{z}^1, \cdots, \mathbf{z}^i, \cdots, \mathbf{z}^{m'}$ have been added to the list already in this order. When a new vector $\mathbf{z}$ is processed for addition to or retrieval from the list, it is compared with each $\mathbf{z}^i$ component by component starting from $\mathbf{z}^1$. Such a comparison is called a pairwise vector comparison. Once any corresponding components of $\mathbf{z}$ and $\mathbf{z}^i$ are found to be different, *i.e.*, $z_k \neq z_k^i$ for any $0 \le k \le K-1$, this pairwise vector comparison between $\mathbf{z}$ and $\mathbf{z}^i$ will stop. The rest of the components do not need to be compared. Therefore, most of the pairwise vector comparisons in this approach are partial comparisons. If $z_k = z_k^i$ for all $0 \le k \le K-1$ for any i, then $\mathbf{z}^i = \mathbf{z}$. In this case, the process stops after $\mathbf{z}^i$ is retrieved. If $\mathbf{z}$ has been compared with all $\mathbf{z}^i$ for $1 \le i \le m'$ but none of them can be retrieved, $\mathbf{z}$ is then added to the end of the list as $\mathbf{z}^{m'+1}$. This approach is called a linear list approach and will be used as a benchmark to measure the performance of the PLQT approach in the computational experiment. The following is the algorithm of the linear list approach.

Routine linearlist(m' , $\mathbf{z}$)

Step 1 Let $i = 1$.

Step 2 Let $k = 0$.

Step 3 If $z_k \neq z_k^i$, go to Step 6.

Step 4 Let $k \leftarrow k+1$. If $k \le K-1$, go to Step 3.

Step 5 Return $\mathbf{z}^i$.

Step 6 let $i \leftarrow i+1$. If $i \le m'$, go to Step 2.

Step 7 Let $\mathbf{z}^{m'+1} = \mathbf{z}$ and $m' \leftarrow m'+1$. Return.

3. The PLQT Data Structure and Algorithms

A PLQT is used to process, store and retrieve data, or data having composite keys, with hierarchical relationships in $\Re^K$ for $K \ge 1$. The data processed with a PLQT in this application are the integer vectors representing the trial solutions, *i.e.*, data in Z^K. Terminologies in genealogy used to describe the hierarchical relationship between a person and his or her ancestors, siblings and successors are usually borrowed to describe the hierarchical relationship among the elements of a PLQT. Sun [2006b] gave a detailed description of PLQTs. In this section, only the terminologies necessary for the application in this study are given and the algorithms managing the PLQT are then developed. In the following, $\mathbf{z} \in \Re^K$ is also used to represent a generic vector and $\mathbf{z}^r \in \Re^K$, $\mathbf{z}^s \in \Re^K$, $\mathbf{z}^t \in \Re^K$ and so on are used to represent specific vectors in $\Re^K$.

3.1 PLQT Structure and Terminology

A PLQT is a finite set of elements one of which, if any, is called the root and the rest are partitioned into 2^K disjoint sets each of which is itself a PLQT, called a subtree. Each element in a PLQT is a node that has two parts, a data part and an address part. The data part represents a record that may be a vector $\mathbf{z} \in \Re^K$ or may have a vector $\mathbf{z} \in \Re^K$ as its composite key. In the later case, the record has other data fields. In the application of this study, $\mathbf{z} \in Z^K$ computed with (10) or updated with (13) is the composite key. Other data fields may include the value of the objective function and possibly the values of other real variables, *i.e.*, $\mathbf{x} \in \Re^{n'}$, of the corresponding solution. Because the composite key uniquely identifies a data record contained in a node, $\mathbf{z}$ refers to the composite key, the record and the node of the PLQT. The address part represents the relationship of the node to other nodes in the hierarchy. Usually, $\mathcal{T}$ is used to represent a generic PLQT and $\mathcal{T}_{\mathbf{z}}$ is used to represent a PLQT with $\mathbf{z}$ as its root. Sometimes, $\mathbf{z} \in \mathcal{T}$ is used to indicate the fact that $\mathbf{z}$ is a node in $\mathcal{T}$.

If $\mathbf{z}^s \in \mathcal{T}_{\mathbf{z}^r}$ is the root of any of the subtrees of $\mathbf{z}^r$, $\mathbf{z}^r$ is called the parent of $\mathbf{z}^s$ and $\mathbf{z}^s$ is called a son of $\mathbf{z}^r$. For data in $\Re^K$, a node may have up to 2^K sons. The sons are ordered, or indexed, sequentially from 0 to $2^K - 1$. Some of the sons may not exist in $\mathcal{T}_{\mathbf{z}^r}$. A position in $\mathcal{T}_{\mathbf{z}^r}$ is available for each son whether it exists or not. If $\mathbf{z}^s$ is a son of $\mathbf{z}^r$ and is put at the ϕth position when the sons are ordered, $\mathbf{z}^s$ is called the $\phi-$ son of $\mathbf{z}^r$ and ϕ is called the successorship of $\mathbf{z}^s$ to $\mathbf{z}^r$. $\mathcal{S}(\mathbf{z})$ is used to denote the successorship of $\mathbf{z}$ to its parent, *i.e.*, $\mathbf{z}$ is the $\mathcal{S}(\mathbf{z})-$ son of its parent. The son of $\mathbf{z} \in \mathcal{T}$ with the smallest successorship among all existing sons of $\mathbf{z}$ is called the eldest (first-born) existing son of $\mathbf{z}$. $\mathcal{F}(\mathbf{z})$ is used to denote the eldest existing son of $\mathbf{z}$.

Each node except for the root has a unique parent. A node that does not have an existing son is a leaf node. If $\mathbf{z} \in \mathcal{T}$ is a leaf node, it does not have any successors; otherwise, each son of $\mathbf{z}$ and all the successors of each son of $\mathbf{z}$ are the successors of $\mathbf{z}$. If $\mathbf{z}^s$ is the $\phi-$ son of $\mathbf{z}^r \in \mathcal{T}$, $\mathbf{z}^s$ and all the successors of $\mathbf{z}^s$ are the $\phi-$ successors of $\mathbf{z}^r$. All the $\phi-$ successors of $\mathbf{z}^r$ form a subtree rooted at the $\phi-$ son of $\mathbf{z}^r$. The root of a PLQT is assigned a level 0. The level of each other node is 1 greater than that of its parent. The level of the PLQT is the

level of the node with the largest level in the PLQT. All the nodes at the same level form one level of the PLQT. The shape of a PLQT refers to the way it is filled. With the same number of nodes, the larger its level is, the more sparsely it is filled.

Nodes sharing the same parent are siblings. If $\mathbf{z}^s$ is the $\phi-$ son of $\mathbf{z}^r$ and $\mathbf{z}^t$ is the $\varphi-$ son of $\mathbf{z}^r$ with $\phi < \varphi$, then $\mathbf{z}^t$ is a younger sibling of $\mathbf{z}^s$. If there does not exist any other sibling between $\mathbf{z}^s$ and $\mathbf{z}^t$, then $\mathbf{z}^t$ is the immediate existing younger sibling of $\mathbf{z}^s$. $\mathcal{N}(\mathbf{z})$ is used to denote the immediate existing younger sibling of $\mathbf{z}$.

For notational convenience, $\mathcal{N}(\mathbf{z})$ and $\mathcal{A}(\mathbf{z})$ are defined as nodes in a PLQT. However, they are used as pointers in the implementation. Therefore, $\mathcal{N}(\mathbf{z})$ and $\mathcal{A}(\mathbf{z})$ refer to pointers and the nodes they point to in the following discussion. In the implementation, a node is represented by a structure, $\mathcal{N}(\mathbf{z})$ and $\mathcal{F}(\mathbf{z})$ are represented by pointers and $\mathcal{S}(\mathbf{z})$ is represented by an integer. For most applications, an integer with one byte or two bytes, *e.g.*, a "unsigned char" or "unsigned short" data type in C, is sufficient for $\mathcal{S}(\mathbf{z})$. Compositions of these notations may be used when convenient. For example, $\mathcal{S}(\mathcal{F}(\mathbf{z}))$ is used to represent the successorship of $\mathcal{F}(\mathbf{z})$ to $\mathbf{z}$ and $\mathcal{S}(\mathcal{N}(\mathbf{z}))$ is used to represent the successorship of $\mathcal{N}(\mathbf{z})$ to its parent.

The address part of a node $\mathbf{z} \in \mathcal{T}$ has three fields, $\mathcal{N}(\mathbf{z})$, $\mathcal{F}(\mathbf{z})$ and $\mathcal{S}(\mathbf{z})$. In the application studied by Sun [2006], another pointer pointing to the parent of the node is also used. However, in the current study, such a pointer is not necessary. A pointer points to NULL, *i.e.*, nowhere, if the node that it points to does not exist. From $\mathbf{z}$, $\mathcal{N}(\mathbf{z})$ and $\mathcal{F}(\mathbf{z})$ can be accessed directly. All other successors of $\mathbf{z}$ can be accessed from $\mathbf{z}$ only through $\mathcal{A}(\mathbf{z})$ and all other younger siblings of $\mathbf{z}$ as well as their successors can be accessed from $\mathbf{z}$ only through $\mathcal{N}(\mathbf{z})$. The siblings are connected and managed as a linked list.

3.2 Hierarchical Relationship among Nodes

Each evaluated solution is treated as a node in the PLQT. As discussed above, the integer vectors $\mathbf{z}$ determined through (10) or updated through (13) representing the trial solutions are the composite keys of the nodes in a PLQT. The way in which these integer vectors are organized in a PLQT is described in the following.

When two integer vectors $\mathbf{z}^r \in Z^K$ and $\mathbf{z}^t \in Z^K$ are compared component wise, a binary digit ϕ_k is defined as

$$\phi_k = \begin{cases} 0, & \text{if } z_k^t \geq z_k^r \\ 1, & \text{otherwise} \end{cases} \tag{14}$$

and an equivalent decimal integer ϕ is

$$\phi = \sum_{k=0}^{K-1} \phi_k 2^k \ . \tag{15}$$

Then ϕ is the successorship of $\mathbf{z}^t$ to $\mathbf{z}^r$, *i.e.*, $\mathbf{z}^t$ is a $\phi-$ successor of $\mathbf{z}^r$, and $\phi_{K-1}\phi_{K-2}...\phi_0$ is the binary equivalent of ϕ. With ϕ_k defined in (14) and ϕ defined in (15), $\mathbf{z}^t = \mathbf{z}^r$ is possible only if $\phi = 0$. Although there are other ways to define the successorship of $\mathbf{z}^t$ to $\mathbf{z}^r$, *i.e.*, to order the successors of $\mathbf{z}^r$, the way given in (14) and (15) is convenient to implement and efficient to execute.

When a pairwise vector comparison is made in the PLQT approach, all corresponding components of the two vectors must be compared and the integer ϕ is computed. Therefore, unlike the partial comparison in the linear list approach, a pairwise vector comparison in the PLQT approach is a full comparison.

Just like in (10) and (13), the values of 2^k for $0 \le k \le K-1$ do not need to be computed each time when (15) is used in the implementation. They need to be computed only once and then stored in a K dimensional array for later use. In fact, the array of $2^{i'}$ for $0 \le i' \le b-1$ used in (10) and (13) is sufficient for (15) for any application. The value of each 2^k can also be obtained through the "bitwise shift" operation each time when it is used. Obtaining the value of 2^k through the "bitwise shift" operation does not take more time than locating it in an array. The value of ϕ in (15) can also be computed through bitwise manipulations because the ϕ_k th bit is simply toggled on when $\phi_k = 1$. In C, for example, it can be implemented through the "bitwise or" operation or the "bitwise exclusive or" operation.

3.3 Node Insertion and Retrieval in a PLQT

Routine process() in the following is used to process $\mathbf{z}$ for possible insertion into or for possible retrieval from $\mathcal{T}_{\mathbf{z}^r}$. If $\mathbf{z}$ is in $\mathcal{T}_{\mathbf{z}^r}$ already, it is retrieved; otherwise, it is inserted. The retrieved node is passed back through the argument $\mathbf{z}^u$. The process starts from $\mathbf{z}^r$ by determining the successorship ϕ of $\mathbf{z}$ to $\mathbf{z}^r$. If $\phi = 0$ and $\mathbf{z} = \mathbf{z}^r$, then $\mathbf{z}$ is in $\mathcal{T}_{\mathbf{z}^r}$ already and the process returns $\mathbf{z}^u = \mathbf{z}^r$ in Step 1. If the process continues, it checks $\mathcal{A}(\mathbf{z}^r)$, the eldest existing son of $\mathbf{z}^r$. If $\mathcal{F}(\mathbf{z}^r)$ does not exist or if its successorship is larger than ϕ, $\mathbf{z}$ is then inserted as the new $\mathcal{F}(\mathbf{z}^r)$ and the process returns $\mathbf{z}^u = \text{NULL}$ in Step 2. Otherwise, it follows the immediate younger sibling of each node until reaching the last node or reaching a node with a successorship larger than ϕ in Step 3. This step stops at a node $\mathbf{z}^s$ with $\mathcal{S}(\mathbf{z}^s) \le \phi$. If $\mathcal{S}(\mathbf{z}^s) = \phi$, then $\mathbf{z}^s$ becomes the root of the new subtree into which $\mathbf{z}$ is inserted or from which $\mathbf{z}$ is retrieved in Step 4. Otherwise if $\mathcal{S}(\mathbf{z}^s) < \phi$, $\mathbf{z}$ is inserted as $\mathcal{M}(\mathbf{z}^s)$ and as the $\phi-$son of $\mathbf{z}^r$, and the process returns $\mathbf{z}^u = \text{NULL}$ in Step 4. The routine is recursive. A pairwise vector comparison between $\mathbf{z}$ and an existing node of $\mathcal{T}_{\mathbf{z}^r}$ is made only at the root $\mathbf{z}^r$ each time when Routine process() is called.

Routine process($\mathbf{z}^r$, $\mathbf{z}$, $\mathbf{z}^u$)

Step 1. Determine ϕ by which $\mathbf{z}$ is a $\phi-$ successor of $\mathbf{z}^r$ through (14) and (15). If $\phi = 0$ and $\mathbf{z} = \mathbf{z}^r$, let $\mathbf{z}^u \leftarrow \mathbf{z}^r$ and return.

Step 2. If $\mathcal{A}(\mathbf{z}^r) = \text{NULL}$ or $\mathcal{S}(\mathcal{F}(\mathbf{z}^r)) > \phi$, let $\mathcal{M}(\mathbf{z}) \leftarrow \mathcal{A}(\mathbf{z}^r)$, $\mathcal{A}(\mathbf{z}^r) \leftarrow \mathbf{z}$, $\mathcal{A}(\mathbf{z}) \leftarrow \text{NULL}$, $\mathcal{S}(\mathbf{z}) \leftarrow \phi$ and $\mathbf{z}^u \leftarrow \text{NULL}$, and then return.

Step 3. Let $\mathbf{z}^s \leftarrow \mathcal{A}(\mathbf{z}^r)$. While $\mathcal{M}(\mathbf{z}^s) \ne \text{NULL}$ and $\mathcal{S}(\mathcal{N}(\mathbf{z}^s)) \le \phi$, let $\mathbf{z}^s \leftarrow \mathcal{M}(\mathbf{z}^s)$.

Step 4. If $\mathcal{S}(\mathbf{z}^s) = \phi$, execute process($\mathbf{z}^s$, $\mathbf{z}$, $\mathbf{z}^u$). Otherwise, let $\mathcal{M}(\mathbf{z}) \leftarrow \mathcal{M}(\mathbf{z}^s)$, $\mathcal{N}(\mathbf{z}^s) \leftarrow \mathbf{z}$, $\mathcal{A}(\mathbf{z}) \leftarrow \text{NULL}$, $\mathcal{S}(\mathbf{z}) \leftarrow \phi$ and $\mathbf{z}^u \leftarrow \text{NULL}$, and then return.

At each level of the PLQT where Routine process() reaches, at most one pairwise vector comparison is made. At the last level where it reaches, a pairwise vector comparison is made only when $\mathbf{z}$ is retrieved but no comparison is made when $\mathbf{z}$ is inserted. Therefore, the number of pairwise vector comparisons needed to process a node is determined by the level of the PLQT. For each node $\mathbf{z}^s$ chased in Step 3, only an integer comparison between ϕ and $\mathcal{S}(\mathbf{z}^s)$ is made. After such an integer comparison is made, all successors of $\mathbf{z}^s$ are jumped over. Furthermore, none of the successors $\mathbf{z}^s$ of $\mathbf{z}^r$ with $\mathcal{S}(\mathbf{z}^s) > \phi$ needs to be visited. Only the sons of one node may need to be searched at the next level where Routine process() reaches. Because a node has at most 2^K sons, the expected number of integer comparisons in Step 3 is at most 2^{K-1} at each level of the PLQT. As a result, only a very tiny portion of the PLQT needs to be searched to find a specific node in the PLQT or to insert a node into the PLQT. Hence, the PLQT approach for trial solution storage and retrieval is computationally very efficient.

When a trial solution is to be checked in a heuristic procedure, an integer vector $\mathbf{z}$ is determined first through (10) or (13). Then $\mathbf{z}$ is processed by Routine process() for possible insertion into or retrieval from $\mathcal{T}_{\mathbf{z}^r}$. If $\mathbf{z}$ is inserted into $\mathcal{T}_{\mathbf{z}^r}$, the process returns $\mathbf{z}^u$ = NULL. In this case, the trial solution will be evaluated and then stored with $\mathbf{z}$ in the PLQT. If $\mathbf{z}$ is in $\mathcal{T}_{\mathbf{z}^r}$ already, the process retrieves the solution with the stored node. In this case, the trial solution does not need to be evaluated again.

The three goals listed by Woodruff and Zemel [1993] for a hash function are now used as criteria to measure the PLQT approach. For criterion 1, the computational effort needed to update $\mathbf{z}$ in the PLQT approach is approximately the same as that needed to update h in the hashing approach if $\Phi = 2^b$. Both of them need an integer addition or subtraction as shown in either (7) or (13). As mentioned above, a "mod" operation is needed in the hashing approach if $\Phi < 2^b$. For criterion 2, although the PLQT approach needs to store the integer vector $\mathbf{z}$, the pointers $\mathcal{N}(\mathbf{z})$ and $\mathcal{A}(\mathbf{z})$, and the integer $\mathcal{S}(\mathbf{z})$ for an evaluated solution, it still uses only a very tiny portion of the memory space used by the hashing approach for any reasonable application because the hashing approach may need memory spaces for 2^b float numbers. The linear list approach needs to store the integer vector $\mathbf{z}$ for each evaluated solution but not the address part in a node of a PLQT. The PLQT approach needs more integer and pairwise vector comparisons than the hashing approach if the hashing approach does not have a mechanism to handle collision. However, comparisons are needed in the hashing approach if a mechanism handling collision is included. As shown by the computational results, the computational effort of the PLQT approach is negligible as compared to that of a heuristic procedure for any reasonable application even though comparisons are needed. For criterion 3, the PLQT approach completely eliminates collision while the probability of collision is high in the hashing approach even with $\Phi = 2^b$ [Carlton and Barnes, 1996].

In summary, as compared to the hashing approach, the PLQT approach needs equal computational effort and much less memory space but completely avoids the difficulty of collision. Compared to the linear list approach, the PLQT approach takes memory space to store $\mathcal{N}(\mathbf{z})$, $\mathcal{F}(\mathbf{z})$ and $\mathcal{S}(\mathbf{z})$ for each trial solution represented by $\mathbf{z}$ but saves tremendous amount of computation time.

3.4 Traversal of a PLQT

Sometimes the PLQT may need to be traversed, *e.g.*, when the contents of all nodes in the PLQT need to be printed. Routine outtree() in the following serves this purpose. To traverse $\mathcal{T}_{\mathbf{z}^r}$, the routine starts with $\mathbf{z}^r$ by

printing the content of $\mathbf{z}^r$ in Step 1. If $\mathcal{F}(\mathbf{z}^r)$ exists, then outtree($\mathcal{A}(\mathbf{z}^r)$) is executed in Step 2. If $\mathcal{N}(\mathbf{z}^r)$ exists, then outtree($\mathcal{M}(\mathbf{z}^r)$) is executed in Step 3. This routine is also recursive. The nodes in a PLQT may be traversed in different orders. The order used in Routine outtree() is called "preorder" by data structure textbooks for other types of tree data structures.

Routine outtree($\mathbf{z}^r$)

Step 1. Print out $\mathbf{z}^r$.

Step 2. If $\mathcal{A}(\mathbf{z}^r) \neq$ NULL, execute outtree($\mathcal{A}(\mathbf{z}^r)$).

Step 3. If $\mathcal{M}(\mathbf{z}^r) \neq$ NULL, execute outtree($\mathcal{M}(\mathbf{z}^r)$).

4. Examples

PLQTs can be depicted graphically. In a graph, each node is represented by a rectangle. The vector $\mathbf{z}$ is at the bottom and the successorshop to its parent $\mathcal{S}(\mathbf{z})$ is on the top of the rectangle. The pointers $\mathcal{N}(\mathbf{z})$ and $\mathcal{A}(\mathbf{z})$ are represented by arrows if they point to other nodes and are not shown if they point to NULL. PLQTs with data in Z^3 are used in the following examples. For easy illustration, each component of $\mathbf{z}$ has only one or two digits in the examples. In storing and retrieving trial solutions in a heuristic procedure with $b = 32$, each component of $\mathbf{z}$ has up to 10 digits. The purpose of these examples is to show how Routine process() works rather than to verify the efficiency of the PLQT approach.

4.1 Insertion of a Node between Siblings

Lets first process the vector $\mathbf{z}^1 =$ (17, 59, 7) for possible insertion into or retrieval from the PLQT in Figure 1. In Step 1 of Routine process(), $\mathbf{z}^1$ is determined to be a 4–successor of (3, 36, 27), the root of the PLQT. Because the root has an eldest existing son already, Step 2 is not executed. In Step 3, it chases the pointers and stops at node (80, 84, 26), the 4–son of the root. In Step 4, Routine process() is called again to insert $\mathbf{z}^1$ into or retrieve $\mathbf{z}^1$ from the subtree rooted at (80, 84, 26).

In Step 1 of Routine process(), $\mathbf{z}^1$ is determined to be a 7–successor of (80, 84, 26), the root of the subtree. Because (80, 84, 26) has an eldest existing son already, Step 2 is skipped. In Step 3, the chasing of pointers stops at node (47, 59, 12). Because the successorship of (47, 59, 12) is also 7, this node becomes the root of the new subtree into which $\mathbf{z}^1$ is to be inserted or from which $\mathbf{z}^1$ is to be retrieved.

In Step 1 of Routine process(), $\mathbf{z}^1$ is determined to be a 5–successor of (47, 59, 12). Step 2 is skipped again because (47, 59, 12) has an eldest existing son already. In Step 3, it stops at node (36, 49, 19) when chasing the pointers. Because the successorship of (36, 49, 19) is less than 5, $\mathbf{z}^1$ is inserted as the immediate existing younger sibling of (36, 49, 19) and as the 5–son of (47, 59, 12). The resulting PLQT is shown in Figure 2.

Figures 1 and 2 approximately here

4.2 Insertion of a Node as the Only Existing Son

Next $\mathbf{z}^2 =$ (48, 34, 63) is processed for possible insertion into or retrieval from the PLQT in Figure 2. In Step 1 of Routine process(), $\mathbf{z}^2$ is determined to be a 2–successor of (3, 36, 27), the root of the PLQT. Because (3, 36, 27) has an eldest existing son already, Step 2 is not executed. In Step 3, it stops at (67, 30, 32) when chasing the

pointers. Because (67, 30, 32) is the 2–son of (3, 36, 27), it becomes the root of the new subtree into which $\mathbf{z}^2$ is inserted or from which $\mathbf{z}^2$ is retrieved.

In Step 1 of Routine process(), $\mathbf{z}^2$ is determined to be a 1–successor of (67, 30, 32). Because (67, 30, 32) does not have any existing son, $\mathbf{z}^2$ is inserted as the 1–son, and also as the eldest existing son, of (67, 30, 32) in Step 2. The resulting PLQT is shown in Figure 3.

Figure 3 approximately here

4.3 Insertion of a Node as the Eldest Existing Son

In the following, $\mathbf{z}^3$ = (88, 38, 26) is processed for possible insertion into or retrieval from the PLQT in Figure 3. In Step 1 of Routine process(), $\mathbf{z}^3$ is determined to be a 4–successor of (3, 36, 27), the root of the PLQT. Because (3, 36, 27) has an eldest existing son already, Step 2 is not executed. In Step 3, the pointer chasing stops at (80, 84, 26). Because (80, 84, 26) is the 4–son of (3, 36, 27), it becomes the root of the new subtree into which $\mathbf{z}^3$ is inserted or from which $\mathbf{z}^3$ is retrieved.

In Step 1 of Routine process(), $\mathbf{z}^3$ is determined to be a 2–successor of (80, 84, 26). Because the successorship of the eldest existing son of (80, 84, 26) is larger than 2, $\mathbf{z}^3$ is inserted as the 2–son, and as the new eldest existing son, of (80, 84, 26) in Step 2. The resulting PLQT is shown in Figure 4.

Figure 4 approximately here

4.4 Retrieval of a Node

Finally $\mathbf{z}^4$ = (33, 54, 29) is processed for possible insertion into or retrieval from the PLQT in Figure 4. In Step 1 of Routine process(), $\mathbf{z}^4$ is determined to be a 0-successor of (3, 36, 27). Because (3, 36, 27) has existing sons already, Step 2 is not executed. In Step 3, the pointer chasing stops at (92, 76, 39) that becomes the root of the new subtree into which $\mathbf{z}^4$ is inserted or from which $\mathbf{z}^4$ is retrieved.

In Step 1 of Routine process(), $\mathbf{z}^4$ is determined to be a 7–successor of (92, 76, 39). Because (92, 76, 39) has existing sons already, Step 2 is not executed. In Step 3, it stops at (33, 54, 29) when chasing the pointers. Because (33, 54, 29) is the 7–son of its parent, it becomes the root of the new subtree into which $\mathbf{z}^4$ is inserted or from which $\mathbf{z}^4$ is retrieved.

In Step 1 of Routine process(), $\mathbf{z}^4$ is determined to be a 0–successor of (33, 54, 29) and to be equal to (33, 54, 29). Hence, (33, 54, 29) is retrieved.

5. Computational Results

A computational experiment is designed to test the performance of the proposed PLQT approach. The data structure and the algorithms were coded in C. To establish a benchmark, the linear list approach, also coded in C, is also used for the same test problems. Because the hashing approach uses a totally different strategy, it is not used in the computational experiment for comparison purpose. A tabu search heuristic procedure for the capacitated facility location problem is used as an example to provide some insight about the CPU time taken by the PLQT approach relative to the total CPU time taken by a heuristic procedure. All the computations were conducted on a Sun Enterprise 450 computer with two 400 Mhz processors (only one is used) and 1.5 GB RAM.

5.1 Test Problems

Randomly generated test problems are used. Each test problem is measured by the number of integer vectors m, *i.e.*, the number of trial solutions to be inserted into or retrieved from a PLQT, the dimension of the integer vectors K, and the rate of duplication r. Integer vectors are randomly generated from Z^K. With $b = 32$, each component z_k of any integer vector $\mathbf{z}$ is in the range $0 \le z_k < 2^{32}$. Five values for m are used with $m = 5{,}000$, 10,000, 20,000, 40,000 and 80,000, respectively. Three values for K are used with $K = 3$, 5 and 7, respectively. With $b = 32$, these K values are for binary optimization problems with up to $n = 96$, 160, and 224 binary variables. The values of r used are $r = 0.0$, 0.2, 0.4, 0.6 and 0.8, respectively. To imitate the process trial solutions are generated in a heuristic solution procedure, some integer vectors are duplicated. The duplicate integer vectors are mingled randomly with the rest. Before an integer vector is generated, a random number r' is generated first. If $r' < r$, one integer vector randomly selected among all those previously generated is copied; otherwise, a new integer vector is generated. Given the method these test problems are generated, the actual duplication rate in each test problem is approximate but very close to r, *i.e.*, approximately rm integer vectors appear more than once among the m integer vectors. Each combination of m, K and r defines a problem category. As a result, 75 problem categories are used in the computational experiment. For each problem category, 30 randomly generated test problems are used. For each problem, the integer vectors are processed one by one for possible insertion into or retrieval from a PLQT. An integer vector is inserted into the PLQT if it is not in the PLQT already and is retrieved otherwise. The integer vectors of each test problem are processed in the same order in the linear list approach as in the PLQT approach.

Some capacitated facility location problems in the OR-Library [Beasley, 1990] are used for the computational experiment to measure the CPU time taken by the PLQT approach relative to that taken by a heuristic procedure. The size of each problem is measured by the number of potential locations for the facilities n and the number of clients n''. The 12 problems with $n \times n'' = 50 \times 50$ and the 12 problems with $n \times n'' = 100 \times 1000$ are used. The test problems are organized into 6 groups with 4 problems in each group in the OR-Library. The names of the problems are originally used in the OR-Library. For each of these problems, a total of approximately 5,000 trial solutions are either evaluated or retrieved from the PLQT. When a trial solution is evaluated, a transportation problem is solved. ILOG CPLEX® 10.0.1 is used to solve transportation problems through the function call CPXNETprimopt(). After the first transportation problem is solved, each of the subsequent transportation problems is solved from the optimal solution of another. Starting from the optimal solution of one trial solution, 10 other trial solutions are checked. The basis of this starting solution is saved through the function call CPXNETgetbase(). Each of the 10 trial solutions is obtained from this starting solution by opening or closing one facility. A facility is closed by setting its capacity to 0 and is opened by restoring its original capacity through the function call CPXNETchgsupply(). The basis of the starting solution is passed to CPLEX by the function call CPXNETcopybase().

5.2 Test Results

For each test problem, the number of pairwise vector comparisons and CPU time used to process all vectors, denoted by c_p and t_p for the PLQT approach and c_l and t_l for the linear list approach, are recorded. The averages of c_p and t_p of the 30 test problems in each problem category, denoted by $\bar{c}_p$ and $\bar{t}_p$, are reported in Table 1. The averages of c_l and t_l of the 30 test problems in each problem category, denoted by $\bar{c}_l$ and $\bar{t}_l$, are

reported in Table 2. The ratio t_l/t_p is computed for each test problem. The minimum (min), average (avg) and maximum of these ratios of the 30 test problems in each problem category are reported in Table 3.

Tables 1, 2 and 3 approximately here

Given fixed m and K, $\bar{c}_p$ and $\bar{t}_p$ all decrease as r increases with one exception. As r increases, the number of nodes in the PLQT decreases. Therefore, fewer nodes need to be searched or compared when a vector is processed for possible insertion or retrieval. However, each vector must be processed whether it is inserted or retrieved. Hence, when r increased from 0.0 to 0.8, the number of nodes in the PLQT reduced by about 80%, $\bar{c}_p$ reduced slightly and $\bar{t}_p$ reduced less than 50%. When r increased from 0.0 to 0.2, $\bar{c}_p$ increased, rather than decreased, slightly for $K = 5$ and 7 as shown in Table 1. This exception is possibly caused by the shape of the PLQT. The decrease in $\bar{c}_l$ or $\bar{t}_l$ is more drastic as r increases as shown in Table 2. The ratio t_l/t_p also decreases as r increases as shown in Table 3. With this decrease, the linear list approach still takes much more CPU time than the PLQT approach even for problems with $r = 0.8$.

Given fixed m and r, $\bar{c}_p$ decreases but $\bar{t}_p$ increases when K increases. When K increases, the number of sons of each node increases and the PLQT becomes shorter and flatter in general. Therefore, the number of nodes to be compared decreases when a vector is processed. However, more components need to be compared when a pairwise vector comparison is made and more nodes need to be searched when a vector is processed and, hence, more CPU time is needed. For fixed m and r, no specific pattern is noticed for the changes in $\bar{c}_l$ while $\bar{t}_l$ increases slightly as K increases. The ratio t_l/t_p also decreases as K increases. However, t_l is from several times to several hundred times larger than t_p even with $K = 7$.

Given fixed K and r, $\bar{c}_p$, $\bar{t}_p$, $\bar{c}_l$ and $\bar{t}_l$ all increase as m is doubled each time. However, $\bar{c}_l$ grows much faster than $\bar{c}_p$ does and $\bar{t}_l$ grows much faster than $\bar{t}_p$ does. When m increased 16 folds from 5,000 to 80,000, $\bar{c}$ increased slightly over 20 folds and $\bar{t}$ increased slightly over 25 folds, but both $\bar{c}_l$ and $\bar{t}_l$ increased over 200 folds. The ratio t_l/t_p also increases as m increases. Hence, the more trial solutions need to be processed in a heuristic procedure, the more efficient the PLQT approach is as compared to the linear list approach.

The PLQT approach is undoubtedly very efficient as compared to the linear list approach. For any reasonable application, the PLQT approach does not cause any computational burden for the heuristic procedure as the results in Table 1 show. However, the CPU time taken by the linear list approach becomes considerable when n, therefore, K is large, when m is large, *i.e.*, when many solutions need to be processed, and when r is small, *i.e.*, when the duplication rate of the trial solutions is low. The number of pairwise vector comparisons needed when a vector is processed in a PLQT is roughly determined by the level of the PLQT. However, the number of comparisons in a linear list is determined roughly by the length of the list. Although most of the pairwise vector comparisons in the linear list approach are partial comparisons, it takes much more CPU time because it needs many more comparisons.

5.3 CPU Time Taken in a Heuristic Procedure

The computational results of the tabu search heuristic procedure for the capacitated facility location problem are reported in Table 4. The column labeled $\bar{m}_1$ is the average number of trial solutions stored in the final

PLQT, *i.e.*, average number of trial solutions evaluated for the 4 problems in each group. The column labeled $\bar{m}_2$ is the average number of solutions retrieved from the PLQT in the solution process for the 4 problems in each group. The minimum (min), average (avg) and maximum (max) CPU times used for the 4 problems in each group are reported. The CPU time includes the time used to manage the PLQT. For problems with $n = 50$ facilities, $K = 2$ components are required and for problems with $n = 100$ facilities, $K = 4$ components are required in each integer vector $\mathbf{z}$. Although $K = 2$ and $K = 4$ are not used in the computational experiment above, the results in Table 1 for $m = 5{,}000$ for other values of K indicate that the CPU time used to manage the PLQT is smaller than the rounding error of the CPU time taken by the tabu search heuristic procedure.

Table 4 approximately here

The amount of CPU time taken to manage the PLQT relative to the total CPU time taken by a heuristic procedure depends on the effort needed to evaluate trial solutions of the problem being solved. The capacitated facility location problem is used as an example only. For those problems with trial solutions easier to evaluate, such as the uncapacitated facility location problem [Sun, 2005, 2006a], relatively more CPU time may be taken to manage the PLQT. For such problems, the heuristic procedure may benefit less from the PLQT approach. When storing and retrieving trial solutions take more time than evaluating them, storing the evaluated solutions becomes unnecessary. For problems with trial solutions harder to evaluate, such as the single source capacitated facility location problem [Delmaire, Diaz, Fernandez and Ortega, 1998], the evaluation of trial solutions is more time consuming and the CPU time taken to manage the PLQT becomes less noticeable. For such hard problems, a heuristic procedure will benefit much more from the PLQT approach.

6. Conclusions

A PLQT data structure is proposed to store and retrieve trial solutions in heuristic procedures for binary optimization problems. Each trial solution represented by a binary vector is encoded into an integer vector that is used as the composite key to store and retrieve the trial solution in a PLQT. An algorithm is developed to insert trial solutions into or retrieve trial solutions from the PLQT and another algorithm is developed to traverse the PLQT. The PLQT approach is surprisingly efficient. It uses a very tiny portion of the CPU time used by the linear list approach for the same test problems and for any reasonable application. Compared to the total time used by a heuristic procedure to solve a reasonably complicated binary optimization problem, the CPU time taken to manage the PLQT is hardly noticeable.

The PLQT approach can be employed by any heuristic procedure to store and retrieve evaluated solutions. By employing the PLQT approach, heuristic procedures may become more powerful in solving hard binary optimization problems. Future research may be directed to applications of this PLQT approach to heuristic procedures for different binary optimization problems. With minor modifications in encoding a solution into an integer vector, the PLQT approach may be applied to combinatorial optimization problems with more general integer variables.

References

Beasley, J. E. (1990), "OR-Library: Distributing Test Problems by Electronic Mail," *Journal of the Operational Research Society*, **41**(11), 1069-1072.

Carlton, W.B. and J.W. Barnes (1996), "A Note on Hashing Functions and Tabu Search Algorithms," *European Journal of Operational Research*, **95**(1), 237-239.

Delmaire, H.J., J. A. Diaz, E. Fernandez and M. Ortega (1998), "Reactive Grasp and Tabu Search Based Heuristics for the Single Source Capacitated Plant Location Problem," *Information Systems and Operations Research*, **37**, 194-225.

Deneubourg, J.L. (1983). "Probabilistic Behaviour in Ants: A Strategy of Errors?" *Journal of Theoretical Biology*, **105**, 259–271.

Deneubourg, J.L. and S. Goss. (1989). "Collective Patterns and Decision-Making." *Ethology, Ecology and Evolution*, **1**, 295–311.

Ducati, E. A., V. A. Armentano, and M. Sun (2004), "A Tabu Search Heuristic Procedure for the Capacitated Facility Location Problem," Faculdade de Engenharia Elétrica e de Computação,Universidade Estadual de Campinas, Caixa Postal 6101, Campinas - SP, CEP 13083-970, Brazil.

Finkel, R. A. and J. L. Bentley (1974), "Quad-Trees, A Data Structure for Retrieval on Composite Keys," *Acta Informatica* **4**, 1-9.

Glover, F. (1989), "Tabu Search, Part I," *ORSA Journal on Computing*, **1**(3), 190-206.

Glover, F. (1990a), "Tabu Search, Part II," *ORSA Journal on Computing*, **2**(1), 4-32.

Glover, F. (1990b), "Tabu Search: A Tutorial," *Interfaces*, **20**(4) 74-94.

Glover, F. and M. Laguna (1997), *Tabu Search*, Kluwer Academic Publishers, Hingham, MA.

Glover, F., M. Laguna and R. Martí (2000), "Fundamentals of Scatter Search and Path Relinking," *Control and Cybernetics*, 39(3), 653-684.

Habenicht, W. (1982), "Quad Trees, A Datastructure for Discrete Vector Optimization Problems," *Lecture Notes in Economics and Mathematical Systems*, **209**, 136-145.

Habenicht, W. (1991), "ENUQUAD An Enumerative Approach to Discrete Vector Optimization Problems," Presented at the International Workshop on Multicriteria Decision Making: Methods, Algorithms and Applications, Liblici, Czechoslovakia, March 18-20.

Holland, J. (1992), *Adaptation in Natural and Artificial Systems, An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*, MIT Press, Cambridge, MA.

Kirkpatrick, S., C. D. Gelatt Jr., and M. P. Vecchi (1983), "Optimization by Simulated Annealing," *Science*, **220**, 671-680.

Klein, R. (2000), "Project Scheduling with Time-Varying Resources Constraints," *International Journal of Production Research*, **38**(16), 3937-3952.

Nemhauser, G. L. and L. A. Wosley (1988), *Integer and Combinatorial Optimization*, Wiley, New York.

Stummer, C and M. Sun (2005), "New Multiobjective Metaheuristic Solution Procedures for Capital Investment Planning," *Journal of Heuristics*, **11**(3), 183-199.

Sun, M. (2005), "A Tabu Search Heuristic Procedure for the Uncapacitated Facility Location Problem," in C. Rego and B. Alidaee (eds.), *Metaheuristic Optimization via Memory and Evolution: Tabu Search and Scatter Search*, Kluwer Academic Publishers, Boston, MA, pp. 191-211.

Sun, M. (2006a), "Solving Uncapacitated Facility Location Problems Using Tabu Search," *Computers and Operations Research*, **33**(9), 2563-2589.

Sun, M. (2006b), "A Primogenitary Linked Quad Tree Data Structure and Its Application to Discrete Multiple Criteria Optimization," *Annals of Operations Research*, **147**(1), 87-107.

Sun, M. and R. E. Steuer (1996a), "Quad Trees and Linear List for Identifying Nondominated Criterion Vectors," *INFORM Journal on Computing*, **8**(4), 367-375.

Sun, M. and R. E. Steuer (1996b), "InterQuad: An Interactive Quad Tree Based Procedure for Solving the Discrete Alternative Multiple Criteria Problem," *European Journal of Operational Research*, **89**(3), 462-472.

Woodruff, D.L. and E. Zemel (1993), "Hashing Vector for Tabu Search," *Annals of Operations Research*, **41**, 123-137.

Wosley, L. A. (1998), *Integer Programming*, Wiley, New York.

Table 1. Computational Results of the PLQT Approach

m / r	5000		10000		20000		40000		80000	
	$\bar{c}_p$	$\bar{t}_p$	$\bar{c}_p$	$\bar{t}_p$	$\bar{c}_p$	$\bar{t}_p$	$\bar{c}_p$	$\bar{t}_p$	$\bar{c}_p$	$\bar{t}_p$
$K = 3$										
0.0	28424	0.0237	61464	0.0503	132169	0.1120	282830	0.2570	602615	0.5943
0.2	27905	0.0213	60404	0.0480	129966	0.1067	278058	0.2437	590427	0.5613
0.4	26321	0.0200	57157	0.0460	123327	0.0973	264339	0.2237	560455	0.5103
0.6	23441	0.0173	51157	0.0380	110891	0.0867	238473	0.1913	504945	0.4363
0.8	17599	0.0127	38639	0.0290	84274	0.0620	182265	0.1387	386249	0.3080
$K = 5$										
0.0	18634	0.0253	40054	0.0557	85651	0.1297	182384	0.3013	386971	0.7167
0.2	18734	0.0233	40217	0.0567	85922	0.1257	182744	0.2900	386206	0.6790
0.4	18179	0.0237	39063	0.0517	83538	0.1170	177681	0.2663	374456	0.6197
0.6	16865	0.0210	36363	0.0473	77921	0.1050	165850	0.2347	348809	0.5373
0.8	13360	0.0153	28889	0.0343	62062	0.0777	132466	0.1757	278231	0.3900
$K = 7$										
0.0	14291	0.0337	30557	0.0800	65087	0.1887	138076	0.4630	288573	1.1743
0.2	14555	0.0330	31080	0.0763	66106	0.1800	140023	0.4353	294839	1.0923
0.4	14548	0.0310	31056	0.0733	65996	0.1717	139542	0.4007	292762	0.9853
0.6	13842	0.0277	29558	0.0633	62855	0.1543	132965	0.3570	278096	0.8473
0.8	11525	0.0207	24638	0.0477	52403	0.1120	110960	0.2603	231667	0.6030

Table 2. Computational Results of the Linear List Approach

m / r	5000		10000		20000		40000		80000	
	$\bar{c}_l$	$\bar{t}_l$	$\bar{c}_l$	$\bar{t}_l$	$\bar{c}_l$	$\bar{t}_l$	$\bar{c}_l$	$\bar{t}_l$	$\bar{c}_l$	$\bar{t}_l$
$K = 3$										
0.0	12496858	1.587	49993215	6.430	199980036	25.783	799937352	103.234	3199785257	414.213
0.2	8879556	1.118	35467261	4.533	141379282	18.191	563330492	72.655	2192434852	283.706
0.4	5608330	0.693	22336170	2.836	88765623	11.383	352124469	45.327	1332403868	172.168
0.6	2873861	0.340	11359885	1.412	44851864	5.709	176338038	22.611	647377099	83.455
0.8	847069	0.102	3331913	0.395	13029665	1.614	50748882	6.427	180148163	23.061
$K = 5$										
0.0	12497234	1.654	49992805	6.638	199979979	26.595	799943768	106.752	3199802667	427.956
0.2	8867034	1.165	35399221	4.687	141388758	18.776	563229350	75.005	2192990373	293.282
0.4	5577161	0.723	22234677	2.935	88517096	11.727	351148881	46.654	1330281139	177.178
0.6	2866275	0.362	11347746	1.479	44707230	5.901	175964365	23.313	646740757	85.882
0.8	846247	0.102	3332865	0.421	13020010	1.691	50731183	6.667	180010235	23.775
$K = 7$										
0.0	12496822	1.663	49991684	6.645	199978553	26.671	799940294	106.880	3199795269	434.051
0.2	8879991	1.176	35433359	4.712	141310223	18.810	563189584	75.112	2189843834	295.678
0.4	5597304	0.735	22303949	2.955	88529166	11.766	351168453	46.821	1330322650	178.794
0.6	2837710	0.367	11278113	1.485	44632921	5.922	175973574	23.386	646913404	86.457
0.8	839395	0.105	3315650	0.430	13030667	1.714	50739120	6.721	180017606	23.906

Table 3. CPU Time Ratios of the Linear List Approach to the PLQT Approach

r \ m	5000			10000			20000			40000			80000		
	min	avg	max	min	avg	max	min	avg	max	min	avg	max	min	avg	max
$K = 3$															
0.0	52.67	69.66	79.50	91.57	129.87	161.00	214.42	230.62	257.70	368.89	402.19	431.33	660.35	697.61	738.82
0.2	36.67	53.45	57.50	75.33	95.46	115.00	151.00	171.07	183.70	277.88	298.63	329.55	473.62	505.85	534.64
0.4	33.00	34.63	36.00	46.50	62.56	72.75	102.09	117.39	141.75	179.16	203.12	217.48	299.75	337.75	357.77
0.6	16.00	21.55	36.00	34.00	37.64	47.67	57.40	66.16	72.75	106.62	118.45	128.11	173.31	191.51	205.41
0.8	4.50	8.78	11.00	10.25	13.88	19.50	22.43	26.22	33.80	39.94	46.47	50.77	67.15	75.05	83.11
$K = 5$															
0.0	54.67	67.98	84.00	95.00	120.43	134.20	176.80	205.89	242.73	322.48	355.17	394.37	541.20	598.46	658.38
0.2	38.00	51.79	60.00	76.17	83.32	95.20	132.79	150.16	172.18	233.22	259.33	280.30	396.24	432.75	478.25
0.4	23.33	31.73	38.50	48.17	57.07	60.00	89.62	100.77	118.10	156.20	175.69	196.88	257.30	286.76	318.79
0.6	12.00	17.49	19.00	28.40	31.54	37.75	49.33	56.50	65.44	89.19	99.63	112.81	144.46	160.29	177.57
0.8	4.50	7.57	12.00	9.75	12.49	15.00	18.56	21.91	25.14	31.67	38.11	41.19	52.38	61.18	68.97
$K = 7$															
0.0	41.25	50.36	56.33	66.30	84.68	110.67	120.82	143.13	178.80	197.31	233.17	283.55	310.78	373.15	466.90
0.2	28.75	36.91	58.00	51.56	62.88	93.20	88.67	105.71	133.79	141.45	174.26	215.00	228.47	272.83	312.97
0.4	18.00	24.29	36.00	32.11	40.94	58.20	58.05	69.45	85.00	97.54	118.32	146.28	154.42	183.29	221.23
0.6	11.33	13.67	19.00	18.50	23.86	30.60	31.94	39.05	55.64	54.43	66.51	87.30	86.05	103.42	133.43
0.8	3.67	5.11	6.00	6.67	9.23	15.33	11.86	15.54	20.38	19.85	26.24	34.80	31.88	40.27	53.04

Table 4. Computational Time Taken by Some Capacitated Facility Location Problems

Problem Name	$\bar{m}_1$	$\bar{m}_2$	Problem Size	CPU Time		
				min	avg	max
cap111-cap114	4984	24	50×50	14.180	15.330	16.290
cap121-cap124	4889	112	50×50	14.020	15.023	15.950
cap131-cap134	4842	160	50×50	15.120	15.445	15.620
capa1-capa2	4886	121	100×1000	2504.277	2768.825	3117.097
capb1-capb2	5004	0	100×1000	2655.087	2769.660	2839.397
capc1-capc2	5006	0	100×1000	2489.227	2663.425	2914.147

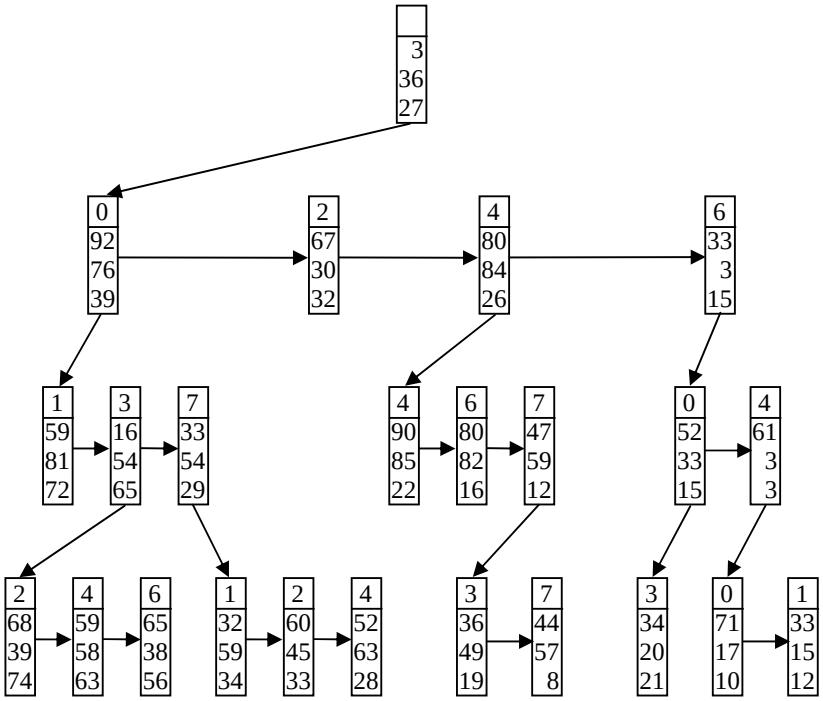

Figure 1. An Example PLQT

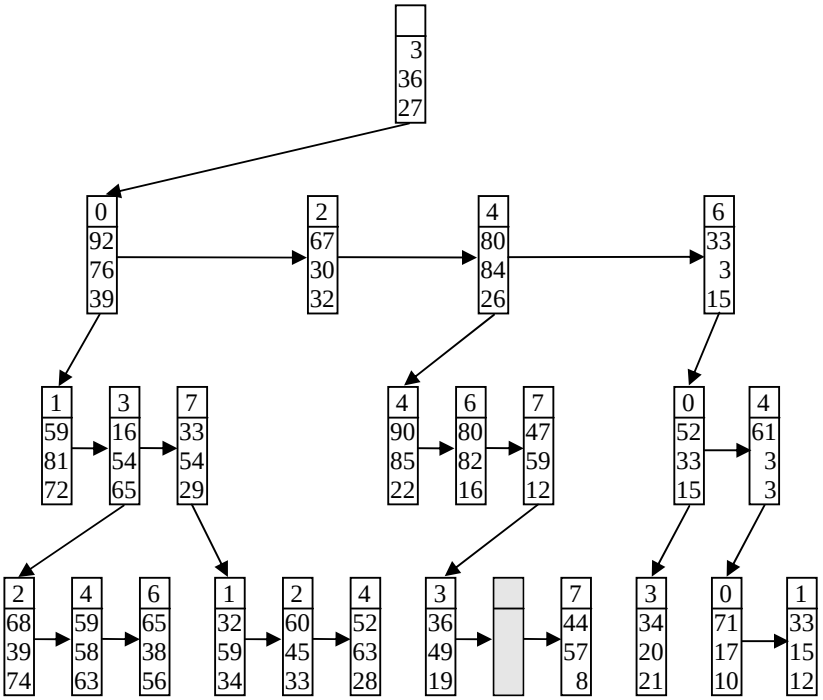

Figure 2. The Example PLQT after the Insertion of (17, 59, 7)

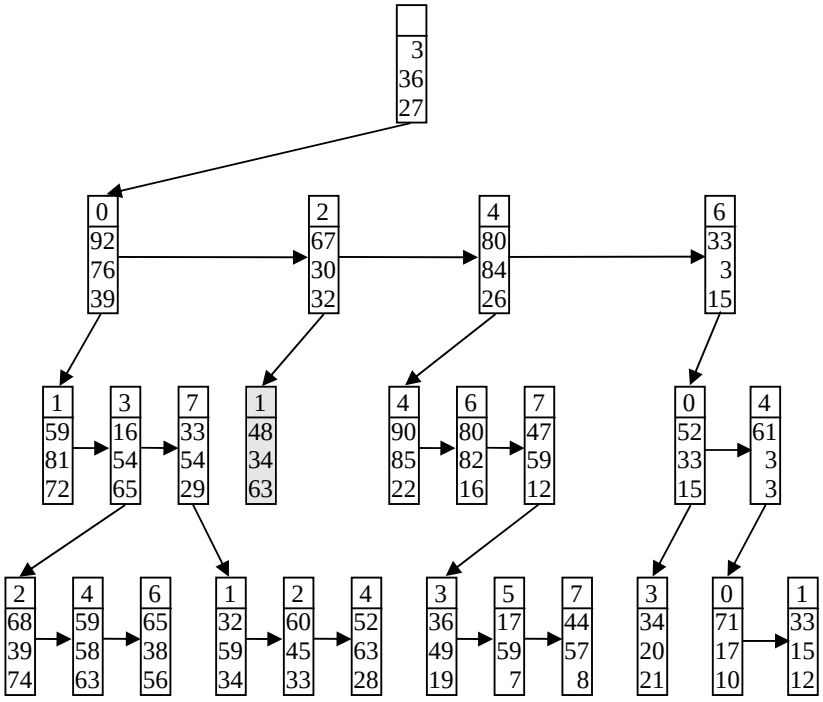

Figure 3. The Example PLQT after the Insertion of (48, 34, 63)

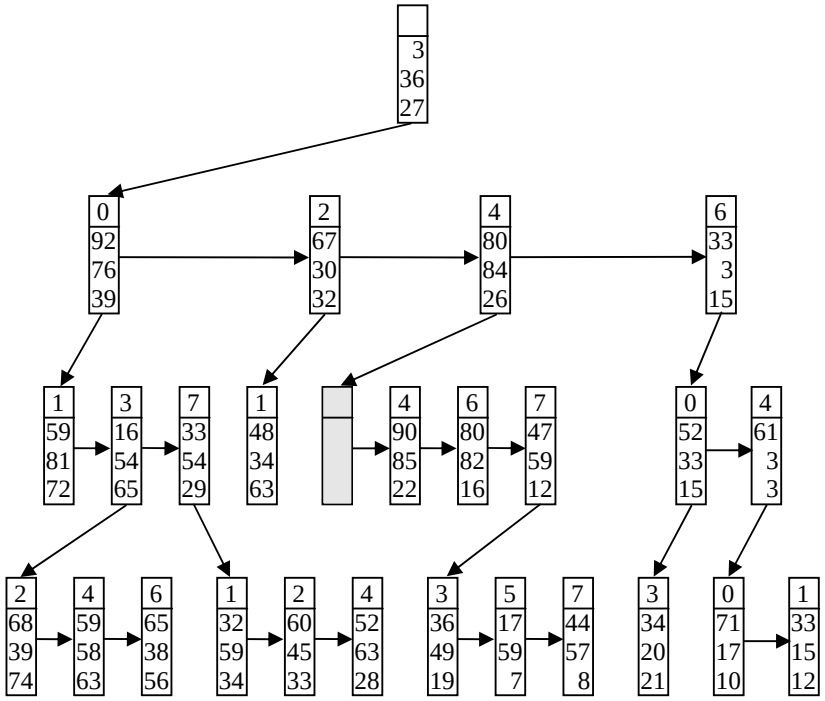

Figure 4. The Example PLQT after the Insertion of (88, 38, 26)