Journal of Heuristics (2022) 28:433–479
https://doi.org/10.1007/s10732-022-09498-0

Data structures for speeding up Tabu Search when solving sparse quadratic unconstrained binary optimization problems

Ricardo N. Liang[1] · Eduardo A. J. Anacleto[1] · Cláudio N. Meneses[1]

Received: 25 August 2021 / Revised: 28 December 2021 / Accepted: 7 April 2022 /
Published online: 27 April 2022

Abstract
The quadratic unconstrained binary optimization (QUBO) problem belongs to the NP-hard complexity class of problems and has been the subject of intense research since the 1960s. Many problems in various areas of research can be reformulated as QUBO problems, and several reformulated instances have sparse matrices. Thus, speeding up implementations of methods for solving the QUBO problem can benefit all of those problems. Among such methods, Tabu Search (TS) has been particularly successful. In this work, we propose data structures to speed up TS implementations when the instance matrix is sparse. Our main result consists in employing a compressed sparse row representation of the instance matrix, and priority queues for conducting the search over the solution space. While our literature review indicates that current TS procedures for QUBO take linear time on the number of variables to execute one iteration, our proposed structures may allow better time complexities than that, depending on the sparsity of the instance matrix. We show, by means of extensive computational experiments, that our techniques can significantly decrease the processing time of TS implementations, when solving QUBO problem instances with matrices of relatively high sparsity. To assess the quality of our results regarding more intricate procedures, we also experimented with a Path Relinking metaheuristic implemented with the TS

Eduardo A. J. Anacleto and Cláudio N. Meneses have contributed equally to this work.

✉ Ricardo N. Liang
ricardo.liang@ufabc.edu.br

Eduardo A. J. Anacleto
eduardo.anacleto@ufabc.edu.br

Cláudio N. Meneses
claudio.meneses@ufabc.edu.br

1 Center of Mathematics, Computation and Cognition, Federal University of ABC, Avenida dos Estados 5001, Santo André, SP 09210-580, Brazil

using our techniques. This experiment showed that our techniques can allow such metaheuristics to become more competitive.

Keywords Zero-one optimization · Tabu Search · Sparsity · Priority queues · Computational efficiency

1 Introduction

The Quadratic Unconstrained Binary Optimization (QUBO) problem belongs to the NP-hard class of computational complexity and can be defined as

$$\begin{aligned} \min \quad & f(x) = x^{\mathsf{T}} Q x \\ \text{s.t.} \quad & x \in \{0, 1\}^n, \end{aligned}$$

where n is a positive integer number and $Q = [q_{ij}]$ is a rational $n \times n$ matrix. The QUBO problem has significant theoretical and practical importance, since many other combinatorial optimization problems, and also problems related to various other areas of research, can be reformulated as QUBO problems (Kochenberger et al. 2014; Branda et al. 2016; Oliveira et al. 2018; Milne et al. 2018; Hua and Dinneen 2019). Moreover, the QUBO problem was proved to be equivalent to the problem of finding ground states of Ising spin glasses (Lucas 2014; Boettcher 2019; Aramon et al. 2019). In recent years, D-Wave quantum annealers (Bian et al. 2014; Boixo et al. 2014; Cruz-Santos et al. 2018; Pastorello and Blanzieri 2019) have been developed to solve instances of the QUBO problem. Hence, this problem also has considerable importance within the context of quantum computing.

Several problems that can be reformulated to QUBO form may result in instances with sparse matrices, that is, matrices with a ratio of zero to nonzero components large enough to be taken advantage of (Duff 1977), typically above 50%. In the cases of graph problems such as Maximum Cut, Maximum Clique and Maximum Weight Edge Clique (Kochenberger et al. 2011, 2014; Chapuis et al. 2018), instances of these problems can be reformulated to QUBO problem instances where the off-diagonal components q_{ij} are zero whenever an edge (i, j) does not exist in the graph. Thus, in these cases, the sparsity of the resulting QUBO instances are approximately the same as those of the original graphs. Conversely, reformulations of instances of the Maximum Independent Set and Maximum Edge Weight Independent Set problems (Chapuis et al. 2018) result in sparse QUBO instances when the original graphs are dense. The reformulation of Maximum Satisfiability problems (Kochenberger et al. 2014) may also result in sparse QUBO instances. In the case of Maximum 2-SAT, off-diagonal components q_{ij} of the reformulated matrix are zero whenever no clause in the 2-SAT formula contains the variables i and j. As for Maximum 3-SAT, instances can be first reformulated to Maximum Independent Set and then to QUBO. Since the graph of the Maximum Independent Set problem instance does not have edges connecting any pair of literals that do not conflict, then the reformulated QUBO problem instances may also be relatively sparse.

Heuristic and metaheuristic methods have been widely used in the literature for solving QUBO problem instances. According to a literature review by Kochenberger et al. (2014), instances where $n \geq 1000$ have almost always been solved heuristically, and Tabu Search (TS) is the most used heuristic method. Thus, research on improvements to TS is valuable.

Tabu Search (TS) (Glover and Laguna 1997; Glover et al. 2018) is a metaheuristic whose main distinguishing feature is the use of memory structures to prevent solutions from being repeatedly visited in the short term or long term. The memory structure is also often called tabu list. Many successful uses of TS to solve QUBO problem instances have involved using it as a subroutine of other metaheuristics. Some examples include Iterated Tabu Search (Palubeckis 2006), Diversification-Driven Tabu Search (D^2TS) (Glover et al. 2010; Kochenberger et al. 2011), Greedy Randomized Adaptive Search Procedures (GRASP) (Wang et al. 2013) and Path Relinking (Wang et al. 2012; Glover 2014; Samorani et al. 2019).

To the best of our knowledge, all TS procedures that have been described in the literature for the QUBO problem, for example, those used in (Palubeckis 2006; Glover et al. 2010; Wang et al. 2011, 2012, 2013; Palubeckis 2004; Lü et al. 2010; Alidaee et al. 2017; Glover et al. 1998), impose an asymptotic time complexity of at least $O(n(1+m))$ per iteration, where n is the number of variables related to the instance being solved, and m is the number of local search iterations performed in that TS iteration. That number may be exponential in n in the worst case (Papp 2016).

The TS procedure explores the solution space of an instance through flip moves, which consist in changing the value of variables in a solution from 0 to 1 or from 1 to 0. The process of selecting a nonforbidden flip move is often described as a loop that iterates through every possible move, and the process of updating the tabu list after applying a move is described as a loop that decrements an iteration counter for each variable. These processes take at least $O(n)$ time each.

Speeding up exact and heuristic methods is important because these methods may become able to explore more solutions in less time, which may also lead them to find better solutions in less time. Using data structures to store information on the instances and solutions has been crucial for that purpose. For example, Glover and Hao (2010) proposed storing changes in objective function value incurred by flip moves in a vector. This allowed evaluating these moves in constant time, and updating the vector in linear time in n. Since then, that technique has been used in several subsequent heuristics that have been developed for solving QUBO problem instances (Alidaee et al. 2017; Wang et al. 2012). Anacleto et al. (2020) also proposed algorithms based on that same data structure for evaluating multiple simultaneous flip moves.

In the literature, several works investigate relationships between the sparsity of an instance and the computational effort required to reach high-quality solutions using different methods (Glover et al. 2002; Merz and Katayama 2004; Boros et al. 2007; Dunning et al. 2018; Lewis et al. 2019). For solving instances with sparse matrices, Glover and Hao (2010) proposed algorithms for quickly evaluating flip moves, and also variants of those algorithms for the cases where the instance matrix is stored in data structures for sparse matrices. However, these results do not improve the asymptotic time complexity of each iteration of the TS procedure for the QUBO problem with respect to $O(n(1+m))$, since it still iterates through n variables to select a flip move and to update the tabu list.

In this work, we seek to speed up implementations of the TS procedure for QUBO problem when the instance matrix is sparse. First, we store the instance matrix in a Compressed Sparse Row (CSR) format, which was proposed by Eisenstat et al. (1982). Storing the instance matrix in this format allowed us to reduce the asymptotic time complexity of accessing the nonzero components of each row and column of a matrix. Next, we adapt the fast objective function evaluation algorithm proposed by Glover and Hao (2010) for the case when the instance matrix is stored in that format. Then, we propose additional data structures based on priority queues that allow reducing the time complexities of several steps of the procedure. Although the CSR format and priority queues are well-known data structures, to the best of our knowledge, this work is the first that aims to use them to speed up heuristics for solving QUBO instances. We believe that the same techniques could be employed to improve methods to solve other combinatorial optimization problems.

As our main theoretical result, our proposed data structures allow executing a single iteration of the TS procedure in $O(D \log n(1+m))$ time, where D is an integer number that can vary from 1 to n and is related to the coefficients of the instance matrix. Depending on the value of D, that time complexity may become $o(n(1+m))$, which is an improvement over $O(n(1+m))$.

The remaining content of this paper is organized as follows. In Sect. 2 we present concepts and notations which we use throughout this work. In Sect. 3 we review the customary TS procedure that is used in the literature. In Sect. 4 we propose a TS procedure using data structures for representation of sparse matrices. In Sect. 5 we extend the results of the previous section with additional data structures for speeding up implementations of TS. In Sect. 6 we present our computational experience with the algorithms developed in this work. Lastly, we give our concluding remarks in Sect. 7.

2 Concepts and notations

Throughout this paper, we assume $\mathbb{Z}_{>0}$ to be the set of positive integer numbers, $\mathbb{Z}_{\geq 0}$ to be the set of nonnegative integer numbers, $n \in \mathbb{Z}_{>0}$, $N = \{1, 2, \ldots, n\}$, $\mathbb{Z}^n_{>0}$ to be the set of n-dimensional vectors whose components are positive integer numbers, $\{0, 1\}^n$ to be the set of n-dimensional binary vectors, $\mathbb{Q}_{>0}$ to be the set of positive rational numbers, $[0, 1]_{\mathbb{Q}}$ to be the set of numbers in the rational interval $[0, 1]$, $\mathbb{Q}^n$ to be the set of n-dimensional rational vectors, and $\mathbb{Q}^{n\times n}$ to be the set of $(n \times n)$-dimensional rational matrices.

Glover and Hao (2010) observed that when the matrix $Q \in \mathbb{Q}^{n\times n}$ is in the lower triangular form, it is possible to mitigate some computational operations when solving the instance. They also noticed that it takes polynomial time in terms of n to preprocess Q into that form. With that in mind, we consider that Q is always in the lower triangular form in this work.

Following, we formally define the QUBO problem.

Problem 1 Quadratic Unconstrained Binary Optimization (QUBO) problem

Instance: a positive integer n and a lower triangular matrix $Q \in \mathbb{Q}^{n\times n}$.

Task: obtaining a vector $x \in \{0,1\}^n$ that minimizes the value of the objective function $f(x) = x^\mathsf{T} Q x = \sum_{i=1}^{n} \sum_{j=1}^{i} x_i q_{ij} x_j$.

In Definition 1, we define a function to obtain the number of nonzero components in a given row and column of the lower triangular matrix. Several algorithms presented throughout this work have asymptotic time complexities that are expressed in terms of this function.

Definition 1 Let $\mathcal{I}$ be the set of all instances of the QUBO problem, $I = (n, Q) \in \mathcal{I}$, $N = \{1, 2, \ldots, n\}$, and $i \in N$. Then, the number of nonzero coefficients in the i-th row and column of Q may be computed using the function $\deg : \mathcal{I} \times \mathbb{Z}_{>0} \to \mathbb{Z}_{\geq 0}$, with

$$\deg(I, i) = |\{j \in N : (j \leq i \text{ and } q_{ij} \neq 0) \text{ or } (j > i \text{ and } \ q_{ji} \neq 0)\}|$$

for $i = 1, 2, \ldots, n$.

In Example 1 we present the values of the deg values for a sample instance.

Example 1 Let $I = (n, Q)$, where $n = 3$ and $Q = \begin{bmatrix} 5 & & \\ 0 & 6 & \\ 7 & 0 & 8 \end{bmatrix}$. Then, $\deg(I, 1) = 2$, $\deg(I, 2) = 1$, and $\deg(I, 3) = 2$.

In Definition 2, we define a function to compute the maximum value that can be obtained by using the function deg. We also use this function for analyzing asymptotic time complexities.

Definition 2 Let $\mathcal{I}$ be the set of all instances of the QUBO problem, $I = (n, Q) \in \mathcal{I}$, and $N = \{1, 2, \ldots, n\}$. Then, the maximum value of $\deg(I, i)$ for $i \in N$ can be computed using the function $D : \mathcal{I} \to \mathbb{Z}_{\geq 0}$, where $D(I) = \max\limits_{i \in N} \deg(I, i)$.

In Definition 3, we define a function to compute the sparsity of an instance matrix. In this work, we focus on solving problem instances with relatively high sparsity.

Definition 3 Let $\mathcal{I}$ be the set of all instances of the QUBO problem, and $I = (n, Q) \in \mathcal{I}$. Then, the sparsity of Q can be computed using the function $\operatorname{spr} : \mathcal{I} \to [0, 1]_{\mathbb{Q}}$, where $\operatorname{spr}(I) = 1 - \dfrac{\sum_{i=1}^{n} \deg(I, i)}{n^2}$.

For simplicity, when an instance I is implicit by the context, we may refer to $\deg(I, i)$, $D(I)$, $\operatorname{spr}(I)$ by $\deg i$, D, and spr.

The customary TS procedure for the QUBO problem described in the literature explores the solution spaces of an instance through the so-called 1-flip move, which consists in changing the value of one component in a solution vector from 0 to 1 or from 1 to 0. Glover and Hao (2010) proposed precomputing, for each i-th component of the solution vector x, the values of $(1 - 2x_i)(q_{ii} + \sum_{j<i} q_{ij} x_j + \sum_{j>i} q_{ji} x_j)$

into a vector labeled Δx, which we refer to as the reevaluation vector of x. Once Δx has been computed, it becomes possible to evaluate each possible 1-flip move on x in constant time. We state some of these results in Fact 1. They are also a specific case of the more general r-flip moves, whose evaluation formulae were proved by Anacleto et al. (2020).

Fact 1 (Glover and Hao 2010) *Let $I = (n, Q)$ be an instance of the* QUBO *problem, $x \in \{0, 1\}^n$ a solution to I, $\Delta x \in \mathbb{Q}^n$ a vector such that the value of each component is defined as $\Delta x_i = (1 - 2x_i)(q_{ii} + \sum_{j<i} q_{ij} x_j + \sum_{j>i} q_{ji} x_j)$ for $i = 1, 2, \ldots, n$, $k \in \{1, 2, \ldots, n\}$, and $x' \in \{0, 1\}^n$ a solution to I obtained by flipping the k-th component of x. Then, the values of the objective function $f(x')$ and components of the $\Delta x'$ vector are given by*

$$f(x') = f(x) + \Delta x_k, \text{ and}$$
$$\Delta x'_j = \begin{cases} \Delta x_j - q_{kj}(1 - 2x'_k)(1 - 2x'_j) & \text{if } j \in \{1, 2, \ldots, k-1\} \\ \Delta x_j - q_{jk}(1 - 2x'_k)(1 - 2x'_j) & \text{if } j \in \{k+1, k+2, \ldots, n\} \\ -\Delta x_j & \text{if } j = k. \end{cases}$$

Finally, in Definition 4, we recall the concepts of asymptotic notations O and o, which we use to compare asymptotic time complexities of our algorithms. More information about these concepts can be found in Cormen et al. (2001) and Manber (1989).

Definition 4 Let g and h be two monotonically increasing functions defined on some subset of the real numbers. Then, $h(w) \in o(g(w))$ if and only if there exist constants $c > 0$ and $w_0 > 0$ such that $h(w) < cg(w)$ for all $w \geq w_0$, and $h(w) \in O(g(w))$ if and only if there exist constants $c > 0$ and $w_0 > 0$ such that $h(w) \leq cg(w)$ for all $w \geq w_0$.

In the next section, we analyze the asymptotic time complexities involved in each step of the TS procedure used in the literature for the QUBO problem, so that we can assess the quality of our results in later sections.

3 TS procedure for the QUBO problem

We present, in Algorithm 1, a description of the TS procedure that has been frequently used for solving instances of the QUBO problem in the literature (Alidaee et al. 2017; Glover et al. 1998, 2010; Lü et al. 2010; Palubeckis 2004, 2006; Wang et al. 2011, 2012, 2013). Although it is a well-known procedure, we find it desirable to describe it here in more detail than usually described in the literature, so that we can accurately determine the asymptotic time complexities of each of its steps and, later on, compare them with those of our proposed algorithms.

This algorithm and its subroutines assume that the instance matrix is stored using a full matrix representation, that is, for each row of the matrix, all components of that row, including zeros, are stored contiguously. This allows accessing any arbitrary component in constant time. Additionally, we assume that generating a pseudo-random number, in Line 15, takes constant time. The vector x represents the solution within the current solution space being explored, and x^* represents the incumbent solution (i.e., the current best solution found so far in the search procedure). The numbers f_x and f_x^* denote the objective function values of x and x^*. The vectors Δx and Δx^* denote the reevaluation vectors of x and x^*, as defined in Fact 1. In Lines 1 and 19 we denote that the vectors and values on the right-hand side are being copied to the left-hand side. The parameters $tt_{\min}$ and $tt_{\max}$ specify the minimum and maximum number of iterations that a variable may remain tabu. The stop condition of the algorithm is usually that no improved solution is found for a number of consecutive iterations. Each iteration of this algorithm has asymptotic time complexity $O(n(1 + m))$, where $m \in \mathbb{Z}_{\geq 0}$ is the number of local search iterations performed.

Algorithm 1 TABU_SEARCH

```
Input: $n, Q, x^*, f_x^*, \Delta x^*, tt_{\min}, tt_{\max}$
Output: $x^*, f_x^*, \Delta x^*$
1: $(x, f_x, \Delta x) \leftarrow (x^*, f_x^*, \Delta x^*)$
2: for $i \leftarrow 1$ to $n$ do
3:     $L_i \leftarrow 0$
4: end for
5: repeat
6:     $k \leftarrow 0$
7:     for $j \leftarrow 1$ to $n$ do
8:         if $(f_x + \Delta x_j < f_x^*$ or $L_j = 0)$ and $(k = 0$ or $\Delta x_j < \Delta x_k)$ then
9:             $k \leftarrow j$
10:        end if
11:        if $L_j > 0$ then
12:            $L_j \leftarrow L_j - 1$
13:        end if
14:    end for
15:    $L_k \leftarrow$ a random integer in the interval $[tt_{\min}, tt_{\max}]$
16:    $(x, f_x, \Delta x) \leftarrow$ FLIP$(n, Q, x, f_x, \Delta x, k)$
17:    if $f_x < f_x^*$ then
18:        $(x, f_x, \Delta x) \leftarrow$ LOCAL_SEARCH$(n, Q, x, f_x, \Delta x)$
19:        $(x^*, f_x^*, \Delta x^*) \leftarrow (x, f_x, \Delta x)$
20:    end if
21: until a stop condition has been fulfilled
```

From this point on, we may refer to Algorithm 1 as `TS-BASIC`. In the subsections that follow, we discuss various aspects of this algorithm, such as the maintenance of the reevaluation vector and the tabu list.

3.1 Reevaluation vector initialization

The solution space explored by this TS procedure is defined in terms of 1-flip moves. To allow fast evaluation of 1-flip moves, in accordance to Fact 1, we maintain reevaluation vectors, which can be initialized using Algorithm 2. This algorithm has asymptotic time complexity $O(n^2)$.

Algorithm 2 INITIALIZE_REEVALUATION_VECTOR

```
Input: $n, Q, x$
Output: $\Delta x$
1: for $i \leftarrow 1$ to $n$ do
2:     $\Delta x_i \leftarrow q_{ii}$
3:     for $j \leftarrow 1$ to $i - 1$ do
4:         $\Delta x_i \leftarrow \Delta x_i + q_{ij} x_j$
5:     end for
6:     for $j \leftarrow i + 1$ to $n$ do
7:         $\Delta x_i \leftarrow \Delta x_i + q_{ji} x_j$
8:     end for
9:     $\Delta x_i \leftarrow (1 - 2x_i)\Delta x_i$
10: end for
```

3.2 Flip move selection

Flip moves are evaluated in Line 8 of Algorithm 1. Given that the move with the best value in Δx is selected by analyzing all such values, the process of considering all possible 1-flip moves takes $O(n)$ time. The selected move is then committed in Line 16 using Algorithm 3, which is based on Fact 1 and takes $O(n)$ time.

Algorithm 3 FLIP

```
Input: $n, Q, x, f_x, \Delta x, k$
Output: $x, f_x, \Delta x$
1: $f_x \leftarrow f_x + \Delta x_k$
2: $x_k \leftarrow 1 - x_k$
3: $\Delta x_k \leftarrow -\Delta x_k$
4: for $j \leftarrow 1$ to $k - 1$ do
5:     $\Delta x_j \leftarrow \Delta x_j - (1 - 2x_k)(1 - 2x_j) q_{kj}$
6: end for
7: for $j \leftarrow k + 1$ to $n$ do
8:     $\Delta x_j \leftarrow \Delta x_j - (1 - 2x_k)(1 - 2x_j) q_{jk}$
9: end for
```

3.3 The Tabu list

After performing a flip move, the inverse move (i.e., changing the value of the component back to 0 if it changed to 1, or back to 1 if it changed to 0) becomes forbidden for a number of iterations between $tt_{\min}$ and $tt_{\max}$, which is also called tabu tenure (Wang et al. 2012). This occurs in Line 15 of Algorithm 1. A counter to keep track of the remaining number of iterations for each variable is maintained in a vector labeled L, which is referred to as the tabu list. Each counter is decremented in Line 12.

Throughout this work, we always assume that the value of $tt_{\max} - tt_{\min}$ is a small nonnegative constant integer number unrelated to any other variable. For example, in (Glover et al. 2010; Wang et al. 2012), that value is always 10. Furthermore, we always assume that the time complexity involved in generating a pseudo-random number is constant.

This TS procedure also uses an aspiration rule, which states that a flip move may be chosen even when it is tabu, as long as it causes the current solution to become better than the incumbent solution. This rule is also used in (Palubeckis 2006; Glover et al. 2010; Kochenberger et al. 2011; Wang et al. 2012; Glover 2014; Samorani et al. 2019; Alidaee et al. 2017).

3.4 Local improvement

When a flip move results in an improvement over the incumbent solution, a local search procedure is performed. We describe it in Algorithm 4. Each iteration of this algorithm has asymptotic time complexity $O(n)$. From this point on, we may refer to Algorithm 4 as `LS-BASIC`.

Algorithm 4 LOCAL_SEARCH

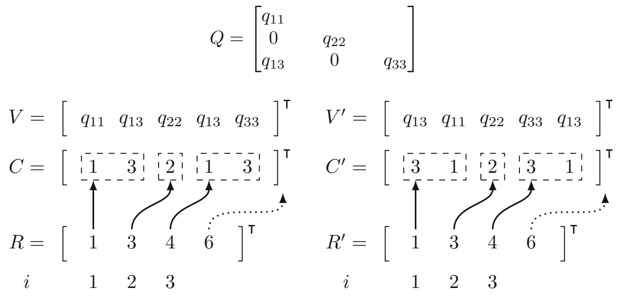
```
Input: $n, Q, x, f_x, \Delta x$
Output: $x, f_x, \Delta x$
1: improved $\leftarrow$ true
2: while improved do
3:     improved $\leftarrow$ false
4:     $k \leftarrow 1$
5:     for $j \leftarrow 2$ to $n$ do
6:         if $\Delta x_j < \Delta x_k$ then
7:             $k \leftarrow j$
8:         end if
9:     end for
10:    if $\Delta x_k < 0$ then
11:        $(x, f_x, \Delta x) \leftarrow$ FLIP$(n, Q, x, f_x, \Delta x, k)$
12:        improved $\leftarrow$ true
13:    end if
14: end while
```

In the next section, we propose a variant of Algorithm 1 for using a data structure specific for representing sparse matrices. We make use of some ideas first proposed by Glover and Hao (2010) in this regard.

4 TS using compressed sparse row representation

From Algorithms 1 to 4, we can observe that retrieving components in the Q matrix is only needed for initializing and updating the reevaluation vector. If Q is highly sparse, many such components have value zero, and based on Fact 1, they may not be needed throughout the execution of TS.

Moreover, there exists a pattern for accessing the components of the Q matrix in Algorithm 3, which is that given some index k, the components q_{kj} such that $j < k$ and the components q_{jk} such that $j > k$ are always accessed sequentially. Taking that into consideration, we may store the components of Q in a way that we are able to quickly retrieve only the nonzero components according to this access pattern. Keeping that in



$Q = \begin{bmatrix} q_{11} & & \\ 0 & q_{22} & \\ q_{13} & 0 & q_{33} \end{bmatrix}$
$V = [\, q_{11} \; q_{13} \; q_{22} \; q_{13} \; q_{33} \,]^{\mathsf{T}}$
$C = [\, 1 \; 3 \; 2 \; 1 \; 3 \,]^{\mathsf{T}}$
$R = [\, 1 \; 3 \; 4 \; 6 \,]^{\mathsf{T}}$
i 1 2 3
$V' = [\, q_{13} \; q_{11} \; q_{22} \; q_{33} \; q_{13} \,]^{\mathsf{T}}$
$C' = [\, 3 \; 1 \; 2 \; 3 \; 1 \,]^{\mathsf{T}}$
$R' = [\, 1 \; 3 \; 4 \; 6 \,]^{\mathsf{T}}$
i 1 2 3

Fig. 1 Example of two CSR representations (V, C, R) and (V', C', R') of a Q matrix

mind, we present, in Definition 5, a formal representation of a QUBO problem instance where the Q matrix is stored in a compressed sparse row format. This is a well-known format used for storing sparse matrices, and further details about it can be found in (Eisenstat et al. 1982; Buluç et al. 2009).

Definition 5 Let $I = (n, Q)$ be an instance of the QUBO problem, $N = \{1, 2, \ldots, n\}$, $\delta = \sum_{i=1}^{n} \deg i$, $V = [v_1, v_2, \ldots, v_\delta]^{\mathsf{T}} \in \mathbb{Q}^\delta$, $C = [c_1, c_2, \ldots, c_\delta]^{\mathsf{T}} \in \mathbb{Z}^\delta_{>0}$, and $R = [r_1, r_2, \ldots, r_{n+1}]^{\mathsf{T}} \in \mathbb{Z}^{n+1}_{>0}$. Then, the tuple (V, C, R) is a Compressed Sparse Row (CSR) representation of Q if:

1. $r_1 = 1$ and $r_{i+1} = r_i + \deg i$ for $i = 1, 2, \ldots, n$;
2. $\bigcup_{h=r_i}^{r_{i+1}-1} \{c_h\} = \{j \in N : (j \leq i \text{ and } q_{ij} \neq 0) \text{ or } (j > i \text{ and } q_{ji} \neq 0)\}$ for $i = 1, 2, \ldots, n$; and
3. $v_h = q_{ij}$ for $i = 1, 2, \ldots, n$, $h = r_i, r_i + 1, \ldots, r_{i+1} - 1$, and $j = c_h$.

It is possible to have multiple CSR representations of the same Q matrix, since each component of C may assume different values, as long as Definition 5 is satisfied. A particular algorithm that we use in our experiments to obtain a CSR representation of a Q matrix is described in Algorithm 12.

In Fig. 1 we present an example of two different CSR representations of the same Q matrix. Notice that, for $i \in N$, the value of r_i indicates the starting index of C that contains all values of $j \in N$ such that $q_{ij} \neq 0$ if $j \leq i$ and $q_{ji} \neq 0$ if $j > i$.

Glover and Hao (2010) also proposed a data structure for storing the nonzero components of the Q matrix. It is similar to the CSR representation in the sense that it may provide reduced memory usage for matrices of relatively high sparsity, and also allows performing 1-flip moves with the same time complexity. It is different from a CSR representation in the sense that it is based on storing an array of pairs of linked lists, one for each row and one for each column of Q.

Following, we present procedures for initializing and updating the reevaluation vector, considering that the instance matrix is in CSR format.

4.1 Maintaining the reevaluation vector

In Algorithm 5 we present a procedure for initializing a reevaluation vector when the coefficient matrix of the problem instance is in CSR format, described in Definition 5. It has asymptotic time complexity $O(n^2)$, and is equivalent to Algorithm 2. By equivalent, we mean that both algorithms return the same output when they receive the same input. In this case, we do not consider the change in the coefficient matrix format as a change in the input.

Algorithm 5 INITIALIZE_REEVALUATION_VECTOR_CSR

```
Input: $n, V, C, R, x$
Output: $\Delta x$
1: for $i \leftarrow 1$ to $n$ do
2:     for $h \leftarrow r_i$ to $r_{i+1} - 1$ do
3:         $j \leftarrow c_h$
4:         if $j = i$ then
5:             $\Delta x_j \leftarrow \Delta x_j + v_h$
6:         else
7:             $\Delta x_j \leftarrow \Delta x_j + v_h x_j$
8:         end if
9:     end for
10:    $\Delta x_i \leftarrow (1 - 2x_i)\Delta x_i$
11: end for
```

In Algorithm 6 we present a procedure for performing a 1-flip move when the coefficient matrix of the problem instance is in CSR format, described in Definition 5. It has asymptotic time complexity $O(\deg k)$, where k is its parameter. This algorithm is equivalent to Algorithm 3.

Algorithm 6 FLIP_CSR

```
Input: $V, C, R, x, f_x, \Delta x, k$
Output: $x, f_x, \Delta x$
1: $x_k \leftarrow 1 - x_k$
2: $f_x \leftarrow f_x + \Delta x_k$
3: for $h \leftarrow r_k$ to $r_{k+1} - 1$ do
4:     $j \leftarrow c_h$
5:     if $j = k$ then
6:         $\Delta x_j \leftarrow -\Delta x_j$
7:     else
8:         $\Delta x_j \leftarrow \Delta x_j - (1 - 2x_k)(1 - 2x_j)v_h$
9:     end if
10: end for
```

4.2 TS algorithm for `QUBO` considering `CSR` format

We notice that `LS-BASIC` and `TS-BASIC` (Algorithms 1 and 4) do not directly access components of the Q matrix. In fact, only Algorithm 3 does. Thus, in order to adapt

LS-BASIC and TS-BASIC to consider the Q matrix in CSR format, it suffices to have them use Algorithm 6 instead of Algorithm 3. We present the modified procedures, with respect to LS-BASIC and TS-BASIC, in Algorithms 7 and 8 . From this point on, we may refer to these algorithms as LS-CSR and TS-CSR.

Algorithm 7 LOCAL_SEARCH_CSR

```
Input: $n, V, C, R, x, f_x, \Delta x$
Output: $x, f_x, \Delta x$
1: improved ← true
2: while improved do
3:     improved ← false
4:     $k \leftarrow 1$
5:     for $j \leftarrow 2$ to $n$ do
6:         if $\Delta x_j < \Delta x_k$ then
7:             $k \leftarrow j$
8:         end if
9:     end for
10:    if $\Delta x_k < 0$ then
11:        $(x, f_x, \Delta x) \leftarrow$ FLIP_CSR$(V, C, R, x, f_x, \Delta x, k)$
12:        improved ← true
13:    end if
14: end while
```

Algorithm 8 TABU_SEARCH_CSR

```
Input: $n, V, C, R, x^*, f_x^*, \Delta x^*, tt_{\min}, tt_{\max}$
Output: $x^*, f_x^*, \Delta x^*$
1: $(x, f_x, \Delta x) \leftarrow (x^*, f_x^*, \Delta x^*)$
2: for $i \leftarrow 1$ to $n$ do
3:     $L_i \leftarrow 0$
4: end for
5: repeat
6:     $k \leftarrow 0$
7:     for $j \leftarrow 1$ to $n$ do
8:         if $(f_x + \Delta x_j < f_x^*$ or $L_j = 0)$ and $(k = 0$ or $\Delta x_j < \Delta x_k)$ then
9:             $k \leftarrow j$
10:        end if
11:        if $L_j > 0$ then
12:            $L_j \leftarrow L_j - 1$
13:        end if
14:    end for
15:    $L_k \leftarrow$ a random integer in the interval $[tt_{\min}, tt_{\max}]$
16:    $(x, f_x, \Delta x) \leftarrow$ FLIP_CSR$(V, C, R, x, f_x, \Delta x, k)$
17:    if $f_x < f_x^*$ then
18:        $(x, f_x, \Delta x) \leftarrow$ LOCAL_SEARCH_CSR$(n, V, C, R, x, f_x, \Delta x)$
19:        $(x^*, f_x^*, \Delta x^*) \leftarrow (x, f_x, \Delta x)$
20:    end if
21: until a stop condition has been fulfilled
```

A particular benefit of using a CSR representation of a Q matrix is reduced memory usage when the sparsity of Q is relatively high. Since it only stores the nonzero components of Q, it requires $O(nD)$ space, where D is given by Definition 2. However, the asymptotic time complexity of each iteration of `TS-CSR` (Algorithm 8) excluding local searches, and of each iteration of Algorithm 7, is still $O(n)$. Hence, the asymptotic time complexity of each iteration of `TS-CSR` is $O(n(1+m))$, where $m \in \mathbb{Z}_{\geq 0}$ is the number of local search iterations performed. In the next section, we propose additional data structures that are able to lower that.

5 TS using CSR representation and priority queues

In this section, we propose employing priority queues for speeding up implementations of TS, when solving instances of the `QUBO` problem using the CSR representation, stated in Definition 5.

5.1 The indexed priority queue data structure

Given a priority queue data structure (Cormen et al. 2001), it is possible to insert elements to it, or remove elements from it, in logarithmic time on the number of elements in the queue. It is also possible to change the priorities of the elements in logarithmic time, as long as one already knows their location in the queue. Here we formally define a priority queue that guarantees that.

In Definition 6, based on the definition of a priority queue data structure, we define what we call an indexed priority queue data structure, which additionally keeps track of the positions of its elements so that several operations can be performed relatively quickly. This data structure can be implemented, for example, using an array-backed binary heap and an auxiliary array that associates each element to its position in the heap. Whenever that position changes, the auxiliary array also must be updated. For implementation purposes, in Algorithms 13 to 22, we provide detailed descriptions of these operations.

Definition 6 Let $p = (n, m, h, z)$ be an indexed priority queue data structure, where $n \in \mathbb{Z}_{>0}$ determines that the possible stored elements belong to the set $\{1, 2, \dots, n\}$, $m \in \mathbb{Z}_{\geq 0}$ is the current number of stored elements, h is an array storing a minimum-heap that contains m elements from $\{1, 2, \dots, n\}$ with rational priority values and has capacity for n such elements, and z is an array of n integers such that the value of z_i is the position of i in the heap along with its priority if i is in the heap and zero otherwise. Furthermore, let $v \in \mathbb{Q}^n$ be an array such that, for $i \in \{1, 2, \dots, n\}$, if i is in p, then the value of v_i is the priority value of i in p. Then, p supports the operations:

1. IPQ_SIZE(p): retrieves the current number of elements in p, in $O(1)$ time;
2. IPQ_CONTAINS(p, i): checks whether p contains the element i, in $O(1)$ time;
3. IPQ_TOP(p): retrieves the element in p that has the lowest priority, in $O(1)$ time;
4. IPQ_INSERT(p, i, v): inserts i in p with priority v_i, in $O(\log \text{IPQ\_SIZE}(p))$ time;
5. IPQ_REMOVE(p, i, v): removes i from p, in $O(\log \text{IPQ\_SIZE}(p))$ time; and
6. IPQ_CHANGE_PRIORITY(p, i, v, new_value): changes the value of v_i to new_value and updates p accordingly, in $O(\log \text{IPQ\_SIZE}(p))$ time.

Following, we propose a data structure that makes use of indexed priority queues to make several operations performed throughout a TS procedure faster.

5.2 Storing TS data in indexed priority queues

Here we propose a structure composed of three indexed priority queues to store information regarding TS:

1. An indexed priority queue p^{Δ} stores the indices $i \in \{1, 2, \ldots, n\}$ that are not in the tabu list, and their priority values are the respective values of Δx_i. This allows finding the best nontabu flip move in $O(1)$ time;
2. An indexed priority queue $p^{L\Delta}$ stores the indices $i \in \{1, 2, \ldots, n\}$ that are in the tabu list, and their priority values are also the respective values of Δx_i. This allows checking whether a flip move meets the aspiration rule despite being tabu; and
3. An indexed priority queue p^{L} stores the indices $i \in \{1, 2, \ldots, n\}$ that are in the tabu list, and their priority values are the iteration numbers where they should leave the tabu list. This allows finding, in $O(1)$ time, the indices that should leave the tabu list the earliest, thus avoiding having to lower iteration counters for all n indices as done in TS-BASIC and TS-CSR.

In Definition 7, we formally describe these indexed priority queues and the data that they store.

Definition 7 Let $I = (n, Q)$ be a QUBO problem instance, $N = \{1, 2, \ldots, n\}$, $x \in \{0, 1\}^n$ a solution to I, Δx its reevaluation vector, and p^{Δ}, p^{L} and $p^{L\Delta}$ be indexed priority queues according to Definition 6. Then, we say that the tuple $(p^{\Delta}, p^{L}, p^{L\Delta})$ is a Tabu Search queue structure if, before each iteration of a Tabu Search procedure, the following properties hold:

1. The priority value of each $i \in N$ in p^{Δ}, and of each $i \in N$ in $p^{L\Delta}$, is the value of Δx_i;
2. The priority value of each $i \in N$ in p^{L} is the iteration number of the search where i should leave the tabu list;
3. IPQ_CONTAINS(p^{L}, i) if and only if i is currently tabu for each $i \in N$;
4. IPQ_CONTAINS(p^{L}, i) if and only if IPQ_CONTAINS($p^{L\Delta}, i$) for each $i \in N$; and
5. Either IPQ_CONTAINS(p^{L}, i) or IPQ_CONTAINS(p^{Δ}, i) for each $i \in N$.

Following, we propose algorithms for maintaining a Tabu Search queue structure throughout the execution of a TS procedure.

5.3 Updating TS data stored in indexed priority queues after a flip move

In TS-BASIC and TS-CSR (Algorithms 1 and 8), the tabu list L stores, for each variable, the number of iterations left for each variable to stop being tabu. In contrast, from Definition 7, item 2, we notice that, instead, we must keep track of a monotonically-increasing iteration counter, and have the tabu list store the exact iteration number when each variable will stop being tabu.

We propose Algorithm 9 for performing a 1-flip move, considering that the Q matrix is represented in CSR format, and also that the p^{Δ} and $p^{L\Delta}$ structures from Definition 7 are available. Lines 1, 2, and 9 to 11 take constant time each to execute. Lines 4 and 6 take $O(\log n)$ time to execute each since $\text{IPQ\_SIZE}(p^{\Delta}) + \text{IPQ\_SIZE}(p^{L\Delta}) = n$. The loop in Line 8 is executed for $\deg k$ iterations, and Lines 12 and 14 take $O(\log n)$ time each to execute. Thus, the entire loop, and also the entire algorithm, takes $O(\deg k \log n)$ time to execute.

Algorithm 9 FLIP_CSR- PQ

```
Input: $V, C, R, x, f_x, \Delta x, k, p^{\Delta}, p^{L\Delta}$
Output: $x, f_x, \Delta x, p^{\Delta}, p^{L\Delta}$
1: $x_k \leftarrow 1 - x_k$
2: $f_x \leftarrow f_x + \Delta x_k$
3: if IPQ_CONTAINS($p^{\Delta}, k$) then
4:     $(p^{\Delta}, \Delta x) \leftarrow$ IPQ_CHANGE_PRIORITY($p^{\Delta}, k, \Delta x, -\Delta x_k$)
5: else
6:     $(p^{L\Delta}, \Delta x) \leftarrow$ IPQ_CHANGE_PRIORITY($p^{L\Delta}, k, \Delta x, -\Delta x_k$)
7: end if
8: for $h \leftarrow r_k$ to $r_{k+1} - 1$ do
9:     $j \leftarrow c_h$
10:    if $j \neq k$ then
11:        if IPQ_CONTAINS($p^{\Delta}, j$) then
12:            $(p^{\Delta}, \Delta x) \leftarrow$ IPQ_CHANGE_PRIORITY($p^{\Delta}, j, \Delta x, \Delta x_j - (1 - 2x_k)(1 - 2x_j)v_h$)
13:        else
14:            $(p^{L\Delta}, \Delta x) \leftarrow$ IPQ_CHANGE_PRIORITY($p^{L\Delta}, j, \Delta x, \Delta x_j - (1 - 2x_k)(1 - 2x_j)v_h$)
15:        end if
16:    end if
17: end for
```

Based on Algorithm 9, we are now ready to present modified versions of the TS procedure and its local search procedure that use `CSR` matrix representations and indexed priority queues.

5.4 TS algorithm for `QUBO` considering `CSR` format and indexed priority queues

In Algorithm 10 we present a local search procedure that is equivalent to `LS-BASIC` and `LS-CSR` (Algorithms 4 and 7), and in Algorithm 11, a TS procedure that is equivalent to `TS-BASIC` and `TS-CSR` (Algorithms 1 and 8). These procedures use the CSR representation from Definition 5 and also maintain the queue structures presented in Definition 7. From this point on, we may refer to Algorithms 10 and 11 as `LS-CSR-PQ` and `TS-CSR-PQ`. As follows, first we analyze the asymptotic time complexity of Algorithm 11.

Every iteration of `TS-CSR-PQ` (Algorithm 11) that does not perform a local search has asymptotic time complexity $O(D \log n)$. This can be verified as follows. In Lines 13 to 19, the index of the component to flip is selected, which is either that with the minimum priority in p^{Δ}, or that with the minimum priority in $p^{L\Delta}$ if it falls under the aspiration rule. This selection takes $O(1)$ time. In Lines 20 to 29, the iteration counter `iter` is incremented, and thus indices in p^L with priority value less than the iteration

Algorithm 10 LOCAL_SEARCH_CSR- PQ

```
Input: $V, C, R, x, f_x, \Delta x, p^{\Delta}, p^{L\Delta}$
Output: $x, f_x, \Delta x, p^{\Delta}, p^{L\Delta}$
1: improved ← true
2: while improved do
3:     improved ← false
4:     $i \leftarrow$ IPQ_TOP($p^{\Delta}$)
5:     if IPQ_SIZE($p^{L\Delta}$) $> 0$ then
6:         $i' \leftarrow$ IPQ_TOP($p^{L\Delta}$)
7:         if $\Delta x_{i'} < \Delta x_i$ or ($\Delta x_{i'} = \Delta x_i$ and $i' < i$) then
8:             $i \leftarrow i'$
9:         end if
10:    end if
11:    if $\Delta x_i < 0$ then
12:        $(x, f_x, \Delta x, p^{\Delta}, p^{L\Delta}) \leftarrow$ FLIP_CSR- PQ($V, C, R, x, f_x, \Delta x, i, p^{\Delta}, p^{L\Delta}$)
13:        improved ← true
14:    end if
15: end while
```

counter must be removed. Since we specified that $tt_{\max} - tt_{\min}$ is a small nonnegative constant number, we also have that, in the worst case, a constant number of indices will be removed per iteration. Thus, this process takes $O(\log n)$ time. In Lines 30 to 38, the selected index is added to the tabu list by setting a priority value in p^L greater than or equal to `iter`. Based on Definition 7, IPQ_SIZE(p^{Δ}) + IPQ_SIZE(p^L) $= n$. Thus, the processes described in Lines 20 to 29 and Lines 30 to 38 take $O(\log n)$ time each. Finally, in Line 39, the selected flip move is committed, which takes $O(\deg k \log n)$ time. Since we are considering iterations that do not perform local searches, we do not consider Lines 40 to 43. Thus, in total, each iteration that does not perform a local search takes $O(\deg k \log n)$ time. From Definition 2, $D = \max_{k \in \{1,2,\cdots,n\}} \deg k$. Hence, the asymptotic time complexity of each iteration of `TS-CSR-PQ` (Algorithm 11), excluding local searches, is also $O(D \log n)$.

By analyzing `LS-CSR-PQ` (Algorithm 10) in the same way, we have that each iteration of that algorithm has asymptotic time complexity $O(D \log n)$. This means that each iteration of `TS-CSR-PQ` (Algorithm 11) has asymptotic time complexity $O(D \log n(1+m))$, where $m \in \mathbb{Z}_{\geq 0}$ is the number of local search iterations performed.

5.5 Conditions under which `TS-CSR-PQ` is faster

Now, we discuss a necessary condition for each iteration of `TS-CSR-PQ` (Algorithm 11) to be asymptotically faster than `TS-BASIC` and `TS-CSR` (Algorithms 1 and 8), which we formally state in Proposition 1. Before we go into the proof of Proposition 1, recall the little-o notation in Definition 4.

Let $I = (n, Q)$ be an instance of the `QUBO` problem and $\text{spr}(I)$ the sparsity of Q as defined in Definition 3. If there exist constants $c > 0$ and $n_0 > 0$ such that $D \log n(1+m) < cn(1+m)$, and thus $D < c\frac{n}{\log n}$, for all $n \geq n_0$, then the asymptotic time complexity of each iteration of `TS-CSR-PQ` (Algorithm 11) is $o(n(1+m))$, which is an improvement over that of `TS-BASIC` and `TS-CSR` (Algorithms 1 and 8).

Algorithm 11 TABU_SEARCH_CSR- PQ

```
Input: $n, V, C, R, x^*, f_x^*, \Delta x^*, tt_{\min}, tt_{\max}$
Output: $x^*, f_x^*, \Delta x^*$
1: $(x, f_x, \Delta x) \leftarrow (x^*, f_x^*, \Delta x^*)$
2: for $i \leftarrow 1$ to $n$ do
3:     $L_i \leftarrow 0$
4: end for
5: iter $\leftarrow 0$
6: $p^{\Delta} \leftarrow$ IPQ_INITIALIZE($n$)
7: $p^{L} \leftarrow$ IPQ_INITIALIZE($n$)
8: $p^{L\Delta} \leftarrow$ IPQ_INITIALIZE($n$)
9: for $i \leftarrow 1$ to $n$ do
10:     $p^{\Delta} \leftarrow$ IPQ_INSERT($p^{\Delta}, i, \Delta x$)
11: end for
12: repeat
13:     $k \leftarrow$ IPQ_TOP($p^{\Delta}$)
14:     if IPQ_SIZE($p^{L\Delta}$) $> 0$ then
15:         $k' \leftarrow$ IPQ_TOP($p^{L\Delta}$)
16:         if $f_x + \Delta x_{k'} < f_x^*$ and ($\Delta x_{k'} < \Delta x_k$ or ($\Delta x_{k'} = \Delta x_k$ and $k' < k$)) then
17:             $k \leftarrow k'$
18:         end if
19:     end if
20:     iter $\leftarrow$ iter $+ 1$
21:     while IPQ_SIZE($p^{L}$) $> 0$ do
22:         $i \leftarrow$ IPQ_TOP($p^{L}$)
23:         if iter $< L_i$ then
24:             break
25:         end if
26:         $p^{L} \leftarrow$ IPQ_REMOVE($p^{L}, i, L$)
27:         $p^{L\Delta} \leftarrow$ IPQ_REMOVE($p^{L\Delta}, i, \Delta x$)
28:         $p^{\Delta} \leftarrow$ IPQ_INSERT($p^{\Delta}, i, \Delta x$)
29:     end while
30:     $L_k \leftarrow$ iter+ a random integer in the interval $[tt_{\min}, tt_{\max}]$
31:     if IPQ_CONTAINS($p^{\Delta}, k$) then
32:         $p^{\Delta} \leftarrow$ IPQ_REMOVE($p^{\Delta}, k, \Delta x$)
33:         $p^{L} \leftarrow$ IPQ_INSERT($p^{L}, k, L$)
34:         $p^{L\Delta} \leftarrow$ IPQ_INSERT($p^{L\Delta}, k, \Delta x$)
35:     else
36:         $p^{L} \leftarrow$ IPQ_REMOVE($p^{L}, k, L$)
37:         $p^{L} \leftarrow$ IPQ_INSERT($p^{L}, k, L$)
38:     end if
39:     $(x, f_x, \Delta x, p^{\Delta}, p^{L\Delta}) \leftarrow$ FLIP_CSR- PQ($V, C, R, x, f_x, \Delta x, k, p^{\Delta}, p^{L\Delta}$)
40:     if $f_x < f_x^*$ then
41:         $(x, f_x, \Delta x, p^{\Delta}, p^{L\Delta}) \leftarrow$ LOCAL_SEARCH_CSR- PQ($V, C, R, x, f_x, \Delta x, p^{\Delta}, p^{L\Delta}$)
42:         $(x^*, f_x^*, \Delta x^*) \leftarrow (x, f_x, \Delta x)$
43:     end if
44: until a stop condition has been fulfilled
```

This imposes that the sparsity of the instance matrix must be greater than $1 - \frac{c}{\log n}$. This can be verified as follows. From Definitions 2 and 3, we have that $1 - \text{spr}(I) = \frac{\sum_{i=1}^{n} \deg i}{n^2} \leq \frac{\sum_{i=1}^{n} D}{n^2} = \frac{nD}{n^2} = \frac{D}{n}$. Consequently, $n(1 - \text{spr}(I)) \leq D$. Thus, if $D < c\frac{n}{\log n}$, then $n(1 - \text{spr}(I)) \leq D < c\frac{n}{\log n}$. This implies that $1 - \text{spr}(I) < \frac{c}{\log n}$, and hence $\text{spr}(I) > 1 - \frac{c}{\log n}$.

Proposition 1 *Let $\mathcal{I}$ be the set of all instances of the* `QUBO` *problem,* $I = (n, Q) \in \mathcal{I}$*, and* $c > 0$ *be a constant number. Each iteration of* `TS-CSR-PQ` *(Algorithm 11) is asymptotically faster than each iteration of* `TS-BASIC` *and* `TS-CSR` *(Algorithms 1 and 8) when* $\operatorname{spr}(I) > 1 - \frac{c}{\log n}$.

Finally, we remark that our proposed priority queues could also be employed in the TS procedure described in Algorithm 1 (`TS-BASIC`), which uses a full matrix representation. Such a variant could be referred to as, for example, `TS-PQ`. In order to update the reevaluation vector using a full matrix representation, $O(n)$ time is required, as described in Algorithm 3. Employing only the priority queues without using a CSR representation would not reduce that time complexity. This means that each iteration of a `TS-PQ` would still have asymptotic time complexity $O(n)$. Hence, it would not be an improvement with respect to the state of the art.

5.6 A publicly available implementation of `TS-CSR` and `TS-CSR-PQ`

For the purpose of allowing our theoretical results to be used in applications, we implemented the algorithms `TS-CSR` and `TS-CSR-PQ` in `C` programming language and made them publicly available as a library. The repository containing the source codes and documentation is available at the URL https://github.com/rliang/libtsqubo.

The library is a so-called header-only library. This means that, in order to use it in `C` or `C++` programs, it suffices to download and include the header file in the sources. By default, the library implements the `TS-CSR` algorithm. When using the compile-time flag `TSQUBO_SPARSE`, the library implements the `TS-CSR-PQ` algorithm.

In the next section, we report on experimental comparisons of the three TS procedures `TS-BASIC`, `TS-CSR` and `TS-CSR-PQ`.

6 Computational experiments

In this section, we report on computational experiments with the intent to analyze the ideas proposed in Sects. 4 and 5. From this point on, when we refer to Algorithms 1, 8, and 11, `TS-BASIC`, `TS-CSR`, and `TS-CSR-PQ` (Algorithms 1, 8, and 11) we are referring to our implementations of these algorithms. In Sect. 6.3, we measure the processing times of `TS-BASIC`, `TS-CSR`, and `TS-CSR-PQ` when solving benchmark instances. In Sect. 6.4, we measure the processing times of `TS-BASIC`, `TS-CSR`, and `TS-CSR-PQ` when solving randomly-generated instances with varied sparsity. In Sect. 6.5, we measure the quality of solutions obtained by a Path Relinking procedure, at different elapsed times, when using each TS implementation.

6.1 Computing environment

All experiments were executed on a computer with processor Intel® Core™ i7–8500U (6 cores of 2,70GHz and 8 GB of RAM), running Linux Ubuntu 20.04 LTS, 64 bits. All algorithms were implemented in `C++` programming language and compiled using GNU `g++` 10.1 with compiler flags `-Ofast -mtune=native -s`. For generating pseudo-random numbers, we chose to use the C++ class `std::mt19937`, which implements a Mersenne Twister pseudo-random number generator. It can be verified that, considering all variables in our implemented algorithms, this generator takes constant time to generate a single pseudo-random number (Matsumoto and Nishimura 1998). It also has a period of $2^{19937} - 1$.

6.2 Benchmark instances

We conducted experiments using a total of 113 benchmark QUBO problem instances. They were obtained from four datasets, from where we selected only the instances with $n \geq 1000$. These instances have been commonly used in the literature surrounding the QUBO problem (Wang et al. 2012; Dunning et al. 2018; Alidaee et al. 2017). We describe each dataset as follows:

1. ORLib: This is a set of maximization QUBO problem instances available at the OR-Library website (http://people.brunel.ac.uk/~mastjjb/jeb/orlib/bqpinfo.html). These instances have matrix sparsity of approximately 90%. Each instance is labeled bqp$k$.n, where n is the number of variables and k is the index of the instance.
2. Stanford Max-Cut: This is a set of Maximum Cut problem instances which we reformulated to the form of the QUBO problem according to the following ideas described by Kochenberger et al. (2011). They are available at the website http://www.stanford.edu/~yyye/yyye/Gset. These instances have matrix sparsities ranging from 97.8 to 99.97%. Each instance is labeled Gk.n, where k is the index of the instance and n is the number of vertices of the graph.
3. Optsicom Max-Cut: This is another set of Maximum Cut problem instances, available at the website http://grafo.etsii.urjc.es/optsicom/maxcut. These instances have matrix sparsities ranging from 99.75% to 99.92%. Each instance is labeled either sg3dl$k$.n or tour$k$.n, where k is the index of the instance and n is the number of vertices of the graph.
4. Palubeckis: This is a set of QUBO problem instances which was originally generated by Palubeckis (2004, 2006), whose generator is available at the website https://www.personalas.ktu.lt/~ginpalu/. These instances have matrix sparsities ranging from 0% to 50%. Each instance is labeled pk.n, where n is the number of variables and k is the index of the instance.

6.3 Measurements of processing times using benchmark instances

Here we present measurements of processing time of TS-BASIC, TS-CSR, and TS-CSR-PQ (Algorithms 1, 8, and 11) when solving the benchmark instances of the QUBO problem. The improvements in processing time are quantified using a measure of speedup, which is defined as follows. Let A and B be two programs used to solve the same problem, where A and B take time t_a and t_b, respectively. Then, we say that B is $\frac{t_a}{t_b}$ times faster than A. We also say that t_a is the reference value. In our experiments, we always measure processing time in seconds.

We report our experimental results for all tested benchmark instances in Tables 1, 2, 3 and 4. For each tested instance, we executed each TS implementation for $500n$ iterations, and tested for the parameters $tt_{\text{min}} = 1\%$ of n and $tt_{\text{min}} = 10\%$ of n, where tt_{max} was always fixed at $tt_{\text{min}} + 10$. These values have been frequently used in the literature for these instances (Palubeckis 2004, 2006; Glover et al. 2010; Wang et al. 2012). We repeated this process for 100 randomly-generated starting solutions, and present the measurements as average values.

In Tables 1 and 2, we observe that `TS-CSR-PQ` (Algorithm 11) provided relatively large speedup values, of up to around 200. In general, these speedup values increased as the instance matrix sparsity increased. In contrast, the speedup values of `TS-CSR` (Algorithm 8) were of up to around 5. As for the results in Table 3, we see that the speedup values of `TS-CSR-PQ` were close to those of `TS-CSR`, and both were still faster than `TS-BASIC` (Algorithm 1). Finally, in Table 4, we observe that `TS-CSR-PQ` was slower than `TS-BASIC` in all cases, and `TS-CSR` was slower than `TS-BASIC` for values of sparsity below 50%. This was expected, since these instances have relatively low sparsity.

From the results presented in Tables 1 to 4, we also notice that the speedup of `TS-CSR` (Algorithm 8) slightly decreased as the value of tt_{min} increased, whereas the speedup of `TS-CSR-PQ` (Algorithm 11) was almost unaffected. Although larger values of tt_{min} have not been commonly used in the literature, we conducted experiments using more values of tt_{min} in order to further study the impact of this parameter. We conducted experiments using values of tt_{min} from 1% of n to 99% of n, and present our measurements in Figs. 2 and 3.

From the results shown in Figs. 2 and 3, we observe that the measured speedup values were consistent with those shown in Tables 1 to 4 when the same values of tt_{min} were used. As for the remaining values of tt_{min}, the speedup behavior varied from instance to instance. In particular, for the instance `bqp2.2500`, setting $tt_{\text{min}} \geq 70\%$ of n caused `TS-CSR-PQ` (Algorithm 11) to become slower than `TS-CSR` (Algorithm 8), but not for any other tested instance.

From the experimental results presented thus far, we observed that using `TS-CSR-PQ` (Algorithm 11) is worthwhile for the instances shown in Tables 1 to 3, which have relatively high sparsity, and not worthwhile for the instances shown in Table 4, which have relatively low sparsity. However, from these results alone, we could not identify a specific threshold of sparsity where `TS-CSR-PQ` becomes faster or slower than `TS-CSR` (Algorithm 8), since the `QUBO` benchmark instances' matrices have either relatively high or low sparsity. To overcome this issue, in the next subsection, we report on further computational experiments using randomly-generated instances of more varied sparsity.

6.4 Measurements of processing time using randomly-generated instances

With the intent to determine for which values of instance matrix sparsity it is worthwhile to use `TS-CSR-PQ` (Algorithm 11) over `TS-CSR` and `TS-BASIC` (Algorithms 1 and 8), here we report on additional measurements of processing time using randomly-generated instances.

Table 1 Speedups of implementations of `TS-CSR` (Algorithm 8) and `TS-CSR-PQ` (Algorithm 11) with respect to `TS-BASIC` (Algorithm 1)

Max-cut instance	spr (%)	$tt_{\min} = 1\%$ of n			$tt_{\min} = 10\%$ of n		
		`TS-BASIC`	`TS-CSR`	`TS-CSR-PQ`	`TS-BASIC`	`TS-CSR`	`TS-CSR-PQ`
		Time	Speedup		Time	Speedup	
G43.1000	97.90	3.24	1.16	4.24	5.10	1.28	4.45
G44.1000	97.90	3.20	1.17	4.51	5.09	1.27	4.42
G45.1000	97.90	3.10	1.16	4.29	4.96	1.28	4.44
G46.1000	97.90	3.21	1.14	4.23	5.09	1.28	4.44
G47.1000	97.90	3.19	1.16	4.38	4.45	1.28	4.44
G51.1000	98.72	3.01	1.06	7.36	6.30	1.26	6.68
G52.1000	98.72	3.04	1.06	7.18	6.50	1.26	6.69
G53.1000	98.72	3.05	1.06	7.28	6.20	1.26	6.66
G54.1000	98.72	3.03	1.06	7.48	7.22	1.26	6.59
G22.2000	98.95	11.42	1.50	11.25	12.48	1.49	9.91
G23.2000	98.95	11.20	1.49	10.79	12.25	1.49	9.95
G24.2000	98.95	11.34	1.48	10.91	12.45	1.48	9.93
G25.2000	98.95	11.44	1.49	10.95	12.73	1.49	9.89
G26.2000	98.95	11.53	1.49	10.93	12.52	1.48	9.95
G27.2000	98.95	11.32	1.47	10.82	12.69	1.46	9.58
G28.2000	98.95	11.37	1.46	10.42	12.37	1.43	9.33
G29.2000	98.95	11.27	1.45	10.68	12.55	1.43	9.43
G30.2000	98.95	11.45	1.44	10.69	12.52	1.43	9.32

Table 1 continued

Max-cut instance	spr (%)	$tt_{\min} = 1\%$ of n			$tt_{\min} = 10\%$ of n		
		`TS-BASIC`	`TS-CSR`	`TS-CSR-PQ`	`TS-BASIC`	`TS-CSR`	`TS-CSR-PQ`
		Time	Speedup		Time	Speedup	
G31.2000	98.95	11.46	1.45	10.55	12.72	1.46	9.37
G32.2000	99.75	11.39	1.30	21.68	11.90	1.38	19.16
G33.2000	99.75	11.38	1.32	21.88	11.90	1.38	19.37
G34.2000	99.75	11.39	1.31	21.68	11.95	1.38	19.19
G35.2000	99.36	11.33	1.39	19.04	12.29	1.43	14.88
G36.2000	99.36	11.39	1.40	17.78	12.48	1.44	14.61
G37.2000	99.36	11.37	1.39	18.82	11.98	1.43	14.76
G38.2000	99.36	11.22	1.39	18.65	12.50	1.43	14.72
G39.2000	99.36	11.28	1.40	19.37	12.21	1.45	15.11
G40.2000	99.36	11.53	1.41	19.25	12.37	1.44	14.88
G41.2000	99.36	11.21	1.40	19.49	12.44	1.45	14.96
G42.2000	99.36	11.33	1.40	19.47	12.26	1.45	14.98
G48.3000	99.83	25.38	0.74	13.24	26.40	0.77	13.11
G49.3000	99.83	25.48	0.77	15.63	26.36	0.78	13.29
G50.3000	99.83	25.18	0.79	16.85	26.58	0.84	13.77
G55.5000	99.88	71.89	1.49	71.34	80.28	1.50	49.60
G56.5000	99.88	70.66	1.49	70.06	79.96	1.50	48.87
G57.5000	99.90	70.97	1.27	56.83	77.05	1.29	50.27
G58.5000	99.74	71.57	1.49	43.58	82.24	1.51	35.83
G59.5000	99.74	68.65	1.54	46.68	72.03	1.56	38.25
G60.7000	99.92	125.11	1.49	96.56	142.04	1.50	66.39

Table 1 continued

Max-cut instance	spr (%)	$tt_{\min} = 1\%$ of n			$tt_{\min} = 10\%$ of n		
		TS-BASIC	TS-CSR	TS-CSR-PQ	TS-BASIC	TS-CSR	TS-CSR-PQ
		Time	Speedup		Time	Speedup	
G61.7000	99.92	126.31	1.49	97.55	141.11	1.49	65.49
G62.7000	99.93	125.41	1.24	78.72	139.83	1.25	67.78
G63.7000	99.82	126.53	1.47	61.77	150.50	1.51	49.16
G64.7000	99.82	126.49	1.55	62.31	143.89	1.58	52.17
G65.8000	99.94	164.19	1.22	88.40	183.76	1.24	76.62
G66.9000	99.94	210.00	1.21	99.35	231.85	1.23	85.37
G67.10000	99.95	259.83	1.21	109.38	287.61	1.23	92.71
G70.10000	99.97	258.10	1.22	202.73	293.71	1.26	166.70
G72.10000	99.95	439.29	1.21	110.18	650.74	1.23	93.19
G77.14000	99.96	716.75	1.19	152.03	1326.83	1.22	123.68
G81.20000	99.97	1803.66	1.15	202.11	3345.34	1.19	157.57

Table 2 Speedups of implementations of TS-CSR (Algorithm 8) and TS-CSR-PQ (Algorithm 11) with respect to TS-BASIC (Algorithm 1)

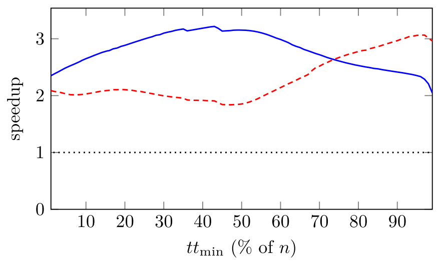
Max-cut instance	spr (%)	$tt_{\min} = 1\%$ of n			$tt_{\min} = 10\%$ of n		
		TS-BASIC	TS-CSR	TS-CSR-PQ	TS-BASIC	TS-CSR	TS-CSR-PQ
		Time	Speedup		Time	Speedup	
sg3dl101000.1000	99.30	1.81	2.12	7.83	2.41	2.70	8.72
sg3dl102000.1000	99.30	1.89	2.14	8.86	2.44	2.73	8.78
sg3dl103000.1000	99.30	1.82	2.10	7.77	2.51	2.81	9.03
sg3dl104000.1000	99.30	1.91	2.16	8.63	2.55	2.87	9.21
sg3dl105000.1000	99.30	1.92	2.26	8.10	2.47	2.79	8.90
sg3dl106000.1000	99.30	1.91	2.25	9.53	2.47	2.75	9.01
sg3dl107000.1000	99.30	1.93	2.25	8.17	2.41	2.71	8.83
sg3dl108000.1000	99.30	1.91	2.21	8.20	2.44	2.58	8.95
sg3dl109000.1000	99.30	1.84	1.95	7.98	2.42	2.71	8.82
sg3dl1010000.1000	99.30	1.84	2.10	9.05	2.43	2.71	8.81
sg3dl141000.2744	99.74	23.57	3.01	36.42	24.73	3.52	30.50
sg3dl142000.2744	99.74	24.46	3.04	36.64	25.47	3.54	31.94
sg3dl143000.2744	99.74	23.85	2.93	34.82	24.51	3.45	30.16
sg3dl144000.2744	99.74	23.34	2.81	34.06	24.47	3.42	30.08
sg3dl145000.2744	99.74	23.67	2.95	35.70	24.39	3.04	30.50
sg3dl146000.2744	99.74	23.76	3.00	37.39	24.76	3.50	30.50
sg3dl147000.2744	99.74	23.62	2.88	34.95	24.29	3.51	30.16
sg3dl148000.2744	99.74	23.68	2.96	37.27	24.33	3.50	30.26
sg3dl149000.2744	99.74	23.73	3.04	34.91	24.55	3.44	30.61
sg3dl1410000.2744	99.74	23.75	2.83	34.65	24.70	3.52	31.41
toursg3-15.3375	99.79	32.80	4.52	47.32	36.60	3.59	37.73
tourspm3-15-50.3375	99.92	32.67	4.79	48.87	32.92	5.00	74.67

Table 3 Speedups of implementations of `TS-CSR` (Algorithm 8) and `TS-CSR-PQ` (Algorithm 11) with respect to `TS-BASIC` (Algorithm 1)

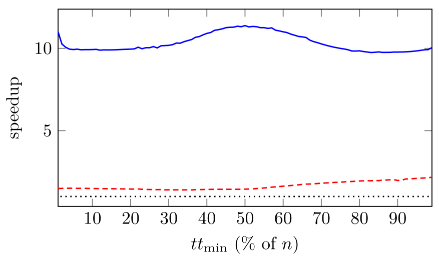
`QUBO` instance	spr (%)	$tt_{\text{min}} = 1\%$ of n			$tt_{\text{min}} = 10\%$ of n		
		`TS-BASIC`	`TS-CSR`	`TS-CSR-PQ`	`TS-BASIC`	`TS-CSR`	`TS-CSR-PQ`
		Time	Speedup		Time	Speedup	
bqp1.1000	90.10	2.63	1.61	1.69	4.25	1.73	1.88
bqp2.1000	90.10	2.75	1.64	1.73	4.86	1.76	1.92
bqp3.1000	90.05	2.82	1.66	1.74	4.32	1.74	1.90
bqp4.1000	90.07	3.11	1.66	1.74	4.63	1.74	1.91
bqp5.1000	90.09	2.77	1.69	1.77	4.11	1.78	1.94
bqp6.1000	90.02	2.73	1.70	1.77	4.14	1.77	1.93
bqp7.1000	90.11	5.23	1.67	1.75	4.11	1.77	1.95
bqp8.1000	90.10	2.97	1.68	1.75	4.18	1.77	1.92
bqp9.1000	90.09	2.76	1.71	1.81	4.35	1.78	1.95
bqp10.1000	90.18	4.14	1.70	1.80	5.04	1.79	1.96
bqp1.2500	90.06	25.22	2.05	2.42	30.80	2.01	2.53
bqp2.2500	90.10	25.76	2.07	2.45	31.51	2.03	2.56
bqp3.2500	90.11	27.35	2.13	2.52	30.92	2.08	2.62
bqp4.2500	90.10	24.81	2.04	2.42	30.79	2.00	2.53
bqp5.2500	90.10	25.07	2.02	2.38	30.96	2.01	2.48
bqp6.2500	90.12	24.93	2.07	2.39	31.16	2.02	2.50
bqp7.2500	90.08	25.53	2.08	2.40	31.31	2.04	2.51
bqp8.2500	90.11	26.22	2.10	2.45	30.85	2.00	2.46
bqp9.2500	90.08	25.69	2.00	2.32	31.16	1.97	2.42
bqp10.2500	90.10	25.98	2.10	2.48	31.01	2.06	2.57

Table 4 Speedups of implementations of `TS-CSR` (Algorithm 8) and `TS-CSR-PQ` (Algorithm 11) with respect to `TS-BASIC` (Algorithm 1)

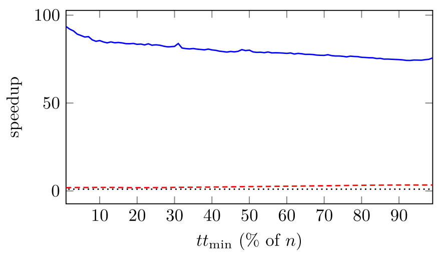
`QUBO` instance	spr (%)	$tt_{\min} = 1\%$ of n			$tt_{\min} = 10\%$ of n		
		`TS-BASIC`	`TS-CSR`	`TS-CSR-PQ`	`TS-BASIC`	`TS-CSR`	`TS-CSR-PQ`
		Time	Speedup		Time	Speedup	
p1.3000	50	27.70	1.29	0.52	26.34	1.30	0.57
p2.3000	20	29.36	0.91	0.33	25.85	0.94	0.36
p3.3000	20	33.63	0.91	0.33	25.83	0.94	0.36
p4.3000	0	32.42	0.90	0.27	29.01	0.92	0.29
p5.3000	0	36.89	0.90	0.27	29.40	0.92	0.30
p1.4000	50	69.30	1.34	0.54	46.31	1.35	0.59
p2.4000	20	86.32	0.96	0.35	49.02	0.98	0.39
p3.4000	20	71.65	0.96	0.35	51.08	0.99	0.39
p4.4000	0	62.79	0.88	0.28	53.68	0.92	0.31
p5.4000	0	73.66	0.89	0.28	57.16	0.92	0.31
p1.5000	50	143.42	1.38	0.56	83.95	1.37	0.62
p2.5000	20	119.08	0.98	0.36	87.60	1.00	0.40
p3.5000	20	105.18	0.98	0.36	77.59	1.01	0.40
p4.5000	0	110.45	0.89	0.29	79.10	0.91	0.32
p5.5000	0	146.60	0.88	0.29	76.11	0.92	0.32
p1.6000	50	145.30	1.39	0.57	110.61	1.38	0.63
p2.6000	20	179.32	1.00	0.37	113.69	1.02	0.41
p3.6000	0	161.21	0.88	0.30	109.97	0.91	0.33
p1.7000	50	243.39	1.40	0.58	172.26	1.39	0.64
p2.7000	20	275.54	1.01	0.37	153.68	1.02	0.41
p3.7000	0	269.41	0.88	0.30	161.30	0.90	0.33



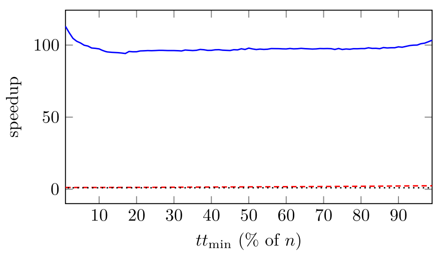
(a) instance `bqp2.2500`



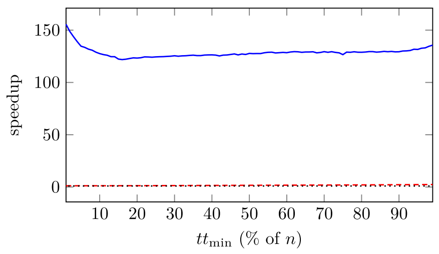
(b) instance `G22.2000`



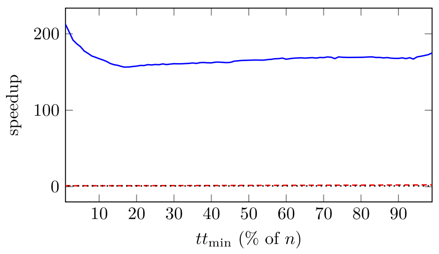
(c) instance `G65.8000`



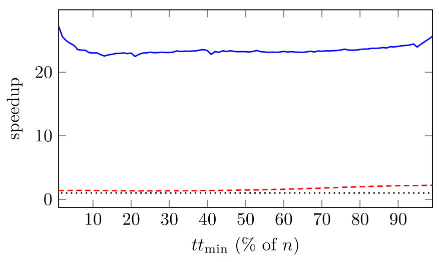
(d) instance `G67.10000`



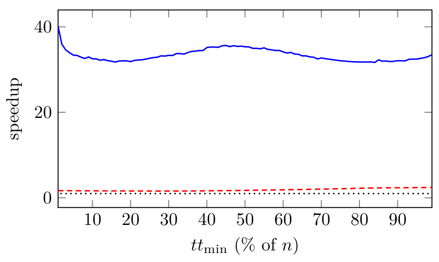
(e) instance `G77.14000`



(f) instance `G81.20000`



(g) instance `sg3dl1410000.2744`



(h) instance `toursg3-15.3375`

Fig. 2 Speedups of implementations of `TS-CSR` (Algorithm 8, red dashed curve) and `TS-CSR-PQ` (Algorithm 11, blue continuous curve) with respect to `TS-BASIC` (Algorithm 1) (Color figure online)

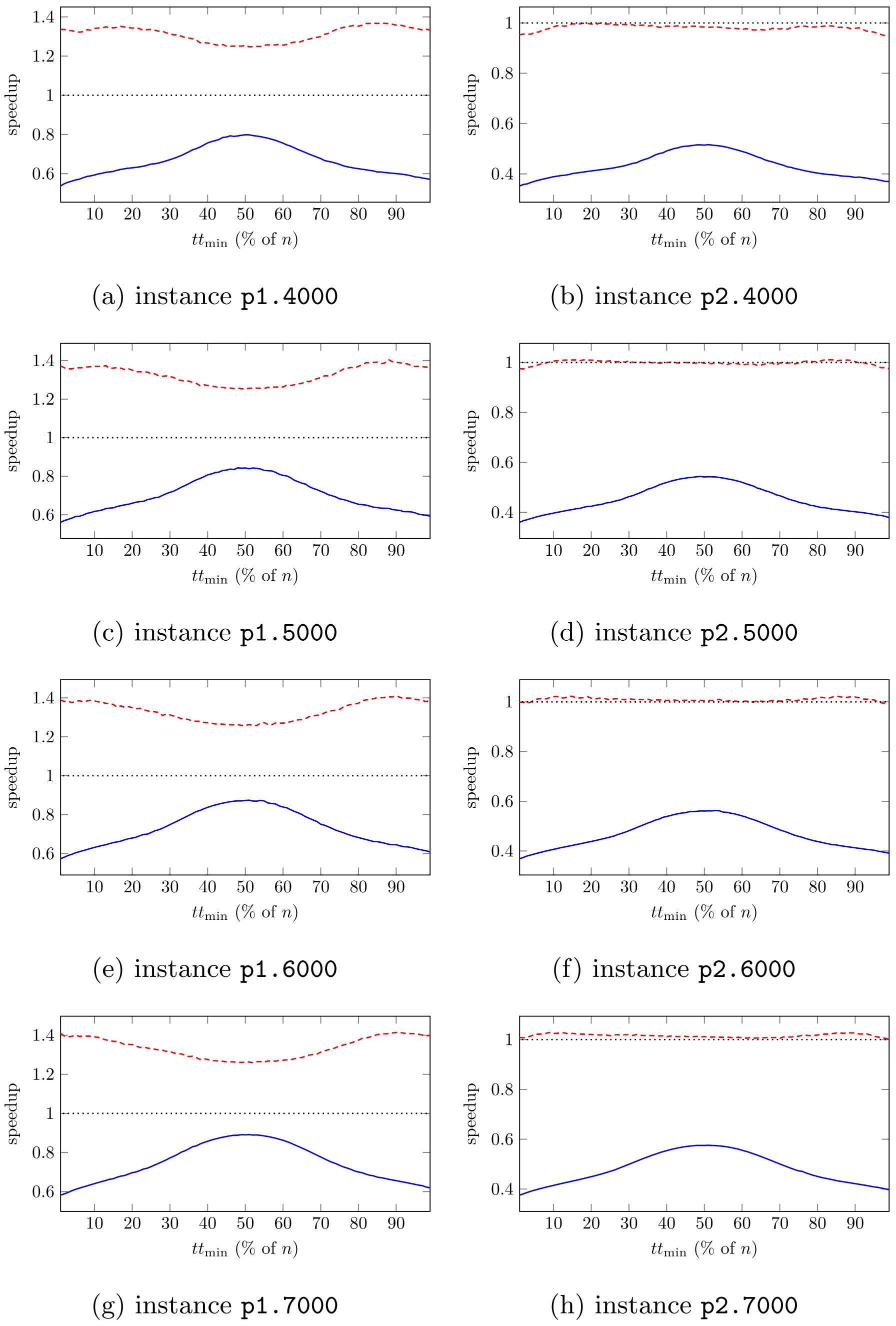


Fig. 3 Speedups of implementations of TS-CSR (Algorithm 8, red dashed curve) and TS-CSR-PQ (Algorithm 11, blue continuous curve) with respect to TS-BASIC (Algorithm 1) (Color figure online)

We used the generator proposed by Pardalos and Rodgers (1990) named Q01MKS, which is specialized for generating QUBO problem instances with matrices of varied sparsity. We generated instances with varying values of n, and values of matrix sparsity from 0% to 95%. The values of the coefficients of the matrix were in the integer interval $[-100, 100]$.

For each generated instance, we executed TS-CSR and TS-CSR-PQ (Algorithms 8 and 11) for $500n$ iterations and using the parameters $tt_{\min} = 1\%$ of n, $tt_{\min} = 10\%$ of n, $tt_{\min} = 50\%$ of n and $tt_{\min} = 75\%$ of n. We present our results in Figs. 4 and 5.

From the measurements shown in Figs. 4 and 5, we observe that the speedups of TS-CSR and TS-CSR-PQ (Algorithms 8 and 11) with respect to TS-BASIC (Algorithm 1) increased as the matrix sparsity increased, which is consistent with the measurements shown in Sect. 6.3. Additionally, as the value of n increased, the speedup of TS-CSR decreased, and the speedup of TS-CSR-PQ increased. This is expected from the analyses of asymptotic time complexities of each iteration of TS-BASIC, TS-CSR, and TS-CSR-PQ. Finally, similarly to Figs. 2 and 3, we found that the value of $tt_{\min}$ influenced slightly the speedups.

Regarding sparsity thresholds, we found that TS-CSR-PQ (Algorithm 11) was faster than TS-BASIC (Algorithm 1) for values of sparsity ranging approximately from 60% to 80%, and faster than TS-CSR (Algorithm 8) for values of sparsity ranging approximately from 75% to 95%. From these results, the constant c given by Proposition 1 could be estimated, for this experimental setup, to be around 2.5 to 4.1 for the instances with $n = 2500$, 3.2 to 5.1 for the instances with $n = 7500$, 3.7 to 5.8 for the instances with $n = 15000$, and 4.4 to 6.1 for the instances with $n = 25000$. The value of this constant may also vary for different classes of instances.

In the next subsection, we analyze the impact of our data structures on using a well-known metaheuristic that employs TS (Wang et al. 2012; Glover 2014; Samorani et al. 2019).

6.5 Measurements of solution quality using a metaheuristic

In this subsection, we seek to show that, as a consequence of speedup, implementations of metaheuristics that employ TS as a subroutine can become more competitive by using implementations of TS-CSR (Algorithm 8) or TS-CSR-PQ (Algorithm 11) instead of TS-BASIC (Algorithm 1), when solving QUBO problem instances with relatively sparse matrices. By competitive, we mean in the sense that implementations of these methods can explore solution spaces faster, and thus can reach the same high-quality solutions as when using TS-BASIC but in less time.

We conducted computational experiments using a Path Relinking (PR) metaheuristic. Specifically, we implemented the algorithm named PR2, which was originally proposed by Wang et al. (2012). We chose this algorithm because it has been often considered by other researchers in the literature for computational comparisons (Wang et al. 2012; Glover 2014; Samorani et al. 2019), and has shown better performance in comparison to several other metaheuristics. In these references, the authors often attribute the success of PR to its use of solution populations for diversification, and recombination operators and TS for intensification. In this work, although we chose

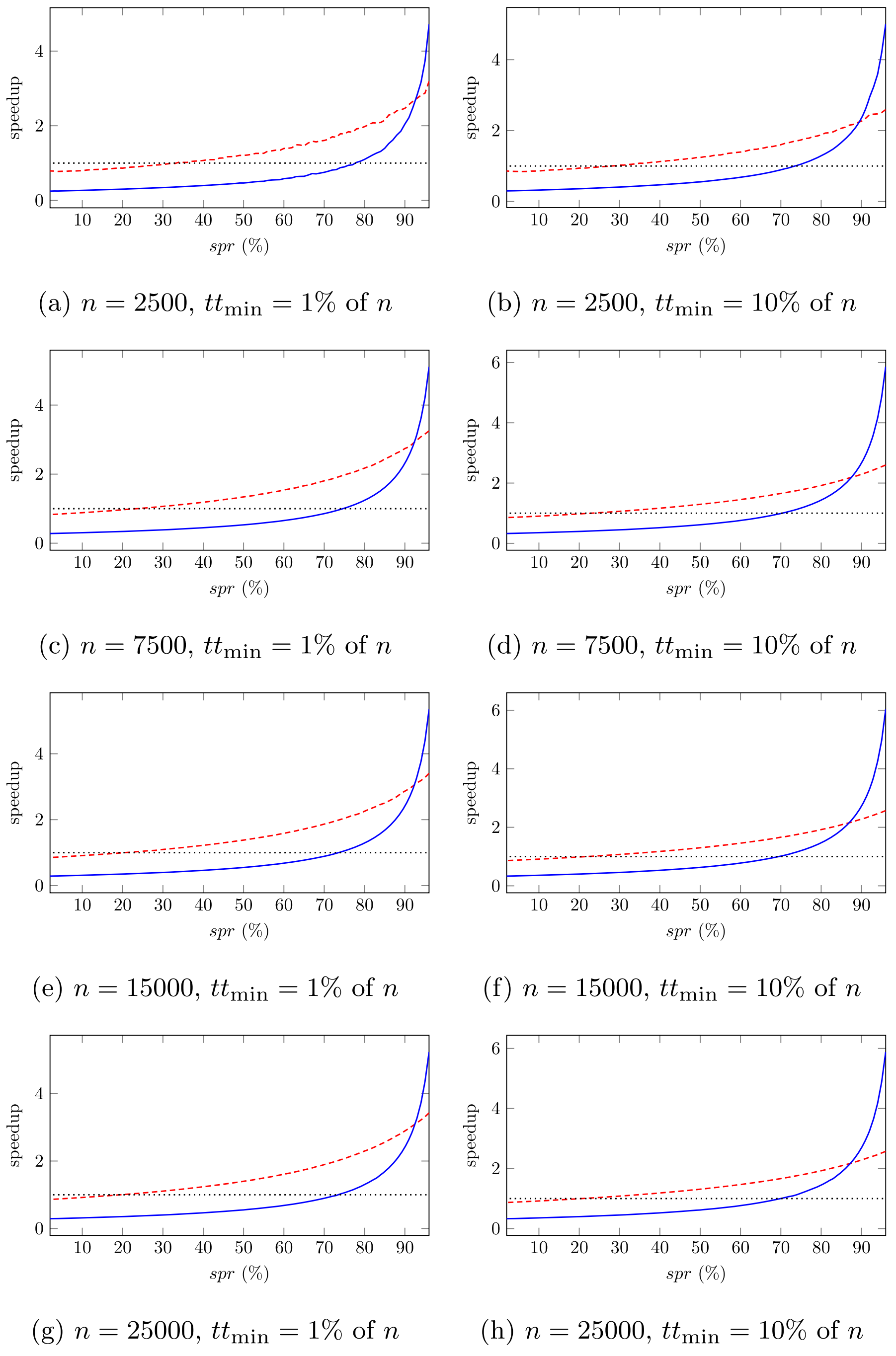
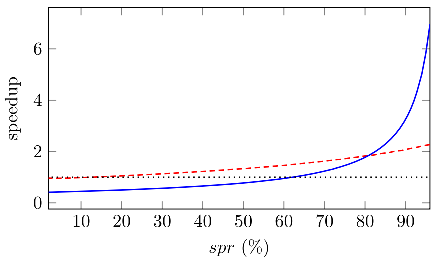
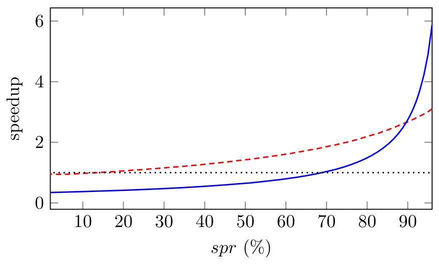


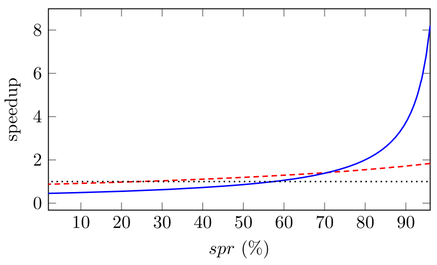
Fig. 4 Speedups of implementations of TS-CSR (Algorithm 8, red dashed curve) and TS-CSR-PQ (Algorithm 11, blue continuous curve) with respect to TS-BASIC (Algorithm 1) (Color figure online)



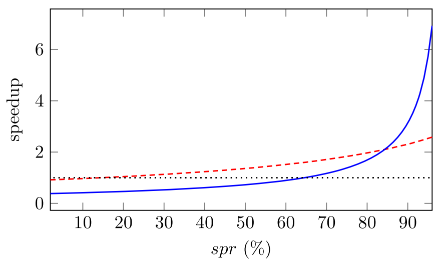
(a) $n = 2500$, $tt_{\min} = 50\%$ of n



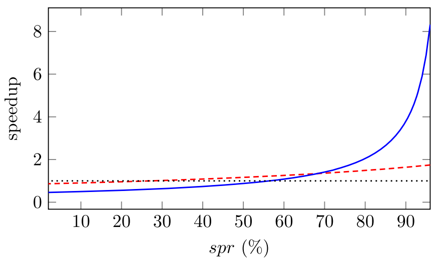
(b) $n = 2500$, $tt_{\min} = 75\%$ of n



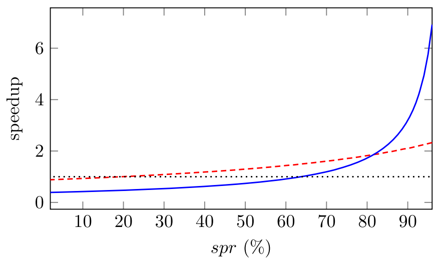
(c) $n = 7500$, $tt_{\min} = 50\%$ of n



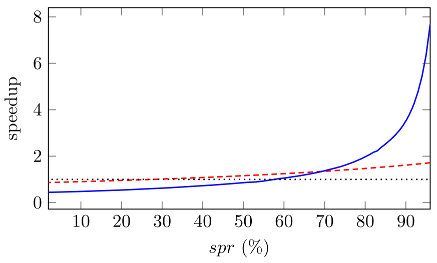
(d) $n = 7500$, $tt_{\min} = 75\%$ of n



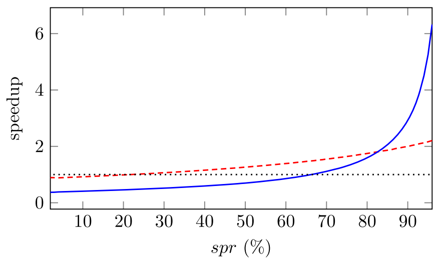
(e) $n = 15000$, $tt_{\min} = 50\%$ of n



(f) $n = 15000$, $tt_{\min} = 75\%$ of n



(g) $n = 25000$, $tt_{\min} = 50\%$ of n



(h) $n = 25000$, $tt_{\min} = 75\%$ of n

Fig. 5 Speedups of implementations of TS-CSR (Algorithm 8, red dashed curve) and TS-CSR-PQ (Algorithm 11, blue continuous curve) with respect to TS-BASIC (Algorithm 1) (Color figure online)

this specific algorithm for comparisons, we remark that any other algorithm that uses TS could have been chosen.

We compare the improvements achieved by using the different TS implementations using a measure of gap reduction, which we define as follows. Let I be an instance of an optimization problem, f_{prev} be the best known objective function value of a feasible solution to I, and A and B be two programs used to solve I, f_a be the best objective function value known to A after executing it for a specific amount of time, and f_b be the best objective function value known to B after the same amount of time. If $f_{\text{prev}} \neq f_a$, then we say that the gap reduction value of B with respect to A at that time is $(100\frac{f_b - f_a}{|f_{\text{prev}} - f_a|})\%$ for maximization problems and $(100\frac{f_a - f_b}{|f_{\text{prev}} - f_a|})\%$ for minimization problems. If $f_{\text{prev}} = f_a$, there is no gap to be reduced, and thus a gap reduction value cannot be computed. In Example 2 we illustrate this measure.

Example 2 Let I be an instance to a maximization problem, f_{prev} be the best known solution value of I, A and B be two programs used to solve it, f_a and f_b be the best objective function values found by A and B, $gap_a = f_{\text{prev}} - f_a$, and $gap_b = f_{\text{prev}} - f_b$. Then,

1. If $gap_a = 600$ and the gap reduction value of B with respect to A is 80%, then we have that $gap_b = 120$, since $80 = 100\frac{f_b - f_a}{|f_{prev} - f_a|} = 100\frac{f_b - f_{prev} - f_a + f_{prev}}{|f_{prev} - f_a|} = 100\frac{-(f_{prev} - f_b) + (f_{prev} - f_a)}{|f_{prev} - f_a|} = 100\frac{-gap_b + gap_a}{|gap_a|}$, hence $gap_b = -\frac{80}{100}|gap_a| + gap_a = 120$,
2. If $gap_a = 600$ and the gap reduction value of B with respect to A is -25%, then $gap_b = 750$, and
3. If $gap_a = 600$ and the gap reduction value of B with respect to A is 100%, then $gap_b = 0$.

We conducted the experiments using the benchmark instances with matrix sparsity greater than 90%. We used the same experimental settings as those used by Wang et al. (2012), which are as follows. The stop condition of TS is that no improved solution is found for $5n$ consecutive iterations. The values of tt_{min} were $\frac{n}{100}$ for the ORLib instances, and $\frac{n}{10}$ otherwise. The value of tt_{max} was $tt_{\text{min}} + 10$. We executed our implementations 20 times for each instance, and report on measurements of solution values as averages.

Results for all tested instances are shown in Tables 5, 6 and 7, where we measured solution values after 10 s and 100 s of execution. We also present measurements for more elapsed times in Figs. 6 and 7. In our tables and figures, the best-known solution values were obtained from (Wang et al. 2012; Alidaee et al. 2017; Ma and Hao 2016). When the gap value obtained by the implementation of `TS-BASIC` (Algorithm 1) is zero, we show a "-" symbol instead of gap reduction values for the implementations of `TS-CSR` and `TS-CSR-PQ` (Algorithms 8 and 11). We remark that, in all such cases in our experimental results, the implementations of `TS-CSR` and `TS-CSR-PQ` always reached the same values as those of `TS-BASIC`.

We observe that using `TS-CSR` and `TS-CSR-PQ` (Algorithms 8 and 11) resulted in gap reductions equal to or greater than zero in Tables 5, 6 and 7. This is expected, since they also provided speedups greater than one in Sect. 6.3. For some instances such

Table 5 Gap and gap reduction values obtained by implementations of Path Relinking using `TS-BASIC` (Algorithm 1), `TS-CSR` (Algorithm 8) and `TS-CSR-PQ` (Algorithm 11)

Max-cut instance	Best known solution value	10 s elapsed			100 s elapsed		
		`TS-BASIC`	`TS-CSR`	`TS-CSR-PQ`	`TS-BASIC`	`TS-CSR`	`TS-CSR-PQ`
		Gap	Gap reduction (%)		Gap	Gap reduction (%)	
G43.1000	6660	2.90	65.52	72.41	0.65	76.92	84.62
G44.1000	6650	4.30	67.44	80.23	0.75	73.33	86.67
G45.1000	6654	3.80	57.89	67.11	0.70	64.29	92.86
G46.1000	6649	5.30	54.72	71.70	1.05	71.43	71.43
G47.1000	6665	13.10	22.14	25.95	9.20	3.26	3.80
G51.1000	3848	9.70	36.08	42.78	5.55	9.91	47.75
G52.1000	3851	10.70	24.77	39.25	6.50	28.46	49.23
G53.1000	3850	10.20	37.25	51.96	5.05	31.68	35.64
G54.1000	3852	14.85	25.59	43.43	8.40	35.12	47.02
G22.2000	13,359	27.45	49.18	77.41	7.45	42.95	55.03
G23.2000	13,344	27.90	22.58	46.77	15.60	25.32	48.40
G24.2000	13,337	32.55	20.12	36.41	21.55	22.27	35.50
G25.2000	13,340	29.45	26.99	37.69	19.20	32.55	41.15
G26.2000	13,328	27.75	14.95	27.03	21.05	23.99	49.41
G27.2000	3341	31.70	22.24	44.16	18.70	18.72	24.06
G28.2000	3298	23.20	25.00	42.24	14.15	28.62	35.69
G29.2000	3405	37.65	23.24	49.80	20.60	38.83	66.99
G30.2000	3413	29.75	32.27	49.75	17.10	46.78	59.36
G31.2000	3310	26.75	21.87	43.93	16.60	34.94	47.89
G32.2000	1410	49.90	39.48	75.75	20.50	56.10	76.59
G33.2000	1382	59.40	38.22	70.88	27.20	46.69	60.66

Table 5 continued

Max-cut instance	Best known solution value	10 s elapsed			100 s elapsed		
		TS-BASIC	TS-CSR	TS-CSR-PQ	TS-BASIC	TS-CSR	TS-CSR-PQ
		Gap	Gap reduction (%)		Gap	Gap reduction (%)	
G34.2000	1384	40.20	49.00	76.62	13.90	41.73	67.63
G35.2000	7687	44.15	17.21	33.18	33.25	21.05	29.17
G36.2000	7680	45.55	17.56	32.27	33.50	16.87	32.24
G37.2000	7691	45.20	18.81	29.31	33.90	15.34	28.47
G38.2000	7688	45.00	19.33	31.56	32.35	21.64	39.26
G39.2000	2408	50.25	26.37	41.00	32.65	24.96	38.44
G40.2000	2400	45.90	19.39	35.51	32.45	21.73	40.37
G41.2000	2405	50.65	20.93	42.55	33.65	23.92	38.93
G42.2000	2481	48.20	18.98	43.57	32.50	29.23	46.15
G48.3000	6000	0	–	–	0	–	–
G49.3000	6000	0	–	–	0	–	–
G50.3000	5880	0	–	–	0	–	–
G55.5000	10,299	254.50	29.53	48.72	171.55	18.45	43.43
G56.5000	4017	239.60	24.12	46.91	171.55	21.68	41.33
G57.5000	3494	307.60	21.13	62.84	197.90	31.53	72.41
G58.5000	19,293	172.10	9.91	29.40	134.75	9.50	28.68
G59.5000	6087	232.65	19.06	43.97	162.25	17.47	35.50
G60.7000	14,190	363.70	19.99	43.02	254.25	10.74	42.87
G61.7000	5798	361.05	15.41	38.44	268.10	8.65	40.40
G62.7000	4870	423.20	0.07	57.30	319.80	28.42	76.33
G63.7000	27,045	247.25	0	27.99	215.40	11.88	31.57

Table 5 continued

Max-cut instance	Best known solution value	10 s elapsed			100 s elapsed		
		TS-BASIC	TS-CSR	TS-CSR-PQ	TS-BASIC	TS-CSR	TS-CSR-PQ
		Gap	Gap reduction (%)		Gap	Gap reduction (%)	
G64.7000	8751	346.20	1.75	33.64	274.80	11.06	34.19
G65.8000	5562	524.00	0	52.31	444.70	33.03	74.52
G66.9000	6364	653.30	0	48.48	589.40	31.98	72.04
G67.10000	6950	655.90	0	45.68	637.20	31.54	76.43
G70.10000	9591	625.55	0.38	39.41	533.50	8.40	60.71
G72.10000	7006	674.80	0	47.70	655.30	32.60	75.90
G77.14000	9938	981.80	0	43.36	949.00	17.83	71.15
G81.20000	14,048	1379.20	0	37.96	1347.00	0	64.77

Table 6 Gap and gap reduction values obtained by implementations of Path Relinking using `TS-BASIC` (Algorithm 1), `TS-CSR` (Algorithm 8) and `TS-CSR-PQ` (Algorithm 11)

Max-cut instance	Best known solution value	10 s elapsed			100 s elapsed		
		`TS-BASIC`	`TS-CSR`	`TS-CSR-PQ`	`TS-BASIC`	`TS-CSR`	`TS-CSR-PQ`
		Gap	Gap reduction (%)		Gap	Gap reduction (%)	
sg3dl101000.1000	896	3.30	24.24	51.52	1.60	43.75	56.25
sg3dl102000.1000	900	1.60	100	100	0	–	–
sg3dl103000.1000	892	3.40	35.29	88.24	0.80	100	100
sg3dl104000.1000	898	1.60	68.75	87.50	0.20	100	100
sg3dl105000.1000	886	2.90	41.38	48.28	1.60	37.50	68.75
sg3dl106000.1000	888	2.40	66.67	100	0.20	100	100
sg3dl107000.1000	900	3.10	25.81	45.16	1.70	5.88	11.76
sg3dl108000.1000	882	2.60	19.23	42.31	1.50	26.67	66.67
sg3dl109000.1000	902	4.30	53.49	74.42	1.20	41.67	58.33
sg3dl1010000.1000	894	1.90	47.37	68.42	0.70	57.14	85.71
sg3dl141000.2744	2446	33.20	33.73	48.49	19.60	20.92	51.02
sg3dl142000.2744	2458	38.10	35.17	63.78	20.80	40.38	61.54
sg3dl143000.2744	2444	33.90	33.63	52.80	20.70	28.02	48.79
sg3dl144000.2744	2450	32.40	28.09	53.09	18.70	20.32	51.34
sg3dl145000.2744	2446	37.50	31.47	55.47	20.40	27.45	48.53
sg3dl146000.2744	2452	36.00	35.83	59.17	19.20	28.65	50.00
sg3dl147000.2744	2444	33.50	24.48	50.75	21.30	27.70	52.58
sg3dl148000.2744	2448	34.60	30.06	59.83	19.30	30.57	43.52
sg3dl149000.2744	2428	36.40	38.74	60.71	18.50	23.24	42.16
sg3dl1410000.2744	2458	37.60	31.38	54.52	21.40	27.57	43.46
torusg3-15.3375	285,149,199	11,600,922.50	0.91	26.52	9,543,996.50	3.80	49.18
toruspm3-15-50.3375	3014	41.00	17.07	63.41	29.00	3.45	68.97

Table 7 Gap and gap reduction values obtained by implementations of Path Relinking using `TS-BASIC` (Algorithm 1), `TS-CSR` (Algorithm 8) and `TS-CSR-PQ` (Algorithm 11)

`QUBO` instance	Best known solution value	10 s elapsed			100 s elapsed		
		`TS-BASIC`	`TS-CSR`	`TS-CSR-PQ`	`TS-BASIC`	`TS-CSR`	`TS-CSR-PQ`
		Gap	Gap reduction (%)		Gap	Gap reduction (%)	
bqp1.1000	371,438	23.25	100	100	0	–	–
bqp2.1000	354,932	14.95	100	100	0	–	–
bqp3.1000	371,236	0	–	–	0	–	–
bqp4.1000	370,675	0	–	–	0	–	–
bqp5.1000	352,760	4.50	100	100	0	–	–
bqp6.1000	359,629	0	–	–	0	–	–
bqp7.1000	371,193	0	–	–	0	–	–
bqp8.1000	351,994	0	–	–	0	–	–
bqp9.1000	349,337	29.05	85.71	85.71	0	–	–
bqp10.1000	351,415	62.70	100	100	0	–	–
bqp1.2500	1,515,944	0	–	–	0	–	–
bqp2.2500	1,471,392	185.55	23.77	27.03	10.25	100	100
bqp3.2500	1,414,192	0	–	–	0	–	–
bqp4.2500	1,507,701	0	–	–	0	–	–
bqp5.2500	1,491,816	0	–	–	0	–	–
bqp6.2500	1,469,162	17.00	100	100	0	–	–
bqp7.2500	1,479,040	0	–	–	0	–	–
bqp8.2500	1,484,199	0	–	–	0	–	–
bqp9.2500	1,482,413	7.00	14.29	100	0	–	–
bqp10.2500	1,483,355	28.50	18.95	97.19	0	–	–

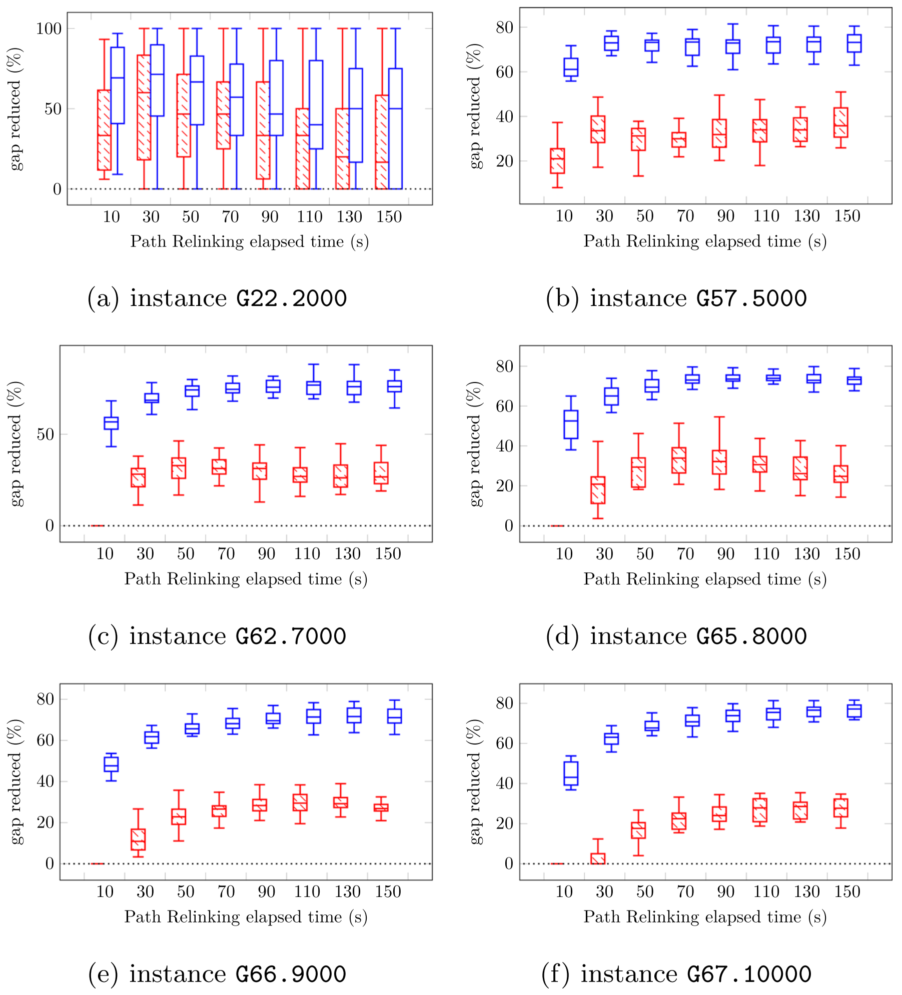
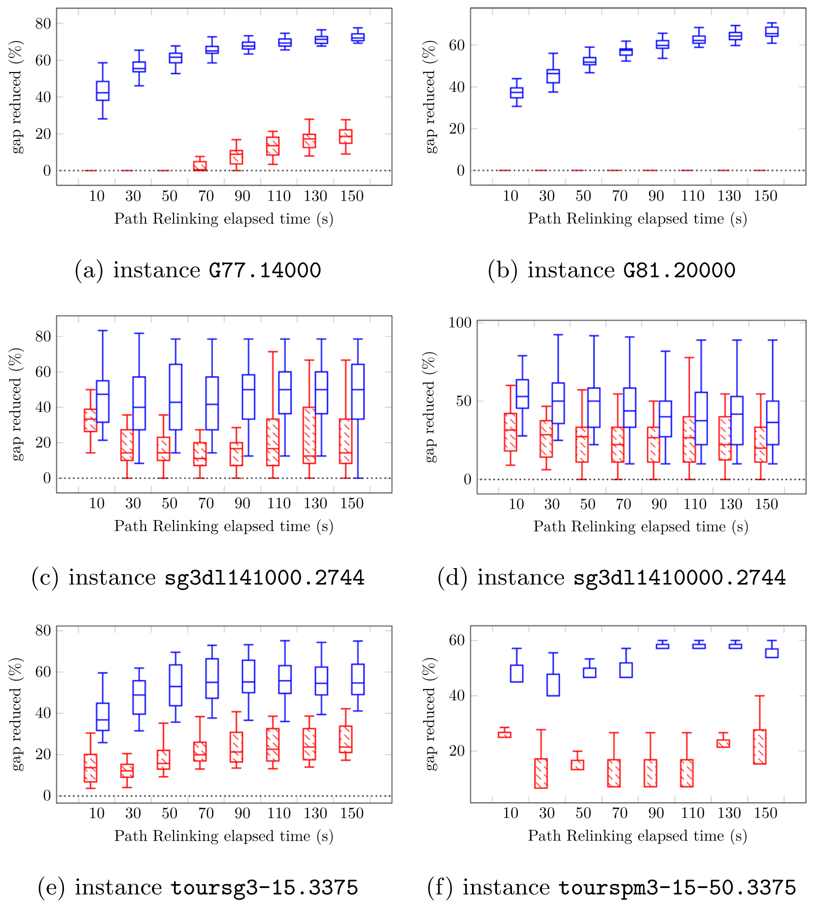


Fig. 6 Gap and gap reduction values of implementations of Path Relinking procedure when using TS-CSR (Algorithm 8, red filled box-plots) and TS-CSR-PQ (Algorithm 11, blue box-plots) with respect to when using TS-BASIC (Algorithm 1) (Color figure online)

as G48, G49, G50, and bqp1.2500, since the implementation using TS-BASIC (Algorithm 1) already reached the best known solution value before 10 s, the gap reduction obtained by using TS-CSR and TS-CSR-PQ was zero. Based on these results, we find that employing our data structures can benefit methods that make use of TS for solving QUBO problem instances with matrices of relatively high sparsity, since the speedups allowed exploring more solutions in less time.

Additionally, we present in Table 8 measurements of solution values using our implementation of PR2 with TS-CSR-PQ (Algorithm 11), for the larger benchmark instances and given longer time limits. We show the best obtained gaps from the best known solution values.



(a) instance `G77.14000` (b) instance `G81.20000`

(c) instance `sg3dl141000.2744` (d) instance `sg3dl1410000.2744`

(e) instance `toursg3-15.3375` (f) instance `torspm3-15-50.3375`

Fig. 7 Gap and gap reduction values of implementations of Path Relinking procedure when using `TS-CSR` (Algorithm 8, red filled box-plots) and `TS-CSR-PQ` (Algorithm 11, blue box-plots) with respect to when using `TS-BASIC` (Algorithm 1) (Color figure online)

For the instance `toruspm3-15-50`, our implementation was able to obtain a solution value better than the best known solution value after the two hour mark. The -4 gap value indicates that the best found solution at that time had an objective function value of 4 above the best known value in the literature. Considering that, from our results shown in Table 2, `TS-CSR-PQ` was several times faster than `TS-BASIC` and `TS-CSR` for the instance `toruspm3-15-50`, then implementations of `PR` using `TS-BASIC` or `TS-CSR` could also take several times longer to find that improved best solution value. This highlights how metaheuristic methods can benefit from our results.

We present our final conclusions in the next section.

Table 8 Gap values obtained by the implementation of Path Relinking using `TS-CSR-PQ` (Algorithm 11) for longer elapsed times

Max-cut instance	Seconds elapsed				
	1575	3150	6300	12600	25200
	Gap from best known solution value				
G55.5000	96	79	68	68	68
G56.5000	76	50	37	36	36
G57.5000	34	34	32	32	32
G58.5000	87	86	86	86	86
G59.5000	89	87	87	87	87
G60.7000	149	146	135	119	115
G61.7000	80	76	75	74	74
G62.7000	44	24	20	20	20
G63.7000	116	114	112	109	106
G64.7000	156	152	150	150	149
G65.8000	72	70	70	62	58
G66.9000	84	66	56	50	50
G67.10000	74	68	64	56	54
G70.10000	176	162	155	146	135
G72.10000	88	72	60	54	48
G77.14000	136	116	102	80	68
G81.20000	178	164	156	152	142
torusg3-15.3375	2,685,956	2,583,075	2,388,016	1,945,312	1,507,850
toruspm3-15-50.3375	10	4	0	− 4	− 4

7 Final comments and future directions

In this work, we proposed data structures to speed up implementations of a Tabu Search (TS) heuristic for solving instances of the `QUBO` problem with relatively sparse matrices. Specifically, we proposed two techniques. The first consists in using a compressed sparse row representation of the instance matrix, and the second, in addition to that, also makes use of priority queues during the search.

Each iteration of the TS procedure consists in selecting the best possible 1-flip move, updating the tabu list, and updating the reevaluation vector of the solution. Our literature review indicates that each of these steps currently take at least $O(n(1+m))$ time, where n is the number of variables and m is the number of local search iterations performed in that TS iteration. In contrast, we found that when using just the CSR representation of the instance matrix, the asymptotic time complexity of updating the reevaluation vector changed to $O(D)$, where D is an integer between 1 and n that is related to the instance matrix. However, the the asymptotic time complexity of an entire TS iteration remained $O(n(1+m))$. Then, when we use a CSR representation of the instance matrix and also priority queues, selecting a 1-flip move became $O(1)$, updating the tabu list became $O(\log n)$, updating the reevaluation vector became

$O(D \log n)$, and each iteration of a local search also became $O(D \log n)$. Thus, the asymptotic time complexity of an entire iteration of the search procedure also became $O(D \log n(1+m))$. For the classes of instances where D is lower than $\frac{n}{\log n}$, that time complexity becomes $o(n(1+m))$, thus an improvement over $O(n(1+m))$.

Our computational experiments show that speedups of up to 5 were obtained when using the implementation that employs just the CSR representation of the instance matrix. Moreover, the speedups were greater than one for values of matrix sparsity above around 10%, which means that the CSR representation is worth using for a relatively wide range of matrix sparsity values. As for the implementations also using priority queues, relatively larger speedups were obtained for the instances with matrices with sparsity values above around 90%. For example, for instances with values of matrix sparsity 98%, 99%, 99.75%, 99.9%, and 99.97%, the speedups were around 5, 11, 50, 65, and 200, respectively.

We also conducted experiments using a Path Relinking (PR) metaheuristic. To the best of our knowledge, current PR procedures described in the literature for solving QUBO problem instances employ TS as a subroutine. We found that, when the TS embedded in a PR was implemented using our proposed data structures, that PR implementation obtained high-quality solutions to instances with relatively sparse matrices in less time than when implemented without them. These results lead us to believe that several QUBO methods that employ TS can become more competitive by using our techniques.

We believe that there are opportunities for future research in this direction. We are currently investigating the possibility of using our proposed data structures for larger neighborhoods, that is, those involving multiple simultaneous flips. It is also possible to investigate using our proposed priority queue structures in heuristics other than TS for solving QUBO problem instances, and even heuristics for solving instances of other optimization problems.

Acknowledgements This work was supported by São Paulo Research Foundation (FAPESP) (Grant 2018/03819–4), CNPq (Grant 312206/2015-1) and the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) Finance Code 001 (Grant 88887.497750/2020-00).

Author Contributions RNL: Conceptualization, Methodology, Formal analysis, Writing—original draft, Writing—review & editing, Software, Validation, Data curation, Investigation, Resources, Project administration, Visualization. EAJA: Conceptualization, Methodology, Formal analysis, Data curation, Writing—review & editing, Validation, Investigation, Project administration, Supervision. CNM: Methodology, Formal analysis, Writing—review & editing, Validation, Investigation, Project administration, Supervision, Resources, Funding acquisition.

Availability of data and materials All data that support the findings of this study are available from the corresponding author upon request.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

Code availability All source codes used in this study are available from the corresponding author upon request.

Ethical approval Not applicable.

Consent to participate Not applicable.

Consent for publication Not applicable.

Appendix A Constructing a CSR matrix from a matrix in full form

In Algorithm 12 we present one algorithm that can build a CSR representation of a Q matrix. It has asymptotic time complexity $O(n^2)$. In Lines 1 to 13, we compute the values of $\deg i$ for each $i \in \{1, 2, \ldots, n\}$. Based on that, we fill the R, C and V vectors in the remaining lines of the algorithm. We use this algorithm in our computational experiments.

Algorithm 12 BUILD_CSR_MATRIX

```
Input: $n$, $Q$
Output: $V$, $C$, $R$
1:  for $i \leftarrow 1$ to $n$ do
2:      $d_i \leftarrow 0$
3:      for $j \leftarrow 1$ to $i$ do
4:          if $q_{ij} \neq 0$ then
5:              $d_i \leftarrow d_i + 1$
6:          end if
7:      end for
8:      for $j \leftarrow i + 1$ to $n$ do
9:          if $q_{ji} \neq 0$ then
10:             $d_i \leftarrow d_i + 1$
11:         end if
12:     end for
13: end for
14: $r_1 \leftarrow 1$
15: for $i \leftarrow 1$ to $n$ do
16:     $r_{i+1} \leftarrow r_i + d_i$
17:     $k \leftarrow 0$
18:     for $j \leftarrow 1$ to $i$ do
19:         if $q_{ij} \neq 0$ then
20:             $c_{r_i+k} \leftarrow j$
21:             $v_{r_i+k} \leftarrow q_{ij}$
22:             $k \leftarrow k + 1$
23:         end if
24:     end for
25:     for $j \leftarrow i + 1$ to $n$ do
26:         if $q_{ij} \neq 0$ then
27:             $c_{r_i+k} \leftarrow j$
28:             $v_{r_i+k} \leftarrow q_{ji}$
29:             $k \leftarrow k + 1$
30:         end if
31:     end for
32: end for
```

Appendix B Algorithms for manipulating indexed priority queues

Here we describe in a detailed manner the operations which we used to manipulate indexed priority queues, as proposed in Definition 6. These algorithms are relatively simple modifications of well-known procedures for maintaining minimum heap data structures (Cormen et al. 2001).

Throughout our algorithms, every indexed priority queue is initialized using Algorithm 13. This algorithm has asymptotic time complexity $O(n)$.

Algorithm 13 IPQ_INITIALIZE

```
Input: $n$
Output: $p = (n, m, h, z)$
1: $m \leftarrow 0$
2: for $i \leftarrow 1$ to $n$ do
3:     $h_i \leftarrow i$
4:     $z_i \leftarrow i$
5: end for
```

Algorithm 14 obtains the current number of elements in the queue. This algorithm and has asymptotic time complexity $O(1)$.

Algorithm 14 IPQ_SIZE

```
Input: $p = (n, m, h, z)$
Output: $m$
```

Algorithm 15 checks whether a given element is in the queue. This algorithm has asymptotic time complexity $O(1)$.

Algorithm 15 IPQ_CONTAINS

```
Input: $p = (n, m, h, z), i$
Output: contains
1: contains $\leftarrow (z_i > 0$ and $z_i \leq m)$
```

Algorithm 16 obtains the element in the queue with the minimum priority value. This algorithm has asymptotic time complexity $O(1)$.

Algorithm 16 IPQ_TOP

```
Input: $p = (n, m, h, z)$
Output: top
1: top $\leftarrow h_1$
```

Algorithm 17 is used as a subroutine of Algorithms 18 to 21, for the purpose of placing an element i at a specific position k in the queue. The element already at the position k is placed at the previous position of i. This algorithm has asymptotic time complexity $O(1)$.

Algorithm 17 IPQ_SWAP_POSITION

```
Input: $p = (n, m, h, z), i, k$
Output: $p$
1: $z_{h_k} \leftarrow z_i$
2: $h_{z_i} \leftarrow h_i$
3: $z_i \leftarrow k$
4: $h_k \leftarrow i$
```

Algorithm 18 is used for correcting the minimum-heap property, starting from a given position k up to the last position of the queue. It is based on the well-known Heapify procedure (Cormen et al. 2001). This algorithm has asymptotic time complexity $O(\log m)$.

Algorithm 18 IPQ_SIFT_UP

```
Input: $p = (n, m, h, z), v, k$
Output: $p$
1: while true do
2:   smallest $\leftarrow k$
3:   left $\leftarrow 2k + 1$
4:   right $\leftarrow 2k + 2$
5:   if left $\leq m$ and $v_{h_{\texttt{left}}} < v_{h_{\texttt{smallest}}}$ then
6:     smallest $\leftarrow$ left
7:   end if
8:   if right $\leq m$ and $v_{h_{\texttt{left}}} < v_{h_{\texttt{smallest}}}$ then
9:     smallest $\leftarrow$ right
10:  end if
11:  if smallest $= k$ then
12:    break
13:  end if
14:  IPQ_SWAP_POSITION($p, h_k$, smallest)
15:  $k \leftarrow$ smallest
16: end while
```

Algorithm 19, like Algorithm 18, is also used for correcting the minimum-heap property, except it starts from a given position k down to the first position of the queue. This algorithm has asymptotic time complexity $O(\log m)$.

Algorithm 20 is used to insert an element into the queue. This algorithm has asymptotic time complexity $O(\log m)$.

Algorithm 21 is used to remove an element from the queue. This algorithm has asymptotic time complexity $O(\log m)$.

Algorithm 22 is used to change the priority value of an element in the queue. This algorithm has asymptotic time complexity $O(\log m)$.

Algorithm 19 IPQ_SIFT_DOWN

```
Input: $p = (n, m, h, z), v, k$
Output: $p$
1: parent ← $\lfloor \frac{k-1}{2} \rfloor$
2: while $k > 1$ and $v_{h_k} < v_{h_{\texttt{parent}}}$ do
3:     IPQ_SWAP_POSITION($p, h_k$, parent)
4:     $k$ ← parent
5:     parent ← $\lfloor \frac{k-1}{2} \rfloor$
6: end while
```

Algorithm 20 IPQ_INSERT

```
Input: $p = (n, m, h, z), i, v$
Output: $p$
1: $m \leftarrow m + 1$
2: IPQ_SWAP_POSITION($p, i, m$)
3: IPQ_SIFT_DOWN($p, m, v$)
```

Algorithm 21 IPQ_REMOVE

```
Input: $p = (n, m, h, z), v, i$
Output: $p$
1: $k \leftarrow z_i$
2: IPQ_SWAP_POSITION($p, i, m$)
3: $m \leftarrow m - 1$
4: IPQ_SIFT_UP($p, k, v$)
```

Algorithm 22 IPQ_CHANGE_PRIORITY

```
Input: $p = (n, m, h, z), i, v$, new_value
Output: $p, v$
1: old_value ← $v_i$
2: $v_i$ ← new_value
3: if new_value < old_value then
4:     IPQ_SIFT_DOWN($p, z_i, v$)
5: else
6:     IPQ_SIFT_UP($p, z_i, v$)
7: end if
```

References

Alidaee, B., Sloan, H., Wang, H.: Simple and fast novel diversification approach for the UBQP based on sequential improvement local search. Comput. Ind. Eng. **111**, 164–175 (2017). https://doi.org/10.1016/j.cie.2017.07.012

Anacleto, E.A., Meneses, C.N., Ravelo, S.V.: Closed-form formulas for evaluating r-flip moves to the unconstrained binary quadratic programming problem. Comput. Oper. Res. **113**(104), 774 (2020). https://doi.org/10.1016/j.cor.2019.104774

Aramon, M., Rosenberg, G., Valiante, E., et al.: Physics-inspired optimization for quadratic unconstrained problems using a digital annealer. Nat. Phys. **7**, 48 (2019). https://doi.org/10.3389/fphy.2019.00048

Bian, Z., Chudak, F., Israel, R., et al.: Discrete optimization using quantum annealing on sparse ising models. Front. Phys. **2**, 56 (2014). https://doi.org/10.3389/fphy.2014.00056

Boettcher, S.: Analysis of the relation between quadratic unconstrained binary optimization and the spin-glass ground-state problem. Phys. Rev. Res. **1**(3), 033,142 (2019). https://doi.org/10.1103/physrevresearch.1.033142

Boixo, S., Rønnow, T.F., Isakov, S.V., et al.: Evidence for quantum annealing with more than one hundred qubits. Nat. Phys. **10**, 218–224 (2014). https://doi.org/10.1038/nphys2900

Boros, E., Hammer, P.L., Tavares, G.: Local search heuristics for quadratic unconstrained binary optimization (QUBO). J. Heuristics **13**(2), 99–132 (2007). https://doi.org/10.1007/s10732-007-9009-3

Branda, M., Novotný, J., Olstad, A.: Fixed interval scheduling under uncertainty—a Tabu Search algorithm for an extended robust coloring formulation. Comput. Ind. Eng. **93**, 45–54 (2016). https://doi.org/10.1016/j.cie.2015.12.021

Buluç, A., Fineman, J.T., Frigo, M., et al.: Parallel sparse matrix–vector and matrix-transpose–vector multiplication using compressed sparse blocks. In: Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures—SPAA '09. ACM Press, pp. 233–244 (2009). https://doi.org/10.1145/1583991.1584053

Chapuis, G., Djidjev, H., Hahn, G., et al.: Finding maximum cliques on the d-wave quantum annealer. J. Signal Process. Syst. **91**(3–4), 363–377 (2018). https://doi.org/10.1007/s11265-018-1357-8

Cormen, T.H., Leiserson, C.E., Rivest, R.L., et al.: Introduction to Algorithms, 2nd edn. The MIT Press, Cambridge, MA (2001)

Cruz-Santos, W., Venegas-Andraca, S., Lanzagorta, M.: A QUBO formulation of the stereo matching problem for D-Wave quantum annealers. Entropy **20**(10), 786 (2018). https://doi.org/10.3390/e20100786

Duff, I.S.: A survey of sparse matrix research. Proc. IEEE **65**(4), 500–535 (1977). https://doi.org/10.1109/PROC.1977.10514

Dunning, I., Gupta, S., Silberholz, J.: What works best when? A systematic evaluation of heuristics for max-cut and QUBO. INFORMS J. Comput. **30**(3), 608–624 (2018). https://doi.org/10.1287/ijoc.2017.0798

Eisenstat, S.C., Gursky, M.C., Schultz, M.H., et al.: Yale sparse matrix package I: the symmetric codes. Int. J. Numer. Methods Eng. **18**(8), 1145–1151 (1982). https://doi.org/10.1002/nme.1620180804

Glover, F.: Exterior path relinking for zero-one optimization. Int. J. Appl. Metaheuristic Comput. **5**(3), 1–8 (2014). https://doi.org/10.4018/ijamc.2014070101

Glover, F., Hao, J.K.: Efficient evaluations for solving large 0–1 unconstrained quadratic optimisation problems. Int. J. Metaheuristics **1**(1), 3 (2010). https://doi.org/10.1504/ijmheur.2010.033120

Glover, F., Laguna, M.: Tabu Search. Springer, Boston, MA (1997). https://doi.org/10.1007/978-1-4615-6089-0

Glover, F., Kochenberger, G.A., Alidaee, B.: Adaptive memory Tabu Search for binary quadratic programs. Manage. Sci. **44**(3), 336–345 (1998). https://doi.org/10.1287/mnsc.44.3.336

Glover, F., Alidaee, B., Rego, C., et al.: One-pass heuristics for large-scale unconstrained binary quadratic problems. Eur. J. Oper. Res. **137**(2), 272–287 (2002). https://doi.org/10.1016/s0377-2217(01)00209-0

Glover, F., Lü, Z., Hao, J.K.: Diversification-driven Tabu Search for unconstrained binary quadratic problems. 4OR **8**(3), 239–253 (2010). https://doi.org/10.1007/s10288-009-0115-y

Glover, F., Laguna, M., Martí, R.: Principles and strategies of Tabu Search. In: Gonzalez, T.F. (ed.) Handbook of Approximation Algorithms and Metaheuristics, 2nd edn., pp. 361–377. Chapman and Hall/CRC, New York (2018). https://doi.org/10.1201/9781351236423-21

Hua, R., Dinneen, M.J.: Improved QUBO formulation of the graph isomorphism problem. SN Comput. Sci. **1**(1), 1–18 (2019). https://doi.org/10.1007/s42979-019-0020-1

Kochenberger, G.A., Hao, J.K., Lü, Z., et al.: Solving large scale max cut problems via Tabu Search. J. Heuristics **19**(4), 565–571 (2011). https://doi.org/10.1007/s10732-011-9189-8

Kochenberger, G.A., Hao, J.K., Glover, F., et al.: The unconstrained binary quadratic programming problem: a survey. J. Comb. Optim. **28**(1), 58–81 (2014). https://doi.org/10.1007/s10878-014-9734-0

Lewis, M., Metcalfe, J., Kochenberger, G.A.: Robust optimisation of unconstrained binary quadratic problems. Int. J. Oper. Res. **36**(4), 441 (2019). https://doi.org/10.1504/ijor.2019.104050

Lü, Z., Glover, F., Hao, J.K.: A hybrid metaheuristic approach to solving the UBQP problem. Eur. J. Oper. Res. **207**(3), 1254–1262 (2010). https://doi.org/10.1016/j.ejor.2010.06.039

Lucas, A.: Ising formulations of many NP problems. Front. Phys. **2**, 5 (2014). https://doi.org/10.3389/fphy.2014.00005

Ma, F., Hao, J.K.: A multiple search operator heuristic for the max-k-cut problem. Ann. Oper. Res. **248**(1–2), 365–403 (2016). https://doi.org/10.1007/s10479-016-2234-0

Manber, U.: Introduction to Algorithms: A Creative Approach. Addison-Wesley, Reading, MA (1989)

Matsumoto, M., Nishimura, T.: Mersenne twister. ACM Trans. Model. Comput. Simul. **8**(1), 3–30 (1998). https://doi.org/10.1145/272991.272995

Merz, P., Katayama, K.: Memetic algorithms for the unconstrained binary quadratic programming problem. Biosystems **78**(1–3), 99–118 (2004). https://doi.org/10.1016/j.biosystems.2004.08.002

Milne, A., Rounds, M., Goddard, P.: Optimal feature selection using a quantum annealer. In: Dempster, M.A.H., Kanniainen, J., Keane, J., Vynckier, E. (eds.) High-Performance Computing in Finance, pp. 561–588. Chapman and Hall/CRC, New York p (2018). https://doi.org/10.1201/9781315372006-19

Oliveira, N.M.D., Silva, R.M.A., Oliveira, W.R.D.: QUBO formulation for the contact map overlap problem. Int. J. Quantum Inf. **16**(08), 1840007 (2018). https://doi.org/10.1142/s0219749918400075

Palubeckis, G.: Multistart Tabu Search strategies for the unconstrained binary quadratic optimization problem. Ann. Oper. Res. **131**(1–4), 259–282 (2004). https://doi.org/10.1023/b:anor.0000039522.58036.68

Palubeckis, G.: Iterated Tabu Search for the unconstrained binary quadratic optimization problem. Informatica **17**(2), 279–296 (2006). https://doi.org/10.15388/informatica.2006.138

Papp, D.: On the complexity of local search in unconstrained quadratic binary optimization. SIAM J. Optim. **26**(2), 1257–1261 (2016). https://doi.org/10.1137/15m1047775

Pardalos, P.M., Rodgers, G.P.: Computational aspects of a branch and bound algorithm for quadratic zero-one programming. Computing **45**(2), 131–144 (1990). https://doi.org/10.1007/bf02247879

Pastorello, D., Blanzieri, E.: Quantum annealing learning search for solving QUBO problems. Quantum Inf. Process. **18**(10), 1–17 (2019). https://doi.org/10.1007/s11128-019-2418-z

Samorani, M., Wang, Y., Wang, Y., et al.: Clustering-driven evolutionary algorithms: an application of path relinking to the quadratic unconstrained binary optimization problem. J. Heuristics **25**(4–5), 629–642 (2019). https://doi.org/10.1007/s10732-018-9403-z

Wang, Y., Lü, Z., Glover, F., et al.: Backbone guided Tabu Search for solving the UBQP problem. J. Heuristics **19**(4), 679–695 (2011). https://doi.org/10.1007/s10732-011-9164-4

Wang, Y., Lü, Z., Glover, F., et al.: Path relinking for unconstrained binary quadratic programming. Eur. J. Oper. Res. **223**(3), 595–604 (2012). https://doi.org/10.1016/j.ejor.2012.07.012

Wang, Y., Lü, Z., Glover, F., et al.: Probabilistic GRASP-Tabu Search algorithms for the UBQP problem. Comput. Oper. Res. **40**(12), 3100–3107 (2013). https://doi.org/10.1016/j.cor.2011.12.006