A Large Population Island Framework for the Unconstrained Binary Quadratic Problem

Olivier Goudet, Adrien Goëffon, Jin-Kao Hao *

LERIA, University of Angers, 2 Boulevard Lavoisier, 49045 Angers, France

Computers & Operations Research 2024
https://doi.org/10.1016/j.cor.2024.106684

Abstract

The unconstrained binary quadratic problem is an NP-hard problem and has applications in many fields. Recently, the problem has attracted much interest in the field of quantum optimization, as it is directly related to the Ising problem in physics and the development of quantum computers. However, effectively solving large instances of this problem remains a major challenge for existing solution methods. To advance the state of the art in solving the problem on a large scale, we propose an evolutionary algorithm with a very large population organized in different islands and integrating a new pairing and recombination method to produce promising offspring in each generation. Numerous experiments are conducted to evaluate the effects of different pairing strategies, crossovers, and migration topologies. This research has led to the discovery of new bounds for difficult instances of the maximum cut problem, which has been transformed using the binary quadratic formulation.

Keywords: Combinatorial optimization, evolutionary search; Island model; Parallel search; Heuristics; Unconstrained binary quadratic problem; Quadratic unconstrained binary optimization

---

1 Introduction

The Unconstrained Binary Quadratic Problem (UBQP) or Quadratic Unconstrained Binary Optimization (QUBO) is to find a vector $x = [x(1), \ldots, x(n)]$ of size n maximizing the function $f : \{0, 1\}^n \to \mathbb{R}$ given by:

* Corresponding author.
Email addresses: olivier.goudet@univ-angers.fr (Olivier Goudet), adrien.goeffon@univ-angers.fr (Adrien Goëffon), jin-kao.hao@univ-angers.fr (Jin-Kao Hao).

$$f(x) = x^t Q x, \tag{1}$$

where Q is real symmetric matrix of size $n \times n$ and x^t is the transposed vector of x.

Many problems that arise in real applications can be formulated with this UBQP model such as job scheduling problems on parallel computing environments [1], clustering of micro-array data in biology [19] or design of manufacturing systems in industry [44]. We refer the reader to [9,20] for a comprehensive overview of the various applications of the UBQP tool. The recent book [34], which deals specifically with the UBQP, is further evidence of the interest in this problem.

Moreover, the UBQP is very general, since various NP-Hard and NP-complete combinatorial optimization problems can be conveniently mapped to UBQP [10,22]. Examples of popular NP-hard problems that can be addressed with the UBQP include the graph coloring problem [18], the maximum clique problem [32], and the maximum cut problem [5]. The latter is further discussed in this article.

The UBQP has also recently attracted much interest in the field of quantum optimization [29,36], due to its direct connection with the Ising spin glass problem [3] in physics and the development of new quantum computers. Indeed, the formulation with binary variables $x(i) \in \{0, 1\}$ of the Ising problem makes it simple to model it with qubits that can be in a superposition of the `0` state and the `1` state at the same time. The problem becomes then to find the minimum global energy state of the associated Hamiltonian.

Several exact approaches have been proposed in the literature to tackle the UBQP with branch and bound algorithms [16,21,31] or semi-definite programming (SDP) approaches [15,35]. According to [34], parallel versions of the best current exact algorithms can solve UBQP instances up to $n = 300$. For larger instances, various heuristics have been proposed in the literature. They partially explore the search space to find a vector x with a *good* score in a limited amount of time. However, this partial search does not guarantee that there is no better solution in the search space. A heuristic only finds a lower bound of the optimal UBQP value.

Given the NP-hard nature of the UBQP, effectively solving large instances of this problem remains a very challenging task for existing solution methods. To advance the state of the art in solving the problem on a large scale, we propose to study an evolutionary algorithm with a very large number of individuals that are placed on different islands (each individual is a solution $x \in \{0, 1\}^n$ of the UBQP). In the context of such a large and diverse population, our goal is also to study the impact of different migration topologies between these islands [37] as well as the importance of the matching strategy used to create new good quality offspring in each generation.

To achieve these goals, we develop a Large Population Island (LPI) framework, based on parallel optimization using Graphics Processing Units (GPU). The implemented LPI algorithm is characterized by its generality and simplicity. We show its effectiveness in solving the UBQP.

This paper is organized as follows. Section 2 presents related studies on the UBQP in the literature and focuses on the most relevant algorithms that contributed to the design of the proposed LPI algorithm. Section 3 describes the LPI algorithm. Section 4 outlines the experimental setting used to empirically validate LPI, then reports the results obtained and compared them to the state of the art. Section 5 discusses the contribution and presents perspectives for future work.

2 Related work

In a very recent and comprehensive survey on heuristics and metaheuristics applied to the UBQP [45], three types of methods are identified: (i) greedy constructive methods [27], (ii) local search based methods with simulated annealing [2,17] or tabu methods [12,49], and (iii) population based algorithms, with hybrid genetic algorithms [26], path-relinking algorithms [23,47] or hyperheuristics [6]. This section revisits the key mechanisms of both tabu search and path-relinking procedures, since our algorithm depends on them.

2.1 Tabu search

The most popular local search used in a significant number of heuristics of the literature for the UBQP [12,38,39,42,47] is the *one-flip* Tabu Search (TS_1). To improve a solution x, TS_1 iteratively makes transitions from x to a neighboring solution x', by *flipping* the value of one of the binary variables $x(i)$ of the vector x to its complementary value $1 - x(i)$. Thus, x and x' differ only by the value of one of their variables. The size of the neighborhood explored by TS_1 is equal to n. At each iteration, TS_1 selects among the eligible neighboring solutions the best neighbor x' according to the evaluation function f (cf. Equation 1), and replaces x by x'. A neighboring solution is eligible if the flip is not forbidden by the tabu list or if it is better than the best recorded solution found so far during the local search. When the flip of a variable $x(i)$ is performed, it is recorded in a tabu list, indicating that this variable cannot be flipped for the next T iterations. In many existing tabu search algorithms on the UBQP [38,39,47], T increases linearly with the size n of the neighborhood and is equal to $\alpha \cdot n + R$, where R is a random integer in $\{1, \ldots, 10\}$ and α is a hyperparameter of the algorithm.

Other neighborhoods with various operators for the UBQP have been studied in the literature (see for example [24]). In particular, a straightforward extension of the *one-flip* operator is the *two-flip* operator consisting in *flipping* at the same time a pair of variables $\{x(i), x(j)\}$ to their respective complementary values $1 - x(i)$ and $1 - x(j)$. Computing the difference of scores of the neighboring solutions when using a *two-flip* operator can be done efficiently from the Q matrix using the formula provided by [8]. When considering all the pairs of variables, the *two-flip* neighborhood associated with this operator is of size $\binom{n}{2}$. In practice, this neighborhood becomes huge for large problem instances.

In general, such a basic tabu search algorithm can be very efficient at reaching the optimal result for small instances. However, when the size of the instance increases, the search can get stuck in a local optimum. To overcome this difficulty, iterated tabu search algorithms such as the D2TS algorithm [12] allow to escape local traps by coupling the tabu search with perturbation and restart procedures. This D2TS algorithm has recently been improved using a parallel environment [39], where different tabu searches start from different initial starting points and produce different search trajectories. During the search, these tabu searches can communicate with each other in order to exchange useful information and get out of local traps. Another stream of work using tabu search concerns hybrid population-based algorithms, which alternate between such a local search and a combination operator to produce new offspring solutions. This framework has proven highly successful in solving the UBQP in several algorithms introduced since 2010 [23,38,40,47]. The combination operator used in these algorithms is the path-relinking procedure detailed in the next subsection.

2.2 Path-relinking

The path-relinking (PR) method [11] provides an interesting means to generate new solutions (offspring) by exploring a trajectory (or path) connecting two solutions of the population (parents). The first parent x^k, which is the starting solution on the path, is called the *initiating solution*, and the second parent x^l, which corresponds to the last solution on the path, is called the *guiding solution*. From a general point of view, A PR procedure aims to achieve three goals simultaneously: (i) generate new offspring that share some similarity with both parents in order to pass on useful information to the next generations; (ii) build offspring that are sufficiently different from both parents in order to explore new areas of the search space; (iii) generate offspring solutions of good quality that are further improved by local search.

Different versions of PR have been proposed for the UBQP in the literature. The most popular PR proposed in [38,47] uses a greedy strategy. If NC denotes the set of variable indices for which x^k and x^l have different values, the greedy PR procedure starts from the initiating solution x^k and performs greedy *one-flip* moves on the set of variables indexed by NC to find a good quality trajectory connecting x^k and x^l. Once the full trajectory is computed, the best solution on the path solutions with a

Hamming distance of at least $\gamma \cdot |NC|$ from both the initiating and guiding solutions is kept for further improvement with the tabu search in the next generation. γ is a hyperparameter of the algorithm, typically set to be the value of 0.3.

In order to perform an efficient PR, the choice of the parent solutions used as initiating solution and guiding solution is crucial. First, the offspring has a better chance of being interesting if the two parents are of good quality, but these two parents must also be sufficiently distanced in the search space, in order to build a new offspring solution allowing to explore a new area in the search space. However, if the parents are too distanced in the search space, even if they are of good quality, no common good backbone of solution can be transmitted to the next generations, resulting in offspring of poor quality.

To investigate the question of which matching strategy to adopt for this problem, the recent hybrid algorithm proposed in [38] for the UBQP studied the importance of carefully choosing parents for the PR procedure. It uses a population of up to 50 individuals that are separated into at most five different clusters using the K-means clustering algorithm based on the Hamming distance calculated between each pair of individuals. Then, two strategies for combining the parents with the PR are proposed. The first one, called "external linkage strategy", consists in considering only the paths connecting the two most distant solutions belonging to two different clusters, while the second one, called "internal linkage strategy" consists in considering only the paths connecting the two highest quality solutions belonging to the same cluster. The external linking strategy increases diversification, while the internal linking strategy increases search intensification, as it combines high quality solutions that are close to each other.

Although this K-means clustering algorithm is interesting for separating individuals into different groups that explore different areas of the search space, it can be time consuming when the population size is very large. Our approach, based on an evolutionary algorithm with a very large population, is to use instead the island model [50], where the different groups are decided at the outset. The recombination procedure should naturally reduce the distances within each island. [1]

The contributions of our paper are the following:

- We introduce a large population island framework for the UBQP with different matching strategies, migration topologies, and a new recombination procedure.
- We implement, based on this framework, a large population algorithm with GPU-based parallel computing.
- We report new upper bounds for difficult maximum cut instances.

[1] This hypothesis is confirmed by an experiment reported in Appendix D, where we show that the average distance between individuals on each island measured over generations decreases faster than the average distance between individuals in the general population (not necessarily belonging to the same islands).

3 Large Population Island Framework

3.1 Main scheme

The proposed LPI framework is a general population-based evolutionary approach that alternates between a tabu search to find high-quality local optima and a recombination procedure (inspired by path-relinking) to generate new offspring solutions. Existing evolutionary algorithms for the UBQP typically have a population between 10 [47] and 50 individuals [38] to avoid high computation time, as they do not use parallel computing. Our LPI algorithm takes advantage of massive parallel computing with GPUs and uses a very large population, whose size is given by

$$|P| = \max\left(2000, \min\left(64000, \left\lfloor \frac{320000}{n} \right\rfloor \times 1000\right)\right). \quad (2)$$

As an example, $|P| = 64000$ for $n \leq 5000$ and $|P| = 32000$ for $n = 10000$. This number of individuals is a multiple of 1000 and decreases when the size n of the instance is greater than 5000, in order to limit the memory required for large instances.

Such a large population has three benefits: (i) it allows to take advantage of massive parallel computation on GPU hardware [13,14]; (ii) it allows to ensure a large diversity of solutions in the population to avoid premature convergence of the algorithm; (iii) it increases the chance for each individual to find a good match in the population for the combination procedure (see Section 3.4).

To ensure its search and computational efficiency, LPI takes full advantage of GPU-based parallel processing. At each generation, $|P|$ different local searches are performed in parallel on the GPU, starting from different starting points and producing different search trajectories in the search space. Then, $|P|$ combination procedures are carried out in parallel to produce $|P|$ new starting points for the next generation.

Managing diversity in such a large population requires that billions of pairwise distance estimates to be made in each generation. In order to limit the number of distance evaluations, and to add parallelism to the population update procedure that merges the current population and the offspring population to create the next population, we place the different individuals on I separate islands. That is, we split the whole population P into I sub-populations P_i, $P = P_1 \cup P_2 \cup \cdots \cup P_I$, such that each sub-population has the same size for easier management with the GPU hardware. Therefore, all sub-populations are of size p. A sensitivity analysis of this parameter p will be conducted in Section 4.4.1.

In addition, migrations take place between islands in order to propagate copies of the best individuals from each island to other islands (see Section 3.6). This gives them a better chance of finding a good neighborhood solution for the combination procedure on another island.

The algorithm takes a matrix Q as input and tries to find a binary vector x such that $f(x)$ given by equation (1) is maximum. The pseudo-code of the proposed Large Population Island framework is shown in Algorithm 1.

Algorithm 1 Large Population Island (LPI) algorithm for the unconstrained quadratic binary optimization problem

---

Input: Symmetric matrix Q of size $n \times n$ describing UBQP function f, number I of islands, size p of the sub-population on each island, number k of nearest neighbors and number m of migrants.
Output: The best vector x^* found so far.
for $i = 1, \ldots, I$ **in parallel do**
    $P^i = \{x_1^i, \ldots, x_p^i\} \leftarrow$ random_subpopulation_initialization()
    for $j = 1, \ldots, p$ **in parallel do**
        $x_j^i \leftarrow$ local_search(x_j^i) // Section 3.2
    end for
end for
$x^* = \text{argmax}_{x_j^i,\ i \in \{1,\ldots,I\},\ j \in \{1,\ldots,p\}}\ f(x_j^i)$
repeat
    for $i = 1, \ldots, I$ **in parallel do**
        for $j = 1, \ldots, p$ **in parallel do**
            $\bar{x}_j^i \leftarrow$ nearest_neighbor_choice(x_j^i, P^i, k) // Section 3.3
            $o_j^i \leftarrow$ combination_procedure($x_j^i, \bar{x}_j^i$) // Section 3.4
            $o_j^i \leftarrow$ local_search(o_j^i) // Section 3.2
        end for
    end for
    $o^* = \text{argmax}_{o_j^i,\ i \in \{1,\ldots,I\},\ j \in \{1,\ldots,p\}}\ f(o_j^i)$
    if $f(o^*) > f(x^*)$ **then**
        $x^* \leftarrow o^*$
    end if
    for $i = 1, \ldots, I$ **in parallel do**
        $\{x_1^i, \ldots, x_p^i\} \leftarrow$ population_update($x_1^i, \ldots, x_p^i, o_1^i, \ldots, o_p^i$) // Section 3.5
    end for
    for $i = 1, \ldots, I$, **in parallel do**
        $\{\dot{x}_1^i, \ldots, \dot{x}_m^i\} \leftarrow$ migration($x_1^i, \ldots, x_p^i$) // Section 3.6
        $\{x_1^{(i\%I)+1}, \ldots, x_p^{(i\%I)+1}\} \leftarrow$ population_update($\dot{x}_1^i, \ldots, \dot{x}_m^i, x_1^{(i\%I)+1}, \ldots, x_p^{(i\%I)+1}$)
    end for
until stopping condition met
return x^*

At the beginning, all the individuals of the population (binary vectors of size n) are initialized at random in parallel and are simultaneously improved by running in parallel a *one-or-two-flip* tabu search (Section 3.2) to maximize the fitness function f.

The algorithm then repeats a loop (generation) until a cutoff time limit is reached. Each generation t involves the execution of three components:

(1) Each individual is randomly matched with one of its nearest neighbors within its island (Section 3.3) and $|P|$ combination procedures are performed in parallel to generate $|P|$ offspring solutions (Section 3.4), which are then improved by the local search (Section 3.2).
(2) For every island, the local search algorithm computes in parallel the distances between all the pairs of existing and new individuals (cf. Section 3.5). Afterward, the population updating procedure (also described in Section 3.5) merges the $2 \cdot |P|$ existing and new individuals in each island, while considering the fitness of each individual and the distances between them, to ensure that diversity is maintained within each sub-population.
(3) Migrations occur between islands. Copies of the m best individuals of island i are sent to island $i+1$ (or island 1 for copies from island I) according to a ring topology (cf. Section 3.6). These copies are retained in island i when sent to island $i+1$ (and not removed from island i).

When the allocated time is consumed, the algorithm returns the best solution x^* found so far and stops. The score $f(x^*)$ is a lower bound of the given UBQP instance.

3.2 Local search

LPI employs a sparse *one-or-two-flip* tabu search ($TS^*_{1|2}$) to simultaneously improve in parallel the individuals of the whole population P. Given a vector solution $x = [x(1), x(2), \ldots, x(n)]$, $TS^*_{1|2}$ uses two move operators:

- the *one-flip* operator flips a bit $x(i)$. It changes its value from $x(i)$ to $1 - x(i)$, leading to a neighboring solution denoted as $x \oplus \langle \text{flip } x(i) \rangle$. When using this operator, the one-move neighborhood $N_1(x, Q)$ of x is given by:

$$N_1(x, Q) = \{x \oplus \langle \text{flip } x(i) \rangle : 1 \leq i \leq n\}; \tag{3}$$

- the *two-flip* operator flips a pair of bits $\{x(i), x(j)\}$ such as $Q(i, j) \neq 0$ changing simultaneously the values of the bits $x(i)$ and $x(j)$ to their complementary values $1 - x(i)$ and $1 - x(j)$, leading to a neighboring solution denoted as $x \oplus \langle \text{flip } x(i), \text{flip } x(j) \rangle$. Note that contrary to [24], our *two-flip* operator does not consider all the $\binom{n}{2}$ possible two-flip moves, but only the pairs $\{x(i), x(j)\}$ such that $Q(i, j) \neq 0$. Indeed, since considering every move is too time-consuming, it is more interesting to restrict the search of the best $\{x(i), x(j)\}$ among those characterized by a non-zero interaction coefficient $Q(i, j)$ between the variables $x(i)$ and $x(j)$. Thus, the two-move neighborhood $N^*_2(x, Q)$ of x using this operator is given by:

$$N_2^*(x,Q) = \{x \;\oplus \langle \text{flip } x(i), \text{flip } x(j)\rangle : 1 \leq i < j \leq n, Q(i,j) \neq 0\}. \tag{4}$$

To summarize, $TS^*_{1|2}$ explores the following union neighborhood of the *one-flip* and *two-flip* neighborhoods:

$$N^*(x,Q) = N_1(x,Q) \cup N_2^*(x,Q). \tag{5}$$

Using the *two-flip* neighborhood in addition to the *one-flip* neighborhood greatly increases the efficiency of the local search for some difficult instances with a relatively sparse Q matrix (as it is shown in Section 4.5). In particular, it can *trigger* in one iteration the combined effect related to an off-diagonal coefficient $Q(i,j) \neq 0$ $(i \neq j)$, which requires that both variables $x(i)$ and $x(j)$ be equal to 1 at the same time.

$TS^*_{1|2}$ makes transitions between different n-vectors with the neighborhood $N^*(x,Q)$ and the fitness function f. It iteratively replaces the current solution x by a neighboring solution x' taken from $N^*(x,Q)$ during a total number of N_L iterations. At each iteration, a best admissible neighboring solution x' is selected to replace x.

After each iteration, the corresponding move (*one-flip* or *two-flip*) is recorded in a tabu list. The tabu tenure classically depends on the size of the neighborhood and is set to the value of $\alpha \cdot \Delta + R$, where R is a random integer from $[0;9]$, α is a hyperparameter of the algorithm, and Δ is the cardinality of $N^*(x,Q)$. A neighboring solution x' is considered to be admissible if it is not forbidden by the tabu list, unless it is better (according to the fitness evaluation function f) than the best solution found so far (aspiration criterion). The *one-flip* and *two-flip* neighborhood evaluations are both performed incrementally using the streamline techniques detailed in [7,8]. It exploits the interaction graph of the variables to efficiently compute the score variation due to each move (delta) with respect to the current solution.

Note that even when considering non-zero inputs of the Q matrix, the size of the neighborhood, Δ, can still become very large if it is dense. If the ratio Δ/n is higher than a density threshold ρ, we replace $TS^*_{1|2}$ with the standard *one-flip* tabu search TS_1 as presented in subsection 2.1. It corresponds to replacing the neighborhood $N^*(x,Q)$ by $N_1(x) = \{x \;\oplus \langle \text{flip } x(i)\rangle : 1 \leq i \leq n\}$. Otherwise, the rest of the algorithm remains unchanged.

At each generation of the algorithm, the $|P| = I \cdot p$ tabu search procedures are launched in parallel on the GPU to raise the quality of the offspring population. The time complexity of the TS_1 and $TS^*_{1|2}$ procedures are respectively in $O(n \cdot N_L \cdot I \cdot p)$ and $O(\Delta \cdot N_L \cdot I \cdot p)$. Their space complexity is respectively in $O(n \cdot I \cdot p)$ and $O(\Delta \cdot I \cdot p)$.

3.3 Matching strategy

At each generation, the LPI algorithm runs $|P| = I \cdot p$ combination procedures in parallel to generate $|P|$ new offspring solutions. To do this, LPI uses each existing solution in the current population as the starting solution and combines it with another solution of the population. This approach ensures that each individual in the population contributes genetic information to the next generation while promoting the creation of diverse offspring that will be improved by the local search procedure.

Due to the use of a large population, the individuals can be highly dissimilar and share minimal information despite being on the same island. Consequently, combining parents that are too dissimilar can lead to the production of offspring solutions that are of poor quality. Therefore, we propose to use a k-nearest neighbor algorithm for this matching strategy, which allows to find pairs of parents sharing some similarity.

For each island i and each individual x_j^i, where $j \in \{1, \ldots, p\}$, another individual, $\bar{x}_j^i$, is chosen randomly from the set of the k-nearest neighbors of x_j^i in island i in terms of the Hamming distance, [2] that have never been combined with x_j^i in the previous generations. [3]

A sensitivity analysis of the hyperparameter k ranging from 1 (closest deterministic match) to the maximum value $p - 1$ (totally random match) is shown in Section 4.4.3.

Finding the set of k-nearest neighbors for each of the $|P| = I \cdot p$ individuals is done in parallel on the GPU. The time and space complexity of this matching algorithm is $O(I \cdot p^2)$.

3.4 Combination mechanism

The LPI algorithm uses a new combination mechanism, called Restricted Tabu Search Combination (RTSC). RTSC is inspired by path-relinking (PR) algorithms [11] and variable fixing strategies [46,48]. It searches for an offspring solution located between two different high quality solutions of the population, by applying a local

[2] Given two solution n-vectors x^k and x^l, the Hamming distance $HD(x^k, x^l)$ measures the dissimilarity between x^k and x^l, which corresponds to the number of binary variables with different values in x^k and x^l.
[3] Note that the selection of the couple of individuals (x_i, x_j) in a given island does not prevent the algorithm from later selecting the couple (x_j, x_i), since the combination procedure we use in LPI is asymmetric. Indeed, this procedure gives different solutions when it starts from individual x_i instead of individual x_j (cf. Section 3.4).

search algorithm on the restricted subset of variables that are different in the two parents.

Given a solution x_j^i and its neighbor $\bar{x}_j^i$, RTSC starts from x_j^i and performs N_C iterations [4] of the same *one-or-two-flip* tabu search presented in subsection 3.2, but restricted to the variables that take different values between x_j^i and $\bar{x}_j^i$. Moreover, instead of maximizing f directly, RTSC maximizes an extended evaluation function $F : \{0,1\}^n \to \mathbb{R}$ given by:

$$F(x) = f(x) + \kappa \cdot \min(d(x, x_j^i), d(x, \bar{x}_j^i)), \tag{6}$$

where κ is a hyperparameter that adjusts the trade-off between the score and the distance from both parents. Maximizing this function F encourages finding an offspring that is of good quality, but also sufficiently distant from both parents.

All these $|P| = I \cdot p$ combination procedures, which solves $|P|$ UBQP sub-problems, are performed in parallel on the GPU grid.

3.5 Distances computation and population update

The $|P|$ new offsprings generated with the procedure described in the last subsection are improved by the local search presented in Section 3.2. These new solutions are then used to update the population.

3.5.1 Distance computation

LPI maintains I matrices of size p^2 on each island to store the distances between any two solutions in each sub-population. These symmetric matrices are initialized with the $\binom{p}{2}$ pairwise distances computed for each pair of individuals in the initial sub-population. They are then updated each time a new individual is added to each sub-population. To merge the p existing solutions and the p new solutions on each island i, LPI needs to compute (i) the p^2 distances between each individual in the sub-population $P^i = \{x_1^i, \dots, x_p^i\}$ and each improved offspring individual in $O^i = \{o_1^i, \dots, o_p^i\}$, and (ii) the $\binom{p}{2}$ distances between all pairs of offspring individuals in O^i. When operations are performed sequentially, the algorithmic complexity of distance calculations for the whole population is $O(n \cdot I \cdot p^2)$. However, as all these distance computations for the whole population are independent, they can be run in parallel on the GPU, with one computation per thread. With this parallel implementation, the algorithmic complexity can be reduced to $O(n)$. The auxiliary space complexity is $O(I \cdot p^2)$ (to store the distance matrix).

[4] N_C is a hyperparameter of the algorithm whose value is given in Table 1.

3.5.2 Population update procedure

The LPI population update procedure retains the best individuals while also ensuring a minimum distance between them [33]. In parallel, on each island i, the population update procedure greedily selects and adds the best individuals (maximizing the fitness function f) from $P^i_{all} := \{x^i_1, \ldots, x^i_p\} \cup \{o^i_1, \ldots, o^i_p\}$ to the sub-population of the next generation P^i_{t+1}, initialized with an empty set, until P^i_{t+1} contains p individuals, subject to the constraint that $HD(x^k, x^l) \geq \gamma$ for all $x^k, x^l \in P^i_{t+1}$, where $k \neq l$.[5] γ is the minimum spacing required for two different individuals of each sub-population. γ is a hyperparameter of the algorithm whose value is indicated in Table 1. This value is defined as a percentage of n, the size of the instance and the maximum Hamming distance between two individuals in the search space. The time complexity of the population update procedure is $O(I \cdot p^2)$. This procedure is sequential in each island, but the updates on the different islands are independent and thus done in parallel. The auxiliary space complexity of this procedure is $O(n \cdot I \cdot p)$ (to build the populations P^i_{all} and P^i_{t+1}).

3.6 Migration between islands

In the standard version of the algorithm, the different islands are organized according to a *uni-directional ring* migration topology [37], and at each generation copies of the m best individuals of each island i (that have never already been sent before) are sent to the island $i+1$ (see Figure 1). As it is a ring, island $I+1$ corresponds to island 1. A binary vector is updated throughout the execution in each island to know which individual has already been sent.

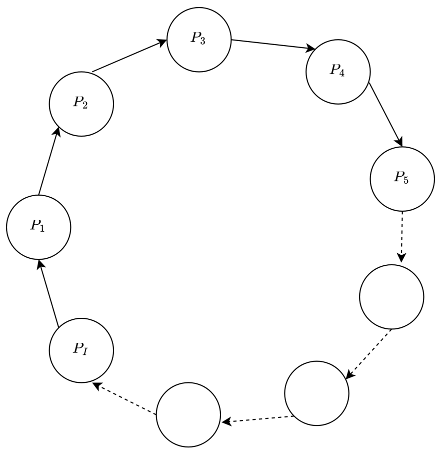

Fig. 1. Organization of the islands according to an uni-directional ring topology.

These migrations make it possible to spread the most promising individuals in

[5] This distance criterion is not met if the required size of each sub-population is not reached, which is very rare.

other islands, offering them a better opportunity to find a good mate and produce an interesting offspring in each generation (see Section 3.4). It then has a role to play in terms of intensification. However, spreading too many individuals to too many islands would quickly lead to too much similarity between the I sub-populations. The implications of other migration topologies are discussed in Section 4.4.6. The number m of individuals migrating from an island is also an important hyperparameter of the algorithm, which must be carefully chosen to ensure a good trade-off between intensification and diversification (see Section 4.4.4).

The acceptance of these migrants into each of the I sub-populations are subject to the same population update procedure as described in subsection 3.5, ensuring a minimum distance between each individual on each island. The time complexity of the distance computation and population update procedure for the migrations is $O(I \cdot m^2 \cdot n + I \cdot p \cdot m \cdot n)$ (computation of all distances between migrants, as well as between migrants and existing individuals in the population), while its auxiliary space complexity is $O(I \cdot m^2 + I \cdot m \cdot p + I \cdot (m + p) \cdot n)$ (to store the corresponding distance matrices and the sub-populations augmented with migrants).

4 Experiments

This section is dedicated to a study of important factors of the algorithms such as the organization the individuals into islands, the choice of the matching strategy and the combination procedure. A comparison with the best state-of-the-art methods is then presented on classical benchmarks from the literature.

4.1 Benchmark Instances

Three main sets of UBQP benchmark instances are considered in the literature [6].

- The first set consists of 21 well-known large instances named p3000.1, . . . , p7000.3 with sizes ranging from $n = 3000$ to 7000 and densities ranging from 0.5 to 1.0. These instances were generated using the generator proposed by [30], and are widely used in the literature [38,46,47].
- The second set contains 10 instances of size $n = 2744$ (named sg3dl141000, sg3dl142000, . . . , sg3dl1410000). The instances are generated by simulating Ising spin glasses on cubic lattices, where the weight values assigned to the spin interactions are restricted to 1, 0, or -1. Computational results on these instances have been reported in [25,41,51].

[6] These instances will be available on the github repository site of the project after publication.

- The third set contains 71 instances derived from the maximum cut problem, named G1, ..., G72, G81 with sizes ranging from $n = 800$ to $n = 20000$. A machine-independent graph generator was used to construct these instances, which consist of toroidal, planar, and randomly weighted graphs. The weight values assigned to the edges of these graphs are limited to 1, 0, or -1. These instances are widely used in the literature to validate algorithms proposed for the UBQP [6,47] as well as to evaluate the performance of specific algorithms dedicated to the maximum cut problem [4,25,40,41,51].

We specifically focus on the instances from the third set which are the most challenging. These instances are regarded as particularly difficult ones, since no algorithm in the literature, including the most recent ones [4,25,40,41,51], has been able to achieve the best-known results for all instances of the maximum cut problem.

The results of LPI for the first two sets of instances are shown in Appendices B and C. For these first two sets, LPI always obtains the best known scores for all instances and all independent runs (perfect success rate). However, the computation time required to obtain them is quite high on average for the largest instances of the first set (almost 6 hours), as can be seen in Table B.1. For the second set of cubic lattice instances (see Table C.1), the computation time required remains reasonable for all instances (less than one hour).

4.2 Implementation and parameter setting

The LPI algorithm[7] was implemented in Python using the Numba 0.53 library for CUDA kernel implementation of local searches, distance computations, and crossovers. LPI is specifically designed to run on GPUs, and in this work, we used an V100 Nvidia graphic card with 32 GB of memory. Note that most of the time used by the algorithm is spent performing local searches and crossovers, which are run in CUDA (via the Numba library) rather than sequentially in Python. The Python language simply calls the various libraries with C++ and CUDA backends.

In each generation of LPI, each of the $|P| = I \cdot p$ tabu searches is executed on a single GPU thread. To optimize memory access, each thread uses a local memory to store specific information, such as the bit vector of the current solution and the tabu list. The threads are arranged in blocks of 64 and launched on the GPU grid. Since each tabu search runs independently on each thread, no shared memory per block is required. However, a global memory is used to store general information, such as the Q matrix of the problem, to avoid duplication of information. All $|P|$ tabu searches are executed using a single CUDA kernel function written in Numba,

[7] The source code of the LPI algorithm is available at `https://github.com/GoudetOlivier/LPI_UBQP`.

and the best result of each tabu search is transmitted to the CPU memory after synchronization.

4.3 Parameter calibration method

To limit memory usage on the GPU device, two of the ten hyperparameters of LPI (see Table 1) are fixed. The population size $|P|$ is determined according to Equation 2 to limit the global memory, while the density threshold ρ is set to 8 to limit the local memory required in each thread for the local search.

The remaining eight parameters are determined using a grid search to maximize the score achieved for four challenging maximum cut instances: G58, G61, G64, and G70. These instances have sizes of n = 5000, 7000, 7000, and 10000, respectively, and are run for 20 hours on the GPU. The values tested during the grid search are summarized in Appendix A, and the best parameter settings obtained are shown in Table 1. These parameter values can be used as the default parameter setting for the subsequent experiments presented in this paper.

Parameter	**Description**	**Value**
Population		
$\lvert P\rvert$	Global population size	[2000,64000] (see (2))
p	sub-population size	1000
γ	Minimum spacing between individuals	$0.05 \cdot n$
m	Number of migrants	10
Local search		
N_L	Number of iterations	$2 \cdot n$
α	Tabu tenure parameter	0.04
ρ	Density threshold	8
RTSC Combination procedure		
k	Number of neighbors	48
N_C	Number of iterations	$0.5 \cdot n$
κ	Score/distance trade-off parameter	1

Table 1
Parameter setting in LPI

4.4 Sensitivity analysis

To gain insight into the behavior of our algorithm, we conducted sensitivity analysis on some key parameters. For these analysis, we adopted the same instances (G58, G61, G64, and G70) that were used during the calibration phase.

To perform the sensitivity analysis, we systematically varied the value of one parameter at a time, while keeping the default values of the other parameters (see Table 1). For each value of the parameter, we ran the LPI algorithm on each instance for 20 hours and recorded the best score obtained. We repeated this process for 10 independent runs and averaged the results to obtain a reliable estimate of the algorithm's performance.

4.4.1 Impact of the islands' size

In our algorithm, the parameter p plays a crucial role as it determines the number of individuals on each island and consequently, the number of islands I in the algorithm, which is given by $I = |P|/p$. A higher value of p results in fewer islands with larger population sizes, while a lower value results in more islands with smaller population sizes.

However, it is important to find a balance when choosing the value of p to ensure that the algorithm performs well. As shown in Figure 2, when p is set to a small value such as 10 (red curve), the performance of the algorithm deteriorates significantly. Indeed, the diversity of individuals within each island progressively decreases, which reduces the chances of generating promising offspring in each generation.

On the other hand, if p is set too high, e.g., 10000 (black curve), the number of generations that the algorithm can perform decreases because the time needed for distance evaluations scales in $O(I \cdot p^2)$ (as described in Section 3.5). Consequently, the final results obtained are not as good as those obtained for $p = 100$ and $p = 1000$.

Empirically, the best results are obtained for $p = 1000$ (green curve). This value strikes a good balance between the number of distance evaluations required in each generation and the population size on each island. A larger population size leads to more efficient matching procedures during the combination mechanism (as explained in Section 3.4), thus increasing the overall efficiency of the algorithm.

4.4.2 Impact of different combination procedures

To study the impact of the RTSC combination procedure described in Section 3.4, we compare it with three other mechanisms:

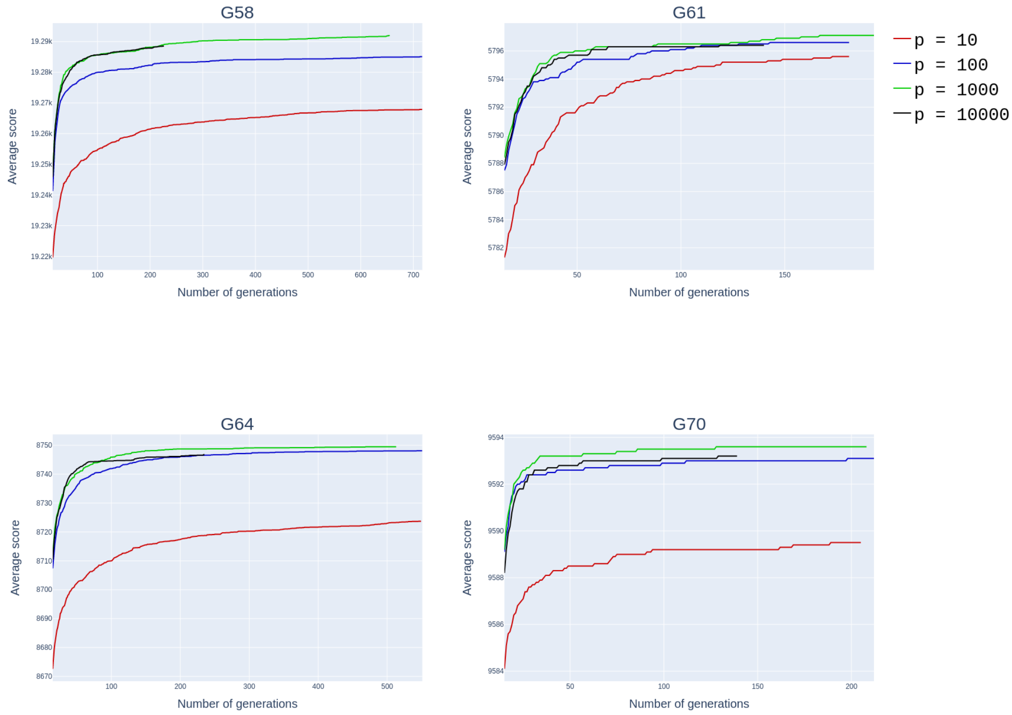

Fig. 2. Impact of the size $|p|$ of each island in LPI: 10 (red), 100 (blue), 1000 (green), 10000 (black).

- The path-relinking (PR) procedure used in [38,47] and described in Section 2.2.
- The uniform crossover (UX) used in [28]: given two parents x_i and x_j, an offspring solution x_o is built such as each value $x_o(l)$ for $l = 1, \ldots, n$ is equal to $x_i(l)$ with probability 0.5 and to $x_j(l)$ otherwise.
- A random mean crossover (MX): given two parents x_i and x_j, a random offspring solution x_o is built such that $|HD(x_i, x_o) - HD(x_j, x_o)| \leq 1$, and $\forall l = 1, \ldots, n$, $[x_i(l) = x_j(l)] \Rightarrow [x_o(l) = x_i(l)]$.
- The partition crossover (PC) [43]: the variables with the same value in both parents are transmitted to the child, then the remaining set of variables is divided into q subsets so that each subset interacts only with other variables in the same subset,[8] and finally the child is completed by inheriting the best possible assignment of variables in both parents for each subset.

As shown in Figure 3, when using RTSC (green line), fewer generations are performed within the same time budget. This is due to the additional computation time required to perform this combination procedure, which works as a local search restricted to the set of different bits between the two parents. However, the RTSC procedure allows to obtain the best results compared to the other crossovers during the search, highlighting the importance of using a combination procedure that

[8] We say that a variable x_i interacts with another variable x_j if the coefficient $Q(i, j)$ is non-zero.

diversifies the search but also favors a good intensification for this problem.

Again according to Figure 3, our RTSC procedure (green line) consistently gives much better results than the PC crossover (light blue line). We have analyzed the reason why the PC crossover does not work well for the UBQP. This is because the interaction graph of the pseudo-Boolean function variables for the UBQP cannot be easily divided into sub-graphs of homogeneous size. The PC crossover produces offspring solutions that are too close (in terms of the Hamming distance) to one of the parents. Thus, when using the PC crossover for the UBQP, the search quickly stagnates.

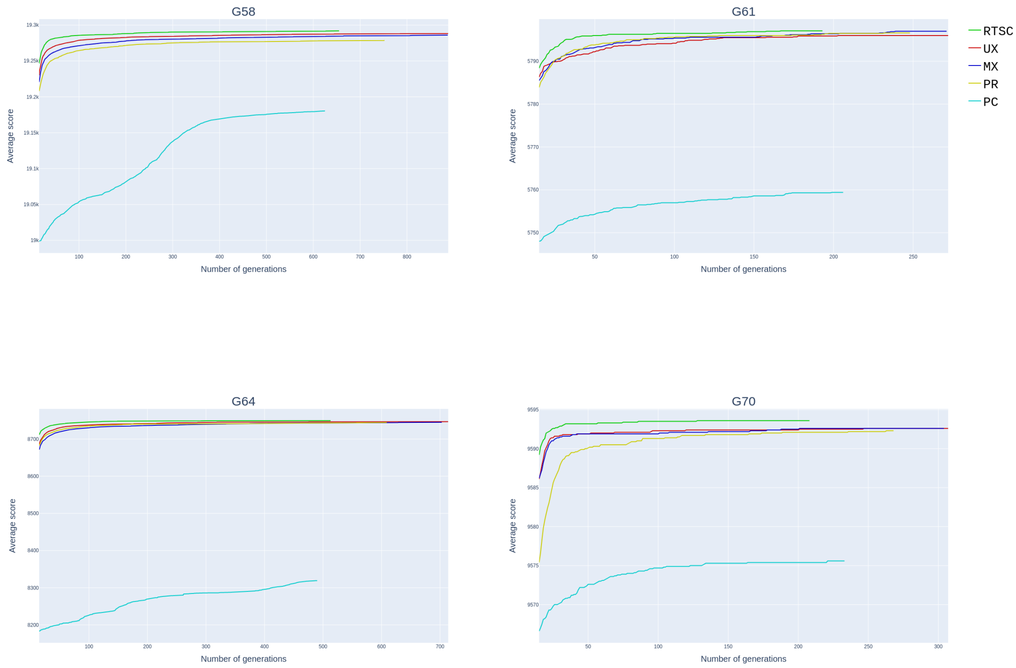

Fig. 3. Impact of different crossovers in LPI: RTSC (green), UX (red), MX (deep blue), PR (yellow) and PC (light blue).

4.4.3 Impact of the interaction neighborhood size for matching

The parameter k corresponds to the number of neighbors considered for the matching procedure described in Section 3.3.

When $k = 1$, the matching procedure is purely deterministic and consists of systematically matching each individual with its closest neighbor on its island. Note that each individual is still different from its neighbor due to the minimum distance imposed by the population update procedure (see Section 3.5). On the other hand, when $k = 1000$, the matching procedure consists of choosing another different individual completely at random on the same island.

We first observe in Figure 4 that the nearest individual matching strategy, when

$k = 1$ (red curve), obtains the worst result for these instances, which can be explained by the fact that it does not diversify the search enough. When $k = 1000$ (yellow curve), the results depend on the type of instance considered. For some instances such as G58 or G70, it is almost as good as using a smaller value of k, while for other values of k, it degrades the results. The best results appear when $k = 24$ (blue curve) or $k = 48$ (green curve), which are robust for all instances tested.

With these experiments, we emphasize that in such a large population algorithm, the number of neighbors considered for the matching strategy is a critical parameter and must be carefully chosen to avoid either a too close matching, which fails to diversify the search sufficiently, or a too far away matching, which leads to very different individuals with no shared relevant information.

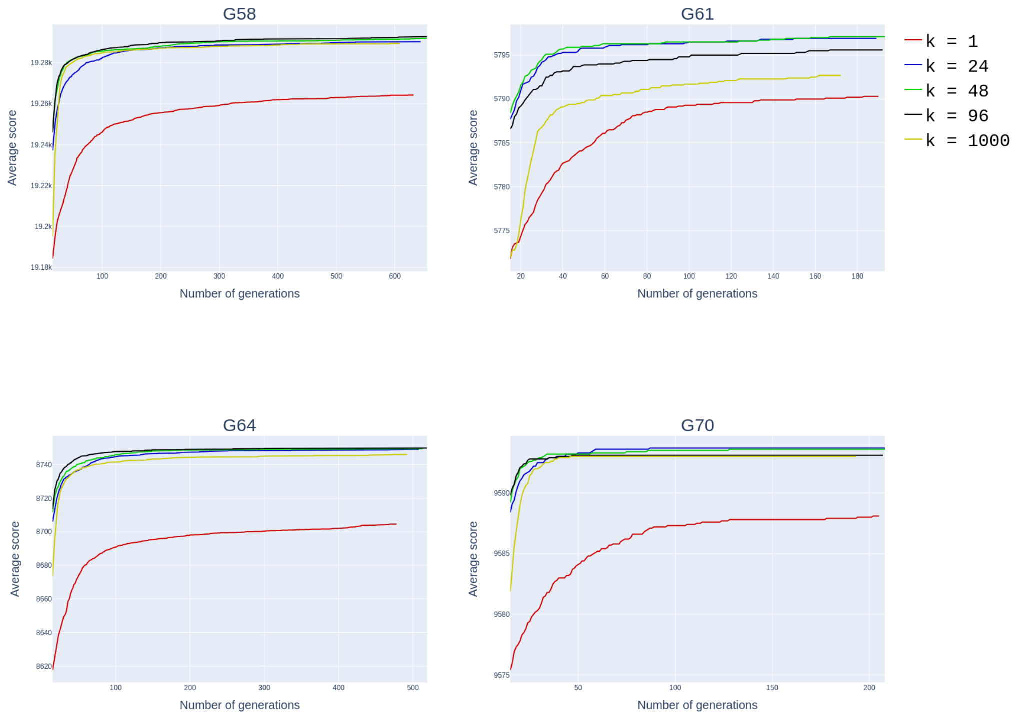

Fig. 4. Impact of different number of neighbors considered in the matching strategy in LPI: 1 (red), 24 (blue), 48 (green), 96 (black), 1000 (yellow)

4.4.4 Impact of the number of migrants

The parameter m determines the number of individuals (i.e., migrants) from each island that are sent to the next island in the one-way ring topology (as shown in Figure 5) at each generation.

First of all, we observe in Figure 5 that the version of the algorithm without migrations, when $m = 0$ (black curve), obtained the worst results for all four instances considered. This highlights the interest of the migrations, which spread the best in-

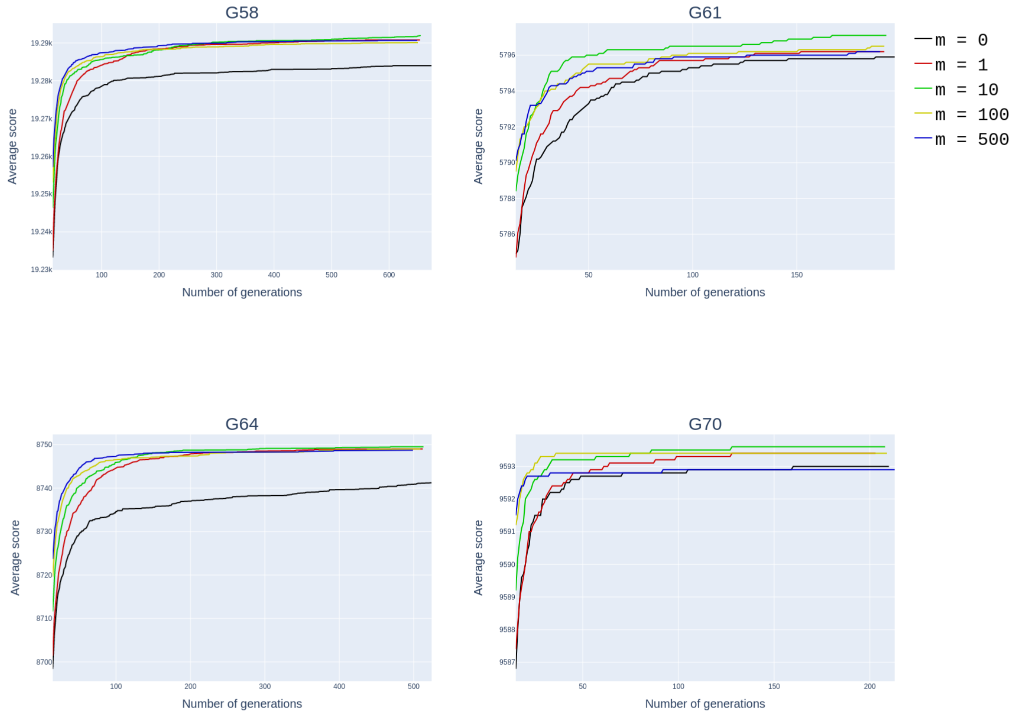

Fig. 5. Impact of different number of migrants between islands at each generation in LPI: 0 (black), 1 (red), 10 (green), 100 (yellow) and 500 (blue).

dividuals to other islands, and increase the chances of producing promising offspring in the following generations.

Conversely, if the number of migrants is too high, such as $m = 100$ or $m = 500$, the average score improves quickly during the first few generations due to the increased intensification of the algorithm. However, the algorithm stagnates more quickly due to a decrease in population diversity, especially for instances G61 and G70.

The best results are obtained for $m = 10$ (green curve), which strikes a good balance between intensification and diversification. This allows enough migration to maintain diversity in the population, while also allowing enough intensification to achieve the best results on average at the end of the search.

4.4.5 Impact of the migration topology

In the standard version of LPI, an unidirectional ring topology is chosen for migration, where $m = 10$ copies of the best individual from each island i are sent to island $i+1$. To study the impact of this unidirectional ring topology, we compare it with two other topologies, in which the total number of migrants sent and received by each island in each generation remains equal to m:

- a bidirectional ring topology, where $m' = 5$ copies of the best individual from island i are sent to both islands $i-1$ and $i+1$;
- a 1+2+3+4+5 ring topology (see [37]) where copies of the best individual from island i are sent to islands $i-5, \ldots, i-1, i+1, \ldots, i+5$.

We observe in Figure 6 that the unidirectional (green curve) and bidirectional (red curve) ring topologies give almost the same results. The one-way ring topology is slightly better in average for the instances G58 and G61. The 1+2+3+4+5 ring topology (blue curve) gives inferior results because it spreads the best individuals too quickly to all the other islands, resulting in less diversity in the overall population and thus worse performance.

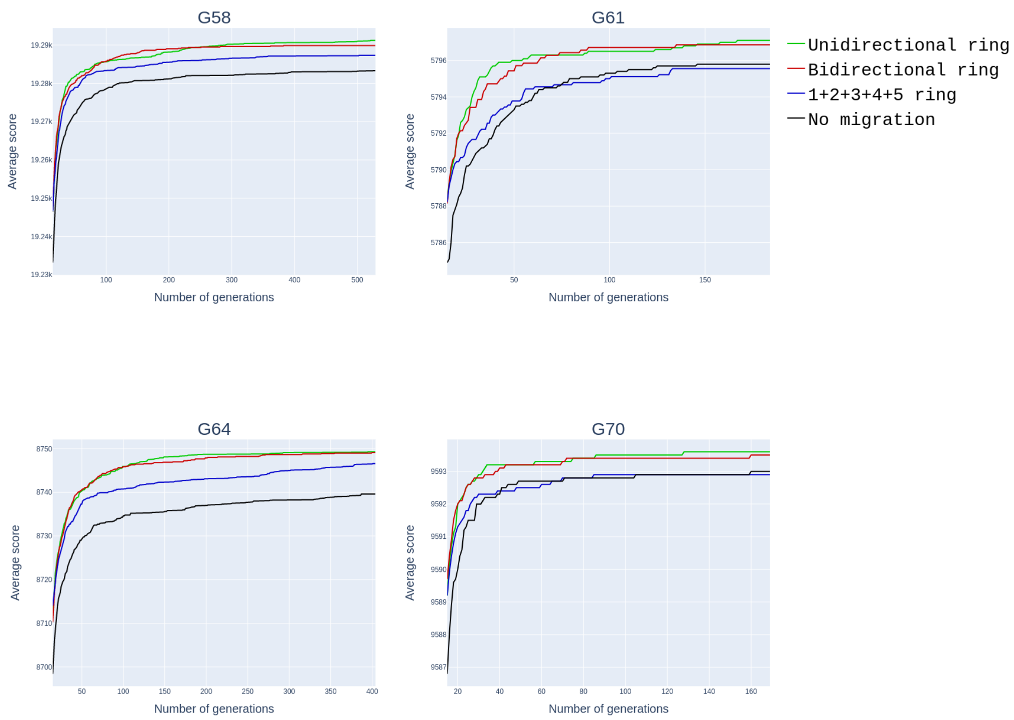

Fig. 6. Impact of the size $|p|$ of each island in LPI: 10 (red), 100 (blue), 1000 (green), 10000 (black).

4.4.6 *Analysis of the interaction between the number of migrants and the number of islands*

In this subsection, we present an analysis of the interaction between two important parameters of the algorithm: the number of islands I and the number m of migrants. For the experiments, we used a population with fixed size $|P| = 40000$ for the same instances G58, G61, G64, and G70. We performed a grid search with the number of islands I varying in $\{4, 40, 400, 4000\}$ and the number of migrants m varying in $\{0, 1, 10, 100\}$. The configuration with $m = 100$ and $I = 4000$ was excluded, as it would result in $p = 10$ individuals per island, fewer than the number of migrants. The average results over 10 independent runs for each of the fifteen (I, m) configurations are depicted in Figure 7.

First of all, this experiment reaffirms that disabling migrations ($m = 0$) yields inferior results across all island sizes.

Using a large number of islands ($I = 4000$), corresponding to a small number of individuals on each island ($p = 10$), also produces poorer results for all these instances. Nonetheless, the results improve when more migrants are sent between islands in each generation. This is because using too small islands reduces the chances of producing effective crossovers at each generation.

We also observe that for the instances G58, G64, and G70, when the number of islands is too low ($I = 4$) or when the number of migrants is too high ($m = 100$), this produces slightly worse results, since it reduces the diversity over generations in the overall population.

The configuration with $I = 40$ (corresponding to $p = 1000$ individuals on each island) and $m = 10$ is the most beneficial for G58, G64, and G70 instances launched for 3 hours. This configuration corresponds to the values chosen in Table 1 and used for the experiments reported in the next section.

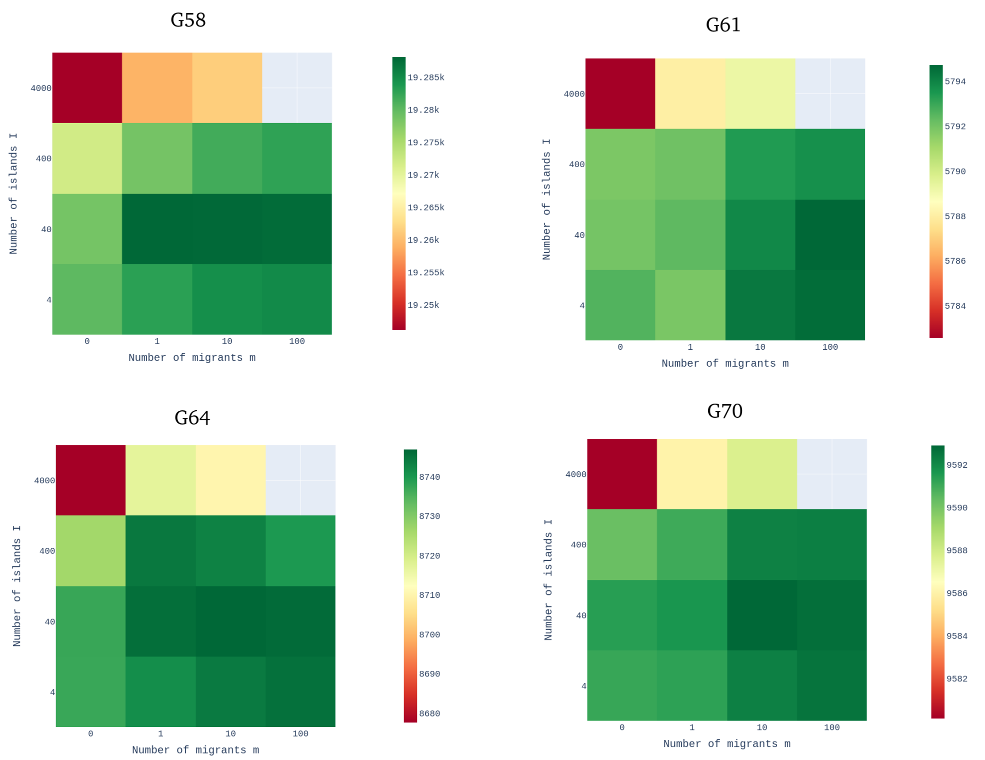

Fig. 7. Analysis of the interaction between the number of migrants and the number of islands

Table 2 reports the computational results of the LPI algorithm on the third set of maximum cut instances presented in Section 4.1. Each instance was solved independently 10 times with random seeds $0, 1, \dots, 9$. A time limit of 2 hours was used for the "small" instances (G1, ..., G54), while a limit of 20 hours was allowed for the larger instances (G55, ..., G81).

For small maximum cut instances, LPI can find all the best known results in the literature in a short time, except for the instance G23, for which a best score of 13344 was found only by the MOH algorithm in [25]. [9]

For these small instances, except for G36 and G37, the required computation time is less than 15 minutes and is always of the same order of magnitude as the computation times required by the BLS [4], PR [47] and MOH [25] algorithms. [10] LPI always finds the best solutions for the different runs launched, unlike the other algorithms which do not always get the best known scores for all runs or all instances. In particular, LPI, like other competing algorithms, easily finds the best solutions for the instances G43-G50 in a short time, because these instances are small and because the density of their Q matrix is low. For G36 and G37, LPI takes longer (up to two hours), but achieves better average scores than competing algorithms. For the largest instances (G55, ..., G81), the computation times required by LPI (up to 20h) and reported in the Table 2 are higher than for the reference methods PR, BLS, MOH and PF-SEL (from 2h to 5h), but LPI still achieves better average results with a large spread, and LPI finds six new lower bounds. The best results reported by the GESPR algorithm in Table 3 are of the same order of quality as those of LPI, but these results were obtained with an unknown experimental framework and computational time. A more precise comparison of computation time and number of objective function calls with the PR algorithm [47], whose source code are available to us, is given in Table 4.

For large maximum cut instances, LPI converges slowly but can find the best results of the literature except for five instances: G63, G67, G72, G77, G81. Remarkably, it obtains new lower bounds (marked with an asterisk) that have never been found in the literature for 5 instances: G58, G59, G62, G64 and G70. Note that other new bounds, not reported in Table 2, but in Table 3, were found during our calibration procedure (e.g., for the instance G61 with a new score of 5799). [11]

Table 3 presents the best results known for the 17 difficult maximum cut instances.

[9] We could not find the certificate for this solution or the MOH source code to reproduce this result.
[10] Algorithms GESPR [40] and PF-SEL [51] do not report results for these small instances.
[11] Certificates for these new solutions will be available on the github repository of the paper.

Instance			LPI			Instance			LPI		
Name	n	BKS	Best	Average	t (s)	Name	n	BKS	Best	Average	t (s)
G1	800	11624	**11624**	11624.0	7	G37	2000	7691	**7691**	7690.2	4082
G2	800	11620	**11620**	11620.0	8	G38	2000	7688	**7688**	7688.0	614
G3	800	11622	**11622**	11622.0	10	G39	2000	2408	**2408**	2408.0	347
G4	800	11646	**11646**	11646.0	7	G40	2000	2400	**2400**	2400.0	314
G5	800	11631	**11631**	11631.0	7	G41	2000	2405	**2405**	2405.0	286
G6	800	2178	**2178**	2178.0	14	G42	2000	2481	**2481**	2481.0	328
G7	800	2006	**2006**	2006.0	7	G43	1000	6660	**6660**	6660.0	19
G8	800	2005	**2005**	2005.0	10	G44	1000	6650	**6650**	6650.0	20
G9	800	2054	**2054**	2054.0	13	G45	1000	6654	**6654**	6654.0	19
G10	800	2000	**2000**	2000.0	10	G46	1000	6649	**6649**	6649.0	21
G11	800	564	**564**	564.0	11	G47	1000	6657	**6657**	6657.0	25
G12	800	556	**556**	556.0	16	G48	3000	6000	**6000**	6000.0	94
G13	800	582	**582**	582.0	23	G49	3000	6000	**6000**	6000.0	93
G14	800	3064	**3064**	3064.0	119	G50	3000	5880	**5880**	5880.0	90
G15	800	3050	**3050**	3050.0	80	G51	1000	3848	**3848**	3848.0	145
G16	800	3052	**3052**	3052.0	69	G52	1000	3851	**3851**	3851.0	119
G17	800	3047	**3047**	3047.0	104	G53	1000	3850	**3850**	3850.0	182
G18	800	992	**992**	992.0	40	G54	1000	3852	**3852**	3852.0	140
G19	800	906	**906**	906.0	49	G55	5000	10299	**10299**	10299.0	6594
G20	800	941	**941**	941.0	31	G56	5000	4017	**4017**	4016.9	49445
G21	800	931	**931**	931.0	32	G57	5000	3494	**3494**	3494.0	3494
G22	2000	13359	**13359**	13359.0	413	G58	5000	19293	**19294***	19292.0	65737
G23	2000	13344	13342	13342.0	150	G59	5000	6087	**6088***	6085.4	66512
G24	2000	13337	**13337**	13337.0	234	G60	7000	14190	**14190**	14189.4	44802
G25	2000	13340	**13340**	13340.0	258	G61	7000	5798	**5798**	5797.1	74373
G26	2000	13328	**13328**	13328.0	291	G62	7000	4870	**4872***	4870.0	26537
G27	2000	3341	**3341**	3341.0	152	G63	7000	27045	27033	27026.6	52726
G28	2000	3298	**3298**	3298.0	197	G64	7000	8751	**8752***	8749.5	49158
G29	2000	3405	**3405**	3405.0	293	G65	8000	5562	**5562**	5560.6	21737
G30	2000	3413	**3413**	3413.0	410	G66	9000	6364	**6364**	6362.0	34062
G31	2000	3310	**3310**	3310.0	412	G67	10000	6950	6948	6944.0	61556
G32	2000	1410	**1410**	1410.0	330	G70	10000	9591	**9594***	9593.6	28820
G33	2000	1382	**1382**	1382.0	349	G72	10000	7006	7004	6999.8	42542
G34	2000	1384	**1384**	1384.0	302	G77	14000	9938	9926	9924.6	66662
G35	2000	7686	**7686**	7686.0	1070	G81	20000	14048	14030	14026.4	66691
G36	2000	7680	**7680**	7680.0	5790						

Table 2
Detailed results of LPI on Max-Cut instances. Bold numbers indicate results that match the Best Known Score (BKS) of the literature. Results marked with an asterisk correspond to new lower bounds.

For each instance, the best score found by each algorithm is indicated (lower bound of the score). Note that this table reports the very best scores obtained by UBQP algorithms and dedicated maximum cut algorithms in the literature (to the best of our knowledge): the path-relinking algorithm (PR) [47], the breakout Local search (BLS) [4], the multiple search operator heuristic MOH [25], the GESPR algorithm [40], which consists of teams of global equilibrium search algorithms run in parallel in a multi-CPU environment, and the recent memetic algorithm PF-SEL [51].

A clear comparison in terms of computational time is difficult because some of these best results, such as those of GESPR, were found under unknown experimental conditions. In addition, we were unable to obtain the original source codes corresponding to the best reported results for the BLS, MOH, GESPR and PF-SEL algorithms.

Instance	n	BKS	LPI	2012 PR [47]	2013 BLS [4]	2015 GESPR [40]	2017 MOH [25]	2022 PF-SEL [51]
G55	5000	10299	**10299**	10265	10294	**10299**	**10299**	-
G56	5000	4017	**4017**	3981	4012	**4017**	4016	-
G57	5000	3494	**3494**	3472	3492	**3494**	**3494**	-
G58	5000	19293	**19294**⋆	19205	19263	19293	19288	-
G59	5000	6087	**6088**⋆	6027	6078	6086	6087	-
G60	7000	14190	**14190**	14112	14176	14188	**14190**	14187
G61	7000	5798	**5799**⋆	5730	5789	5796	5798	5792
G62	7000	4870	**4872**⋆	4836	4868	4870	4868	4868
G63	7000	27045	27042	26916	26997	**27045**	27033	26980
G64	7000	8751	**8752**⋆	8641	8735	8751	8747	8726
G65	8000	5562	**5562**	5526	5558	**5562**	5560	**5562**
G66	9000	6364	**6364**	6314	6360	**6364**	6360	6360
G67	10000	6950	6948	6902	6940	**6950**	6942	6946
G70	10000	9591	**9595**⋆	9463	9541	9591	9544	9587
G72	10000	7006	**7006**	6946	6998	**7006**	6998	7000
G77	14000	9938	9928	-	9926	**9938**	9928	9930
G81	20000	14048	14042	-	14030	**14048**	14036	14038

Table 3
Best scores found by state-of-the-art algorithms on difficult Max-Cut instances. Bold numbers indicate results equal to the Best Known Score (BKS) in the literature. An asterisk indicates a record-breaking new lower bound.

4.6 Ablation study and comparison with a baseline path-relinking algorithm

This section is dedicated to a careful comparison with the popular path-relinking (PR) algorithm of [47], for which we get the source code. The PR algorithm is a sequential hybrid algorithm with a population of 10 individuals. It uses the *one-flip* tabu search (TS_1) combined with the the path-relinking procedure, which are described in subsection 2.1 and 2.2, respectively.

We compare this baseline PR algorithm [47] with three different LPI variants:

- LPI-TS_1 + PR: a variant of LPI using exactly the same tabu search and combination procedure (with the same hyperparameters) as used in the PR algorithm of [47]. The only difference with [47] is the size of the population and the island organization.
- LPI-TS_1 + RTSC: an LPI variant using the same tabu search, but the new RTSC combination procedure is used instead of the path-relinking procedure of [47].
- LPI-$TS^*_{1|2}$ + RTSC: the full LPI algorithm described in this paper.

In order to compare the PR algorithm of [47] with the LPI variants on the same basis, and to remove the effect of using different computing platforms, we compare these variants with the same maximum budget of 20 billion iterations spent in the local tabu search.

To perform these 20 billion iterations, for example for the G55 instance of size $n = 5000$, the PR algorithm of [47] takes 7 days on an Intel Xeon ES 2630, 2.66 GHz CPU, while it takes only about 1 hour with the LPI algorithm whose local searches are parallelized on the Nvidia V100 GPU.

Table 4 shows the comparison of PR with the three LPI variants. Columns 1 and 2 are the name and the size n of the instance. Columns 3-6 show the results of the PR algorithm [47], while columns 7-18 display the results of the LPI variants. The best and average scores over 10 independent runs are shown, as well as the average number of iterations and the time (in hours) required to find the best solutions.

Instance		TS_1 + PR - 10 individuals [47]				LPI-TS_1 + PR				LPI-TS_1 + RTSC				LPI-$TS^*_{1\|2}$ + RTSC			
Name	n	best	avg.	#iter ($\times 10^9$)	time CPU h.	best	avg.	#iter ($\times 10^9$)	time GPU h.	best	avg.	#iter ($\times 10^9$)	time GPU h.	best	avg.	#iter ($\times 10^9$)	time GPU h.
G55	5000	10295	10280.2	8.0	68.3	10295	**10290.7**	17.6	0.8	10296	**10292.8**	16.5	0.9	10299	**10299.0**	12.1	1.8
G56	5000	4011	3995.5	8.2	70.5	4011	**4009.7**	16.9	0.8	4014	**4011.6**	17.0	0.9	4015	**4014.3**	15.4	2.2
G57	5000	3491	3488.4	6.6	56.7	3494	**3492.6**	17.0	0.8	3494	**3494.0**	14.2	0.8	3494	**3493.8**	13.5	1.6
G58	5000	19239	19225.8	8.2	72.3	19266	**19247.5**	18.6	0.9	19278	**19275.6**	18.7	1.0	-	-	-	-
G59	5000	6054	6036.4	6.0	51.6	6079	**6073.1**	18.9	0.9	6081	**6079.3**	18.4	1.0	-	-	-	-
G60	7000	14179	14146.7	8.1	69.6	14175	**14170.8**	18.4	1.1	14178	**14176.6**	18.2	1.3	14189	**14186.0**	16.4	3.4
G61	7000	5773	5759.6	7.6	78.6	5779	**5776.0**	18.1	1.1	5788	**5783.3**	18.4	1.3	5797	**5794.5**	17.8	3.7
G62	7000	4865	4858.8	6.5	55.8	4866	**4863.0**	18.0	1.1	4870	**4868.2**	16.2	1.1	4870	**4868.0**	15.3	2.6
G63	7000	26959	26941.8	10.8	92.9	26960	**26956.3**	19.1	1.2	27011	**27005.3**	19.0	1.4	-	-	-	-
G64	7000	8686	8651.5	9.1	95.2	8736	**8720.9**	18.6	1.1	8742	**8739.0**	17.8	1.3	-	-	-	-
G65	8000	5552	5547.8	15.4	226.3	5552	**5550.6**	18.1	1.2	5560	**5559.3**	15.7	1.2	5560	**5558.2**	16.0	3.1
G66	9000	6351	6339.4	6.7	88.4	6348	**6345.0**	18.2	1.4	6360	**6357.5**	16.9	1.5	6364	**6358.2**	16.4	3.6
G67	10000	6930	6926.4	11.8	202.0	6930	6928.0	18.0	1.6	6942	**6940.6**	16.6	2.1	6942	**6940.4**	17.2	4.1
G70	10000	9538	9526.2	14.1	218.6	9526	9513.6	19.0	1.7	9569	**9564.9**	18.4	1.9	9594	**9592.9**	15.5	3.0
G72	10000	6988	6980.8	12.5	213.3	6984	6980.6	18.2	1.6	6998	**6994.4**	17.9	2.6	6998	**6996.2**	17.5	4.1
G77	14000	9911	9906.6	13.3	322.9	9900	9895.0	18.4	2.5	9920	**9917.2**	18.3	4.0	9922	**9918.4**	18.4	7.5
G81	20000	14018	14009.8	15.3	531.4	13964	13960.0	12.7	3.1	14018	**14015.0**	18.6	6.6	14022	**14016.4**	18.8	12.1

Table 4
Comparison of three LPI variants with the baseline path-relinking algorithm of [47]. All algorithms were executed with a maximum of 20 billion tabu search iterations. Significantly better average results (t-test with p-value 0.001) of the LPI variant compared to the baseline PR variant of [47] are shown in boldface. The best average result of each row is underlined.

When comparing the baseline PR algorithm (columns 3-6) with the first LPI variant (LPI-TS_1 + PR, columns 7-10), which uses the same components for the local search and the combination procedure (the only difference is the large population of LPI), we observe that the LPI variant is better on average (+0.22%) for the smaller instances (G55-G67), while the PR algorithm is better in average (+0.15%) for the larger instances (G70-G81). For small instances, it seems more beneficial to encourage more exploration with a large population, to avoid getting stuck in local optima. Conversely, when the instance is large and when a global total number of iterations is imposed, LPI does not have the time to learn promising areas in the search space and it seems more beneficial to favor more intensification (with a smaller population).

Replacing the path-linking procedure with the RTSC procedure (columns 11-14) improves the results for all instances with the same budget of total number of iterations devoted to local search. This improvement is in average of 0.17% for the 17 instances of this table. An improvement of 0.17% may seem small, but it is very difficult to improve by just one point a UBQP score that is already close to the best lower bound found in the literature.

This shows that explicitly optimizing a trade-off between the quality of the offspring and its distance from both parents, as in the RTSC combination procedure (see Section 3.4), is more advantageous than performing greedy moves that link both parents as in the original path-relinking procedure of [47].

If the tabu search *one-or-two-flip* $TS^*_{1|2}$ is used (columns 15-18) instead of the classical search *one-flip* TS_1, it significantly improves the results for some instances, such as G55 (+0.06%), G56 (+0.07 %), G60 (+0.07 %), G61 (+0.19 %), and G70 (+0.29 %), which are characterized by a low-density Q-matrix. Note that no results are reported for the instances G58, G59, G63 and G64 with this variant LPI-$TS^*_{1|2}$ + RTSC in columns 15-18 because their density ratio Δ/n is higher than the density threshold ρ (cf. Section 3.2).

This improvement achieved with the *one-or-two-flip* tabu search, $TS^*_{1|2}$, instead of the classical *one-flip* search, TS_1, is reported in Table 4, where both algorithms were subjected to the same total number of 20 billion iterations. However, an iteration with the *one-or-two-flip* tabu search takes more time to complete because the size of the neighborhood evaluated at each iteration with $TS^*_{1|2}$ is larger. Therefore, for a clearer comparison of the TS_1 and $TS^*_{1|2}$ tabu search algorithms evaluated with the same time budget, we launched 10 replications of the two variants, LPI-TS_1+RTSC and LPI-$TS^*_{1|2}$+RTSC, on the instances G55, G56, G60, G61 and G70, with the same total time budget of 3 hours. The results obtained are reported in Table 5. We see that using the *one-or-two-flip* tabu search, $TS^*_{1|2}$, significantly improves the results for the instances G55 (+0.035%), G56 (+0.035 %), G60 (+0.036 %), G61 (+0.12 %), and G70 (+0.18 %), when the same time budget is used for both variants.

Instances			LPI- TS_1+ RTSC		LPI- $TS^*_{1\|2}$ + RTSC		
Name	n	BKS	Best	Average	Best	Average	Avg. Spread
G55	5000	10299	10298	10295.4	10299	**10299.0**	+3.6
G56	5000	4017	4016	4014.6	4016	**4016.0**	+1.4
G60	7000	14190	14187	14180.8	14187	**14185.9**	+5.1
G61	7000	5798	5793	5788.5	5797	**5795.6**	+7.1
G70	10000	9591	9578	9575.1	9594	**9592.7**	+17.6

Table 5
All algorithms were executed with a time budget of 3 hours. Significantly better average results (t-test with p-value 0.01) of the LPI with $TS^*_{1|2}$ tabu search compared to the LPI with TS_1 tabu search are shown in boldface. The best average result of each row is underlined.

5 Conclusion

In this work, we investigated a large population island algorithm applied to the unconstrained binary quadratic problem. We studied the impact of several critical parameters of the algorithm, such as the matching strategy, the combination procedure, and the migration topology.

Our experimental results lead to four conclusions. First, it is interesting for this problem to use a k-nearest neighbor strategy to select parents for crossover, instead of randomly selecting pairs of individuals from the population. This parameter k, describing the number of neighbors, must be chosen carefully to achieve a good exploration/exploitation trade-off. Second, we highlight the advantage of an island organization with migrations, which has an impact on search intensification for this problem. Third, for some instances with low-density Q-matrix, we shed light on the advantage of using a sparse tabu search considering both one-flip and two-flip associated to non-zero entries of the Q-matrix. Fourth, we highlight the value of the newly proposed RTSC combination procedure compared to conventional path-relinking or uniform crossovers. This RTSC procedure transmits similar genetic information from both parents, but also explicitly optimizes a trade-off between the quality of the offspring and its distance from both parents, which are two expected properties of an efficient crossover.

This research has led to the discovery of 6 new lower bounds for hard instances of the maximum cut problem, which have never been found before in the literature. The proposed framework with a large population, executed on GPUs, is quite general and could be applied to solve other NP-hard problems. By making the source code of our algorithm available, we hope to facilitate such applications and encourage further research.

Future work could include using more sophisticated local search algorithms that explore different neighborhoods, or include improved matching and combination strategies.

Acknowledgment

We would like to thank Prof. Y. Wang and the co-authors of [47] for sharing the code of their path-relinking algorithm, and Prof. F. Glover for sharing his insightful thoughts with us. We are also grateful to the reviewers for their valuable comments, which helped us to improve the paper. This work has benefited from the HPC resources of IDRIS - GENCI (Grant No. A0130611887) and was supported by the Deep Meta project (Etoiles Montantes, Pays de la Loire) and the COMBO project (ANR-23-CE23-0015, French National Research Agency).

References

[1] Bahram Alidaee, Gary Kochenberger, and Ahmad Ahmadian. 0-1 quadratic programming approach for optimum solutions of two scheduling problems. *International Journal of Systems Science*, 25(2):401–408, 1994.

[2] Talal M Alkhamis, Merza Hasan, and Mohamed A Ahmed. Simulated annealing for the unconstrained quadratic pseudo-boolean function. *European Journal of Operational Research*, 108(3):641–652, 1998.

[3] Francisco Barahona. On the computational complexity of ising spin glass models. *Journal of Physics A: Mathematical and General*, 15(10):3241, 1982.

[4] Una Benlic and Jin-Kao Hao. Breakout local search for the max-cutproblem. *Engineering Applications of Artificial Intelligence*, 26(3):1162–1173, 2013.

[5] Endre Boros and Peter L Hammer. The max-cut problem and quadratic 0–1 optimization; polyhedral aspects, relaxations and bounds. *Annals of Operations Research*, 33(3):151–180, 1991.

[6] Marcelo de Souza and Marcus Ritt. Automatic grammar-based design of heuristic algorithms for unconstrained binary quadratic programming. In *Proceedings of 18th European Conference on Evolutionary Computation in Combinatorial Optimization, Parma, Italy, April 4–6, 2018*, pages 67–84. Springer, 2018.

[7] Fred Glover and Jin-Kao Hao. Efficient evaluations for solving large 0-1 unconstrained quadratic optimisation problems. *International Journal of Metaheuristics*, 1(1):3–10, 2010.

[8] Fred Glover and Jin-Kao Hao. Fast two-flip move evaluations for binary unconstrained quadratic optimisation problems. *International Journal of Metaheuristics*, 1(2):100–107, 2010.

[9] Fred Glover, Gary Kochenberger, and Yu Du. Applications and computational advances for solving the qubo model. In Abraham P. Punnen, editor, *The Quadratic Unconstrained Binary Optimization Problem: Theory, Algorithms, and Applications*, pages 39–56. Springer, 2022.

[10] Fred Glover, Gary Kochenberger, Rick Hennig, and Yu Du. Quantum bridge analytics I: a tutorial on formulating and using QUBO models. *Annals of Operations Research*, pages 1–43, 2022.

[11] Fred Glover, Manuel Laguna, and Rafael Martí. Fundamentals of scatter search and path relinking. *Control and Cybernetics*, 29(3):653–684, 2000.

[12] Fred Glover, Zhipeng Lü, and Jin-Kao Hao. Diversification-driven tabu search for unconstrained binary quadratic problems. *4OR*, 8(3):239–253, 2010.

[13] Olivier Goudet, Cyril Grelier, and Jin-Kao Hao. A deep learning guided memetic framework for graph coloring problems. *Knowledge-Based Systems*, 258:109986, 2022.

[14] Olivier Goudet and Jin-Kao Hao. Massively parallel hybrid search for the partial latin square extension problem. *arXiv preprint arXiv:2103.10453*, 2021.

[15] Nicolo Gusmeroli, Timotej Hrga, Borut Lužar, Janez Povh, Melanie Siebenhofer, and Angelika Wiegele. Biqbin: a parallel branch-and-bound solver for binary quadratic problems with linear constraints. *ACM Transactions on Mathematical Software (TOMS)*, 48(2):1–31, 2022.

[16] Bahman Kalantari and Ansuman Bagchi. An algorithm for quadratic zero-one programs. *Naval Research Logistics*, 37(4):527–538, 1990.

[17] Kengo Katayama and Hiroyuki Narihisa. Performance of simulated annealing-based heuristic for the unconstrained binary quadratic programming problem. *European Journal of Operational Research*, 134(1):103–119, 2001.

[18] Gary Kochenberger, Fred Glover, Bahram Alidaee, and Cesar Rego. An unconstrained quadratic binary programming approach to the vertex coloring problem. *Annals of Operations Research*, 139(1):229–241, 2005.

[19] Gary Kochenberger, Fred Glover, Bahram Alidaee, and Haibo Wang. Clustering of microarray data via clique partitioning. *Journal of Combinatorial Optimization*, 10(1):77–92, 2005.

[20] Gary Kochenberger, Jin-Kao Hao, Fred Glover, Mark Lewis, Zhipeng Lü, Haibo Wang, and Yang Wang. The unconstrained binary quadratic programming problem: a survey. *Journal of Combinatorial Optimization*, 28(1):58–81, 2014.

[21] Duan Li, XL Sun, and CL Liu. An exact solution method for unconstrained quadratic 0–1 programming: a geometric approach. *Journal of Global Optimization*, 52(4):797–829, 2012.

[22] Bas Lodewijks. Mapping NP-hard and NP-complete optimisation problems to quadratic unconstrained binary optimisation problems. *arXiv preprint arXiv:1911.08043*, 2019.

[23] Zhipeng Lü, Fred Glover, and Jin-Kao Hao. A hybrid metaheuristic approach to solving the UBQP problem. *European Journal of Operational Research*, 207(3):1254–1262, 2010.

[24] Zhipeng Lü, Fred Glover, and Jin-Kao Hao. Neighborhood combination for unconstrained binary quadratic problems. In *MIC Post-Conference Book*, pages 49–61. Citeseer, 2011.

[25] Fuda Ma and Jin-Kao Hao. A multiple search operator heuristic for the max-k-cut problem. *Annals of Operations Research*, 248(1):365–403, 2017.

[26] Peter Merz and Bernd Freisleben. Genetic algorithms for binary quadratic programming. In *Proceedings of the 1999 Genetic and Evolutionary Computation Conference*, volume 1, pages 417–424. Morgan Kaufmann Orlando, FL, 1999.

[27] Peter Merz and Bernd Freisleben. Greedy and local search heuristics for unconstrained binary quadratic programming. *Journal of Heuristics*, 8:197–213, 2002.

[28] Peter Merz and Kengo Katayama. Memetic algorithms for the unconstrained binary quadratic programming problem. *BioSystems*, 78(1-3):99–118, 2004.

[29] Hartmut Neven, Geordie Rose, and William G Macready. Image recognition with an adiabatic quantum computer I. mapping to quadratic unconstrained binary optimization. *arXiv preprint arXiv:0804.4457*, 2008.

[30] Gintaras Palubeckis. Multistart tabu search strategies for the unconstrained binary quadratic optimization problem. *Annals of Operations Research*, 131:259–282, 2004.

[31] Panos M Pardalos and Gregory P Rodgers. Computational aspects of a branch and bound algorithm for quadratic zero-one programming. *Computing*, 45(2):131–144, 1990.

[32] Panos M Pardalos and Jue Xue. The maximum clique problem. *Journal of Global Optimization*, 4(3):301–328, 1994.

[33] Daniel Cosmin Porumbel, Jin-Kao Hao, and Pascale Kuntz. An evolutionary approach with diversity guarantee and well-informed grouping recombination for graph coloring. *Computers & Operatios Research*, 37(10):1822–1832, 2010.

[34] Abraham P Punnen. *The Quadratic Unconstrained Binary Optimization Problem: Theory, Algorithms, and Applications.* Springer, 2022.

[35] Franz Rendl, Giovanni Rinaldi, and Angelika Wiegele. Solving max-cut to optimality by intersecting semidefinite and polyhedral relaxations. *Mathematical Programming*, 121:307–335, 2010.

[36] Pooya Ronagh, Brad Woods, and Ehsan Iranmanesh. Solving constrained quadratic binary problems via quantum adiabatic evolution. *Quantum Information & Computation*, 16(11-12):1029–1047, 2016.

[37] Marek Ruciński, Dario Izzo, and Francesco Biscani. On the impact of the migration topology on the island model. *Parallel Computing*, 36(10-11):555–571, 2010.

[38] Michele Samorani, Yang Wang, Zhipeng Lv, and Fred Glover. Clustering-driven evolutionary algorithms: an application of path relinking to the quadratic unconstrained binary optimization problem. *Journal of Heuristics*, 25(4):629–642, 2019.

[39] Jialong Shi, Qingfu Zhang, Bilel Derbel, and Arnaud Liefooghe. A parallel tabu search for the unconstrained binary quadratic programming problem. In *2017 IEEE Congress on Evolutionary Computation (CEC)*, pages 557–564. IEEE, 2017.

[40] Vladimir Shylo, Fred Glover, and IV Sergienko. Teams of global equilibrium search algorithms for solving the weighted maximum cut problem in parallel. *Cybernetics and Systems Analysis*, 51(1):16–24, 2015.

[41] Vladimir Shylo and Oleg V Shylo. Solving the maxcut problem by the global equilibrium search. *Cybernetics and Systems Analysis*, 46:744–754, 2010.

[42] Vladimir Shylo and Oleg V Shylo. Systems analysis; solving unconstrained binary quadratic programming problem by global equilibrium search. *Cybernetics and Systems Analysis*, 47(6):889–897, 2011.

[43] Renato Tinós, Darrell Whitley, and Francisco Chicano. Partition crossover for pseudo-boolean optimization. In *Proceedings of the 2015 ACM Conference on Foundations of Genetic Algorithms XIII*, pages 137–149, 2015.

[44] Haibo Wang, Bahram Alidaee, Fred Glover, and Gary Kochenberger. Solving group technology problems via clique partitioning. *International Journal of Flexible Manufacturing Systems*, 18(2):77–97, 2006.

[45] Yang Wang and Jin-Kao Hao. Metaheuristic algorithms. In Abraham P Punnen, editor, *The Quadratic Unconstrained Binary Optimization Problem: Theory, Algorithms, and Applications*, pages 241–259. Springer, 2022.

[46] Yang Wang, Zhipeng Lü, Fred Glover, and Jin-Kao Hao. Effective variable fixing and scoring strategies for binary quadratic programming. In *Proceedings of the 11th European Conference Evolutionary Computation in Combinatorial Optimization, Torino, Italy, April 27-29, 2011*, pages 72–83. Springer, 2011.

[47] Yang Wang, Zhipeng Lü, Fred Glover, and Jin-Kao Hao. Path relinking for unconstrained binary quadratic programming. *European Journal of Operational Research*, 223(3):595–604, 2012.

[48] Yang Wang, Zhipeng Lü, Fred Glover, and Jin-Kao Hao. Backbone guided tabu search for solving the UBQP problem. *Journal of Heuristics*, 19:679–695, 2013.

[49] Yang Wang, Zhipeng Lü, Fred Glover, and Jin-Kao Hao. Probabilistic grasp-tabu search algorithms for the UBQP problem. *Computers & Operations Research*, 40(12):3100–3107, 2013.

[50] Darrell Whitley, Soraya Rana, and Robert B Heckendorn. The island model genetic algorithm: On separability, population size and convergence. *Journal of Computing and Information Technology*, 7(1):33–47, 1999.

[51] Zhi-zhong Zeng, Zhipeng Lü, Xin-guo Yu, Qing-hua Wu, Yang Wang, and Zhou Zhou. A memetic algorithm based on edge-state learning for max-cut. *Expert Systems with Applications*, 209:118077, 2022.

A Parameter range for grid search

Table A.1 displays the parameter choices for grid search used in the calibration procedure of Section 4.3.

Parameter	Description	Choice
Population		
p	Sub-population size	$[100, 1000]$
γ	Minimum spacing	$[n/10, n/20]$
m	Number of migrants	$[1, 10]$
Local search		
N_L	Number of iterations	$[n, 2n]$
α	Tabu tenure parameter for Max-cut	$[0.02, 0.04, 0.1]$
Combination procedure		
k	Number of neighbors	$[1.48, p]$
N_C	Number of iterations	$[n/2, n]$
κ	Trade-off parameter	$[0.5, 1.2]$

Table A.1
Parameter choices for grid search with LPI.

B Results on Palubeckis instances

Table B.1 displays the results of LPI on the set of 21 UBQP instances of [30]. For this set of instances, we use the parameters described in Table 1 with the exception of α which is set to 0.01 (which was successfully used by [47] for these instances).

Column 1 is the name of the instance, Column 2 is the size n ranging from 3000 to 7000. Column 3, BKS, reports the best known score found in the literature for each instance [38,46,47]. Columns 4, 5 and 6 report the LPI results with the best and average scores obtained over 10 independent runs and the average time required in seconds to obtain the best scores.

We observe that LPI always finds the best known score on each independent run. However the computational time required to obtain it is quite high in average for the largest instances.

Instance			LPI		
Name	n	BKS	Best	Average	t (s)
p3000.1	3000	3931583	**3931583**	**3931583.0**	473
p3000.2	3000	5193073	**5193073**	**5193073.0**	635
p3000.3	3000	5111533	**5111533**	**5111533.0**	637
p3000.4	3000	5761822	**5761822**	**5761822.0**	495
p3000.5	3000	5675625	**5675625**	**5675625.0**	498
p4000.1	4000	6181830	**6181830**	**6181830.0**	860
p4000.2	4000	7801355	**7801355**	**7801355.0**	1185
p4000.3	4000	7741685	**7741685**	**7741685.0**	1192
p4000.4	4000	8711822	**8711822**	**8711822.0**	954
p4000.5	4000	8908979	**8908979**	**8908979.0**	976
p5000.1	5000	8559680	**8559680**	**8559680.0**	1591
p5000.2	5000	10836019	**10836019**	**10836019.0**	2195
p5000.3	5000	10489137	**10489137**	**10489137.0**	4901
p5000.4	5000	12252318	**12252318**	**12252318.0**	3967
p5000.5	5000	12731803	**12731803**	**12731803.0**	2296
p6000.1	6000	11384976	**11384976**	**11384976.0**	5704
p6000.2	6000	14333855	**14333855**	**14333855.0**	8807
p6000.3	6000	16132915	**16132915**	**16132915.0**	6751
p7000.1	7000	14478676	**14478676**	**14478676.0**	21481
p7000.2	7000	18249948	**18249948**	**18249948.0**	17164
p7000.3	7000	20446407	**20446407**	**20446407.0**	11956

Table B.1
Results of LPI on the UBQP instances of [30]

C Results on cubic lattices instances

Table C.1 displays the results of LPI on the set of 10 cubic lattices instances. Column 1 is the name of the instance, Column 2 is the size $n = 2744$. Column 3 reports the best known score found in the literature for each instance [25,41,51]. LPI always obtains the best known score for all instances and all independent runs with reasonable average computational times.

Instance			LPI		
Name	n	BKS	Best	Average	t (s)
sg3dl141000	2744	2446	**2446**	**2446.0**	614
sg3dl142000	2744	2458	**2458**	**2458.0**	601
sg3dl143000	2744	2444	**2444**	**2444.0**	3491
sg3dl144000	2744	2450	**2450**	**2450.0**	1012
sg3dl145000	2744	2446	**2446**	**2446.0**	536
sg3dl146000	2744	2452	**2452**	**2452.0**	823
sg3dl147000	2744	2444	**2444**	**2444.0**	587
sg3dl148000	2744	2448	**2448**	**2448.0**	940
sg3dl149000	2744	2428	**2428**	**2428.0**	1287
sg3dl1410000	2744	2460	**2460**	**2460.0**	1967

Table C.1
Results of LPI on cubic lattices instances.

D Average Hamming distance on each island and between islands

In this appendix, we report the results of an experiment in which we ran the LPI algorithm 10 times for the G70 instance with 10000 variables for 3 hours. In Figure D.1 is displayed in green the average Hamming distance between individuals on each island measured over generations, and in blue the average Hamming distance between individuals in the general population (not necessarily belonging to the same islands). Note that both curves start at the value 5000, which corresponds to the average Hamming distance between random individuals in the population for this instance with 10000 binary variables. Each color range corresponds to an interval of plus or minus one standard deviation around the mean.

With this experiment, we observe that the average distance between individuals on each island measured over the generations decreases more rapidly than the average distance between individuals in the general population, which can be explained by the fact that recombinations between individuals take place within each island, which favors a rapprochement of individuals within each island.

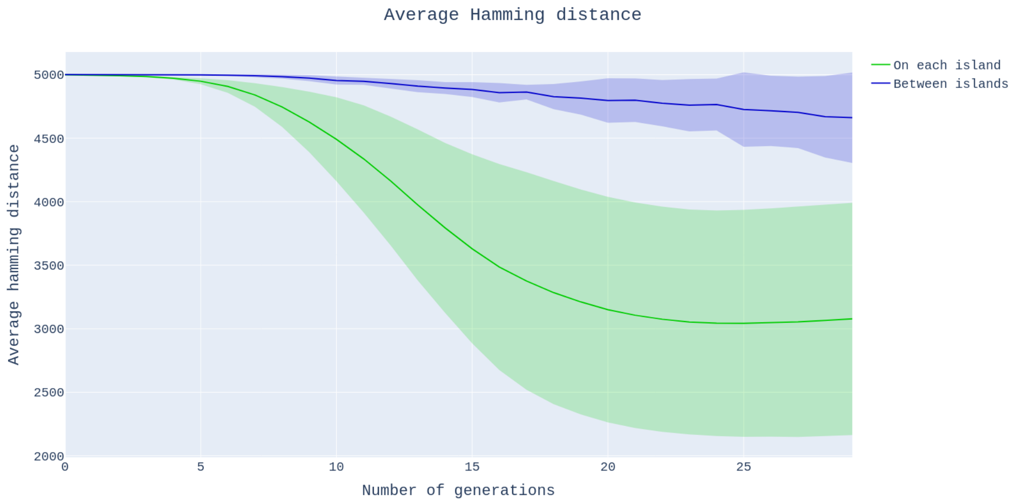

Fig. D.1. Average Hamming distance computed over the generations on each island (green) and between islands (blue) during the resolution of the instance G70.