Master in High Performance Computing

Parallel Implementation and Scalability Analysis of Algorithms Solving QUBO Problems in HPC Systems

Supervisor(s):
Daniele Ottaviani, CINECA,
Sara Marzella, CINECA,
Irina Davydenkova, ICTP

Candidate:
Zakaria Dahbi

10th edition
2023–2024

Abstract

Over time, tremendous efforts have been devoted to developing algorithms for solving complex optimization problems. These problems require finding the best possible solutions in a large search space, which is computationally challenging. For many such problems, no algorithm is currently known to solve them in polynomial time. Despite progress, existing methods often struggle with scalability or fail to find good solutions quickly. In this work, we use high-performance computing techniques to implement Simulated Annealing and Scatter Search for solving QUBO optimization problems. These algorithms achieve significant speedups and efficiently explore the large search space. We also study quantum annealing as a promising quantum algorithm on quantum machines (QPUs). We benchmark our solvers against the NP-hard MaxCut problem and analyze the results collected using the Leonardo supercomputer. As a final milestone, we introduce AlmoQUBO, a high-performance library for solving QUBO problems that can be represented as graphs.

Acknowledgments

I would like to express my appreciation to the ICTP for providing me with a full scholarship to participate in the MHPC programme. I am greatly grateful to Ivan Girotto for his availability, for his discussions, for his support, and for writing letters of recommendation.

I am sincerely grateful to CINECA Quantum Computing Lab for the opportunity to do this internship with them. I am very grateful to Daniele Ottaviani and Sara Marzella for their guidance, discussions, limitless support, and recommendations, which lead to the accomplishment of this work. Without forgetting to thank Gabriella Bettonte for the guidance during the early days of starting this work.

I would like to thank Irina Davydenkova for her availability, insightful discussions, helpful quick feedback, and for writing letters of recommendation.

I would like to thank all of the MHPC staff and lecturers for this incredible opportunity to learn very recent, cutting-edge technologies.

I would like to thank all of the dedicated staff at ICTP and SISSA for their assistance with administrative matters.

I also thank all my MHPC colleagues for the collaborative and friendly international atmosphere. We were a large, loving family.

Lastly, I want to express my profound appreciation to my parents and siblings for their encouragement and unlimited support. I love you so much.

Contents

List of Figures

List of Tables

List of Abbreviations

HPC High-Performance Computing

GPU Graphics Processing Unit

CPU Central Processing Unit

QPU Quantum Processing Unit

SA Simulated Annealing

SS Scatter Search

QA Quantum Annealing

Qubit Quantum Bit

ObjVal Objective Value

QUBO Quadratic Unconstrained Binary Optimization

Gset Graph Set

G Graph

NP Not Polynomial

$|V|$ Cardinality of Vertices

$|E|$ Cardinality of Edges

MaxCut Maximum Cut

LS Local Search

ALG Algorithm

minDelta Minimum Delta

maxDepth Maximum Depth

csize Cluster Size

PR Path Relinking

Introduction

Optimization problems are prevalent across various sectors, from scientific research to engineering and economics. However, solving these problems is a complex task, and achieving accurate solutions has the potential to unlock significant breakthroughs in both research and industry. Due to their broad applicability, optimization problems frequently involve large datasets, complex constraints, and conflicting objectives. These factors make them computationally demanding, particularly in real-world applications where the problem size and complexity grow exponentially. While several advanced techniques were proposed in the literature, they often struggle to provide optimal solutions within reasonable time constraints.

Among these advanced techniques, metaheuristics and quantum-based approaches have gained significant attention for solving complex optimization problems. Some metaheuristics, such as Simulated Annealing and Scatter Search, are particularly effective at exploring large solution spaces and at escaping local optima. Simulated annealing, introduced by Kirkpatrick et *al*. [5], inspired by thermodynamic processes. The algorithm works by probabilistically accepting worse solutions to avoid local minima by gradually converging toward an optimal solution. Recently, Simulated Annealing was extensively studied and applied for solving several optimization problems such as finding the MaxCut of a graph [1, 6], minimizing the cost of the traveling salesman [7] and multiple traveling salesmen problems [8], and optimal route detection between objects [9]. On the other hand, Scatter Search [10, 2] combines diverse solutions in a population-based approach, systematically improving the search for optimal outcomes.

Additionally, quantum-based approaches, particularly quantum annealing, have emerged as promising alternatives for solving optimization problems more efficiently, especially in large, complex search spaces [11, 12, 13, 14]. Quantum annealing offers a distinctive method for finding the global minimum by exploring the solution space through quantum tunneling. The core principle of quantum annealing involves transforming an initial state into a more complex Hamiltonian representing the optimization problem. The initial state is typically the ground state of an easily solvable problem. A key advantage of quantum annealing is its ability to solve certain optimization problems more efficiently than any classical method, especially when the energy landscape is full of local minima.

In this work, we focused on employing HPC paradigms to develop advanced C++ implementations of Simulated Annealing and Scatter Search for solving complex combinatorial optimization problems. We proposed a parallelized, heating-based, Simulated Annealing algorithm optimized for CPU execution. On the other hand, Scatter Search was accelerated using GPU resources, with novel approaches used to

improve both speed and efficiency. We investigate the potential of quantum annealing as an emerging quantum-based alternative. We explored D-Wave QPUs and assessed their potential of solving optimization problems formulated as QUBO. Our implementations are benchmarked against the well-known MaxCut problem [15, 16] to assess their performance in delivering optimal solutions. The goal is to compare the computational efficiency and effectiveness of these methods using HPC resources with the result achieved in the literature to ensure that they provide adequate solutions within acceptable elapsed time. As the final milestone of the work, we created a library named AlmoQUBO, which includes the implemented solvers and provides a straightforward framework for solving optimization problems that can resemble the MaxCut problem.

This thesis is split into five chapters. In Chapter 1, we focus on the Simulated Annealing algorithm. We review the state-of-the-art and then proceed to explaining our proposed parallel algorithmic implementation. In Chapter 2, we introduce Scatter Search, with a particular focus on our GPU-enabled version and two new algorithms for performing local refinement. In Chapter 4, we examine quantum annealing and provide a quick overview of the QUBO formalism. We also explain submitting jobs to D-Wave QPUs. In Chapter 3, we present AlmoQUBO, a high-performance and user-friendly library that integrates the developed solvers and other features to facilitate the control of optimization tasks. Finally, in Chapter 5 we discuss the benchmarking results against the MaxCut problem and compare the results with other known results in the literature.

Chapter 1

Simulated Annealing Algorithm

In this chapter, we present a modified algorithm of Simulated Annealing (SA), along with a optimized and parallel C++ implementation. The proposed SA implementation was designed to enhance search capabilities and reduce convergence time to region of high-quality solutions. To provide context, we begin with a brief overview of the MaxCut problem, which will later serve as a benchmark for evaluating the performance of the Simulated Annealing algorithm examined in this chapter and the Scatter Search and Quantum Annealing algorithms discussed in subsequent chapters.

1.1 The MaxCut problem

Optimization problems are often effectively represented using graphs, where vertices (nodes) correspond to entities or decision variables, and edges capture the relationships or interactions between them. This graph-based representation provides a clear and intuitive framework for modeling complex systems, where the objective is to determine the best arrangement of elements according to specific criteria, such as minimizing or maximizing an objective function. In this section, we provide a brief overview of the MaxCut problem, a classical graph-based optimization problem that arises in various fields. We then present two mathematical formulations to solve the problem: the linear optimization form, which relies on a linear cost function, and the quadratic optimization form, which aligns with Quadratic Unconstrained Binary Optimization (QUBO).

1.1.1 Definitions

Given a graph $G = (V, E)$ where V is the set of vertices and E is the set of edges, each edge (i, j) has a weight $w_{ij} = w_{ji}$. The accumulated sum ($\mathcal{W}_G$) of all weights of the graph is given by

$$\mathcal{W}_G = \frac{1}{2} \sum_{j \in V} \sum_{i \in V} w_{ij}. \tag{1.1}$$

A valid cut of G is equivalent to finding a non-empty set $\mathcal{A}_0$ in V ($\mathcal{A}_0 \subset V$), such that a number of vertices of G resides in $\mathcal{A}_0$ and the rest resides in its complement

$\mathcal{A}_1 = V \setminus \mathcal{A}_0$, where $V = \mathcal{A}_0 \cup \mathcal{A}_1$. By partitioning the graph in this way, the edges can be grouped into three classes as illustrated in Fig. 1.1.

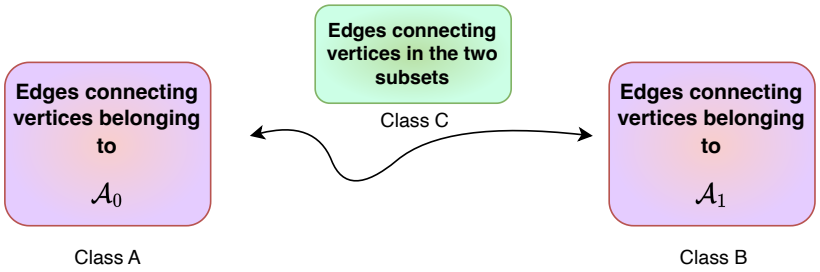


Figure 1.1: The three edge classes of a MaxCut problem.

The value of the cut of the bipartiton $\mathcal{A}_0 \cup \mathcal{A}_1$ is given by the sum of the weights w_{kl} of the edges in class C:

$$\text{Cut}(\mathcal{A}_0, \mathcal{A}_1) = \sum_{k \in \mathcal{A}_0, l \in \mathcal{A}_1} w_{kl}. \tag{1.2}$$

It turns out from Eq. (1.2) that solving a MaxCut problem is nothing but searching an optimal partitioning $V = \mathcal{A}_0 \cup \mathcal{A}_1$, such that $\text{Cut}(\mathcal{A}_0, \mathcal{A}_1)$ is maximized. It is important to note that different partitions may result in the same cut value.

1.1.2 Mathematical formulation

Usual translation

Given that, and linking to the fact that a solution for the MaxCut problem is nothing but finding two partitions that lead to the maximum cut. The membership of the vertices to a bipartition can, indeed, be represented by a binary value. To provide a proper mathematical formulation, we associate to each vertex in the graph a binary variable x_i, where:

$$x_i = \begin{cases} 0 & \text{if vertex } i \in \mathcal{A}_0 \\ 1 & \text{if vertex } i \in \mathcal{A}_1 \end{cases}. \tag{1.3}$$

For each cut edge $(i, j) \in E$, if $i \in \mathcal{A}_0$ and $j \in \mathcal{A}_1$, then

$$|x_i - x_j| = \begin{cases} 1 & \text{if } (i \in \mathcal{A}_0,\ j \in \mathcal{A}_1) \\ 0 & \text{else} \end{cases} \tag{1.4}$$

meaning if $i \in \mathcal{A}_0$ and $j \in \mathcal{A}_1$, the edge (i, j) contributes with $1 \times w_{ij}$ to the cut, otherwise it is not. To find the MaxCut of a given graph G, one should maximize the following linear cost function:

$$\text{MaxCut}(G) = \max \sum_{(i,j) \in E} w_{ij} |x_i - x_j|. \tag{1.5}$$

The expression $|x_i - x_j|$ ensures that the contribution to the cost is maximized when the vertices i and j are placed in different subsets. Once a valid bipartition is determined, the MaxCut value can be verified by summing over all the cut edges

$$\text{Cut}(\mathcal{A}_0, \mathcal{A}_1) = \sum_{i \in \mathcal{A}_0} \sum_{j \in \mathcal{A}_1} w_{ij} |x_i - x_j|. \tag{1.6}$$

In the following, we will discuss how this problem can be addressed by formulating the problem using QUBO representation.

QUBO translation

A general QUBO problem is defined by the following quadratic cost function

$$f(\mathbf{x}) = \sum_i Q_{ii} x_i + \sum_{i<j} Q_{ij} x_i x_j, \tag{1.7}$$

where the variable x_i takes binary values, Q_{ii} are the linear terms, and Q_{ij} are the quadratic terms. We can rewrite $f(\mathbf{x})$ as a product of an $n \times n$ upper triangular matrix and a binary vector of size n:

$$f(\mathbf{x}) = \mathbf{x}^T Q \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix}^T \begin{pmatrix} Q_{11} & Q_{12} & Q_{13} & \cdots & Q_{1n} \\ 0 & Q_{22} & Q_{23} & \cdots & Q_{2n} \\ 0 & 0 & Q_{33} & \cdots & Q_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & Q_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix}. \tag{1.8}$$

Solving a QUBO optimization problem requires finding a binary vector $\mathbf{x}$ that minimizes or maximizes the quantity:

$$c_{min} = \min_{\{\mathbf{x_i}\} \in \{0,1\}} \mathbf{x}^T Q \mathbf{x} \quad \text{or} \quad c_{max} = \max_{\{\mathbf{x_i}\} \in \{0,1\}} \mathbf{x}^T Q \mathbf{x}. \tag{1.9}$$

A quadratic formulation for maximum cut can be derived as

$$\text{Cut}(\mathcal{A}_0, \mathcal{A}_1) = \sum_{i \in \mathcal{A}_0} \sum_{j \in \mathcal{A}_1} w_{ij} (x_i + x_j - 2x_i x_j), \tag{1.10}$$

where we used the property $x_i^2 = x_i$. In summary, finding a maximum cut can be achieved by maximizing one of the following objective functions:

- Linear objective function:

$$\text{MaxCut}(G) = \max \sum_{(i,j) \in E} w_{ij} |x_i - x_j|. \tag{1.11}$$

- Quadratic objective function:

$$\text{MaxCut}(G) = \max \sum_{(i,j) \in E} w_{ij} (x_i + x_j - 2x_i x_j). \tag{1.12}$$

1.1.3 Graph instances

To evaluate the performance and robustness of our algorithms embedded in AlmoQUBO library, we employ a diverse collection of graph instances. These instances are sourced from the Gset dataset [16], a well-established standard in graph optimization research, particularly for the MaxCut problem. The Gset format provides a concise and structured representation of graphs. Each graph instance is named as `gxx`, where `xx` varies from 1 to 54 for graphs in `Set1`. A typical Gset instance (`gxx`) contains the following information:

- Header: the first line specifies the number of vertices ($|V|$) and edges ($|E|$) in the graph.
- Edge List: subsequent lines define the edges, with each line representing an edge and its associated weight (w).

The Gset dataset provides a rich collection of graphs with varying sizes, densities, and structural properties. More details about the specific graph instances used in our experiments are provided in Chapter 5.

With the foundational background established, we proceed to introduce Simulated Annealing (SA), which represents the first algorithm implemented for solving the MaxCut problem as well as other optimization problems reducible to the MaxCut formulation.

1.2 SA physical interpretation

Simulated annealing is a physics-inspired algorithm based on the analogy with the annealing process in metallurgy. In the field of condensed matter physics, annealing refers to the procedure whereby a physical system is subjected to thermal treatments encompassing two primary phases. The first phase involves heating the system to a maximum temperature, during which the particles start to move freely and become randomly clustered in some way. In the second phase, the system is gradually cooled down by reducing the temperature in accordance with a suitable cooling schedule. The cooling process causes the particles to move slowly until they arrange themselves into a more ordered configuration (referred to as the ground state of the system) with the smallest energy (see Fig. 1.2). The statistical description of the cooling phase can be derived by assuming that the system reaches thermal equilibrium at each temperature T, with the probability of the system being in a configuration of energy E' given by:

$$P(E = E') = Z(T)^{-1} \exp(-\beta E'), \tag{1.13}$$

where $\beta = (k_B T)^{-1}$ with k_B is the Boltzmann constant, and $Z(T)$ is the partition function (plays the role of a normalization factor). Physically, as the temperature of the system decreases, the Boltzmann distribution shifts towards configurations with low energies. When the temperature is very low, only those with the least amount of energy are likely to occur. To achieve stable configurations, the cooling schedule should

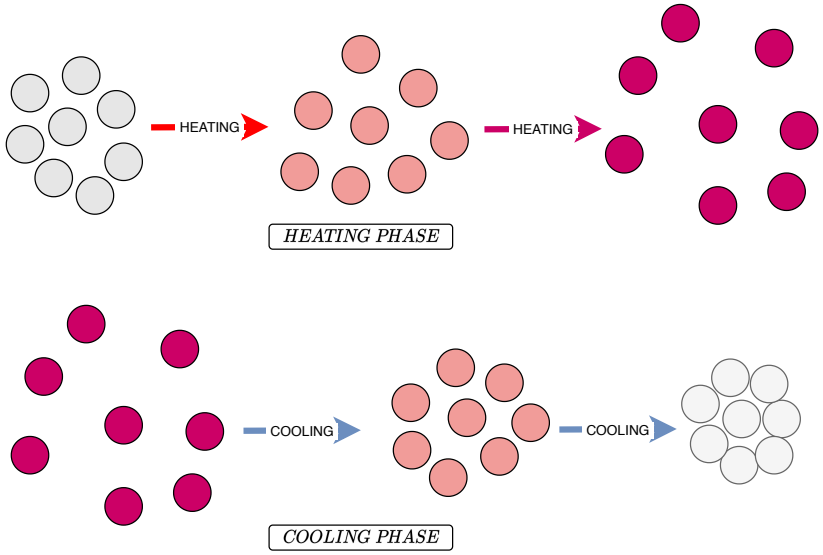


Figure 1.2: An illustration of the annealing process.

not be too fast. Slow cooling process allows the system to reach thermal equilibrium, minimizing quenching and the likelihood of defects being frozen into the system. While fast cooling, on the other hand, can result in a metastable amorphous structure rather than the desired low-energy crystal lattice structures.

To model the evolution of a physical system towards thermal equilibrium at a temperature T, the Monte Carlo method was introduced by Metropolis et *al.* (1953) [17]. Beginning with an initial state determined by the positions of all its particles, the method generates sequences of states by applying a slightly random change in position to a randomly selected particle. The energy difference between the current state and the new state is then calculated. If this energy difference is negative, the new state (lower energy state) becomes the current state. Conversely, if the energy difference is positive, the new state is accepted with a decreasing probability of

$$P_{accept} = \exp(-\beta \Delta E), \tag{1.14}$$

where $\Delta E = E_{new} - E_{current}$. This rule of acceptance is called the Metropolis criterion. By consistently implementing the same procedure, the system continues to evolve towards thermal equilibrium, and the probability distribution of the states gets closer to the Boltzmann distribution. Simulated annealing algorithm can be seen as an extension of the Metropolis algorithm evaluated at a series of decreasing temperatures.

1.3 Generic algorithm

The generic algorithm of Simulated Annealing [5] is structured as follows:

Algorithm 1 Simulated Annealing

Input: Initial solution s_0, initial temperature T_{max}, cooling rate α, stopping criterion T_{min}
Output: Best solution found s_{best}
$s \leftarrow s_0$
$s_{best} \leftarrow s$
$T \leftarrow T_{max}$
while $T > T_{min}$ **do**
    Generate a neighbor $s' \in \mathcal{N}(s)$ ▷ s' from neighborhood of s
    $\Delta E \leftarrow E(s') - E(s)$ ▷ Change in energy
    if $\Delta E < 0$ **then**
        $s \leftarrow s'$ ▷ Accept the new solution
    else
        $p_{threshold} \leftarrow$ random number in $[0, 1]$
        $p_{accept} \leftarrow \exp\left(-\frac{\Delta E}{T}\right)$
        if $p_{threshold} < p_{accept}$ **then**
            $s \leftarrow s'$ ▷ Accept the new solution with a certain probability
        end if
    end if
    if $E(s) < E(s_{best})$ **then**
        $s_{best} \leftarrow s$ ▷ Update the best solution found
    end if
    $T \leftarrow \alpha T$ ▷ Cool down the temperature
end while
return s_{best}

A perturbing mechanism similar to that by Metropolis algorithm [18] is used to generate new solutions from the neighborhood of a current solution. The process takes two steps and involves performing iterative transitions by replacing a current solution with a newly discovered solution. During the step, it generates a new solution from the neighborhood of a current solution, and then calculates the difference between their objective functions and accept the new solution with a probability of acceptance.

1.4 Optimized algorithm

As discussed earlier, traditional Simulated Annealing uses a cooling schedule to gradually reduce the temperature, lowering the probability of accepting worse solutions. However, when solving difficult problems like the MaxCut problem, the cooling schedule can cause the algorithm to not reach other regions of the search. In this section, we present an enhanced heating-based variant of Simulated Annealing that focuses more on exploration of the search space using a perturbed heating schedule. The fundamental principle behind this approach is to facilitate the initial refinement of promising solutions (through early fine-tuning of local solutions) while gradually extending the search and enhancing the capacity to bypass local encountered minima. Our proposed algorithm of the optimized SA is presented in Alg. 3.

Algorithm 3 Optimized SA algorithm.

Input: δ_{vec}, n
initialize $E_{current} \leftarrow 0$, $E_{best} \leftarrow 0$, $T_{current} \leftarrow 0$, s_0, s_f, and other parameters
for $step \leftarrow 0$ **to** $maxSteps$ **do**
    $\Delta E \leftarrow T_{current} \cdot \frac{\delta_{vec}[k]}{E_{best}+1}$
    $p_{accept} \leftarrow \text{FastExp}(\Delta E)$
    $p_{threshold} \leftarrow$ random value between $[0, 1]$
    if $p_{accept} > p_{threshold}$ **then**
        $E_{current} \leftarrow E_{current} + \delta_{vec}[k]$
        $s_0[k] \leftarrow 1 - s_0[k]$
        $\delta_{vec}[k] \leftarrow -\delta_{vec}[k]$
        for each $e \in \text{edges}[k]$ **do**
            $v_{idx} \leftarrow e.\text{first}$
            $w \leftarrow e.\text{second}$
            $\delta_{vec}[v_{idx}] \leftarrow \delta_{vec}[v_{idx}] + 2w \cdot (1 - 2 \cdot (s_0[k] \neq s_0[v_{idx}]))$
        end for
        if $E_{current} > E_{best}$ **then**
            $E_{best} \leftarrow E_{current}$
            $s_f \leftarrow s_0$
        end if
    end if
    $k \leftarrow k + 1$
    if $k == n$ **then**
        $k \leftarrow 0$
    end if
    $T_{current} \leftarrow T_{current} + \alpha$
    if $E_{best} == E_{current}$ **then**
        $c_{nimprv} \leftarrow c_{nimprv} + 1$
        perturbation 1
        $step \leftarrow 0$
    else
        $c_{nimprv} \leftarrow 0$
    end if
    perturbation 2
    if $c_{nimprv} == limit$ **or** $T_{current} \geq T_{max}$ **then**
        break
    end if
end for
return $\{E_{best}, s_f\}$

The input δ_{vec} is a vector of size n, where each entry $\delta_{vec}[k]$ initially stores the sum of the weights of the edges connected to vertex k. When starting SA, we use the vector $\delta_{vec}[k]$ to keep tracking the impact of flipping a vertex on the objective function:

$$\delta_{vec}[v_{idx}] \leftarrow \delta_{vec}[v_{idx}] + 2w\,(1 - 2(s_0[k] \neq s_0[v_{idx}]))\,. \tag{1.15}$$

For each neighboring vertex v_{idx} connected by an edge with weight w, the value of $\delta_{vec}[v_{idx}]$ is updated based on whether vertex k and vertex v_{idx} belong to the same or different partitions. If vertex k and vertex v_{idx} are in the same partition, the objective is negatively impacted, and the change is updated by a factor of $-2w$. If they are in different partitions, the objective is positively impacted, and the change is updated by a factor of $+2w$. When we flip a vertex, we essentially change the partition of its neighboring vertices as well. To accurately quantify this change, we multiply the weight of each edge connected to the flipped vertex by 2. This technique allows to evaluate the objective value incrementally to avoid the overhead of recomputing the objective value for middle solution at every iteration. The decision variables of the input vector s_0 are all initialized to 1 to ensure compatibility with the null input initial objective value:

$$s_0 = \{x_0 = 1,\ x_1 = 1,\ x_3 = 1,\ \ldots,\ x_{n-1} = 1\}. \tag{1.16}$$

The parameter c_{nimprv} counts the number of non-improvements. If c_{nimprv} hits the defined limit, the algorithm breaks the loop and returns the last best solution.

1.4.1 Heating schedule

For the proposed SA variant, the temperature T is set to start from zero to allow an early exploration of the solution space. Then after each step, the temperature is incrementally increased following a well defined schedule (we refer to the heating schedule by α):

$$T \leftarrow T + \alpha. \tag{1.17}$$

It is noteworthy to mention that SA is highly sensitive to the selection of the heating schedule, as a small heating step may result in a significant slowdown for the algorithm. If the heating step is high, the algorithm will be faster, but it will eventually reach a poor solution. To balance the time-solution and increase the chance of finding a global optimum, the heating schedule is carefully determined based on several runs of the algorithm. After conducting multiple tests pertaining to different problem sizes, we found that the optimal heating step can be set to

$$\alpha = 0.00001 = 10^{-5}. \tag{1.18}$$

The algorithm begins from $T_{current} \leftarrow 0$ and keeps increasing $T_{current}$ by α, as in Eq. (1.17). The algorithm terminates when $T_{current}$ reaches T_{max} or a non-improvement counter constraint reaches its limit. We also emphasize that the choice of T_{max} depends on the complexity of the problem to be solved.

1.4.2 Parameter of acceptance

The parameter of acceptance (p_{accept}) plays a critical role in balancing the exploration of search space with the convergence toward a viable solution. p_{accept} depends on the objective value difference (ΔE) calculated based on the current temperature ($T_{current}$), the change in objective value (δ) and the best objective value (E_{best}) reached so far:

$$\Delta E = T_{current} \times \frac{\delta}{E_{best} + 1}. \tag{1.19}$$

p_{accept} is computed as

$$p_{accept} = \exp(\Delta E). \tag{1.20}$$

The importance of p_{accept} lies in the fact that it controls whether worse solutions can be accepted or not. From Eq. (1.20) and Eq. (1.19), it can be seen that ΔE will increase as $T_{current}$ increases, leading to higher chance of acceptance.

1.4.3 Code optimization

To evaluate the parameter of acceptance, SA uses the formula in Eq. (1.20) which computationally demanding, particularly when it is evaluated repeatedly across numerous iterations. Since p_{accept} needs to be computed at each step, the effect of exponential evaluation time becomes noticeable. Also, as Simulated Annealing typically requires many iterations to explore the solution space, the cumulative cost of these evaluations can become a significant bottleneck. To avoid this, we optimized the exponential calculation by using techniques such as approximating the exponential function through *polynomial expansions* (Taylor series) or utilizing hardware-specific optimizations like *intrinsics* provided by Intel. These techniques permitted reducing the overhead arising from computing p_{accept}. In our C++ implementation, we made it possible to use these techniques depending on the compiler used.

```
1       #ifdef __USE_INTEL__
2           DELTAE = temp * change[k] / (best_obj + 1.00);
3           __m512d DELTAE_VEC = _mm512_set1_pd(DELTAE);
4           __m512d exp_E = _mm512_exp_pd(DELTAE_VEC);
5           p_accept = exp_E[0];
6       #else
7           DELTAE = temp * change[k] / (best_obj + 1.00);
8           p_accept = FastTayExp(DELTAE, 15);
9       #endif
```

In the provided snippet, the use of *SIMD* instructions, supported by Intel compilers, speeds up the evaluation of p_{accept} by vectorizing the computations across multiple data points, thereby enhancing efficiency. Alternatively, when *SIMD* is not available, the function *FastTayExp* approximates $\exp(\Delta E)$ using a truncated Taylor expansion to strike a balance between accuracy and computational speed. These optimizations have improved the performance of the SA algorithm by enabling faster evaluations of p_{accept}, which, in turn, allows SA to scale to larger problem instances and to run

more iterations in a reasonable time frame. Without such optimizations, the cost of evaluating p_{accept} would dominate the computational complexity of the algorithm, thereby limiting its practical application for large-scale optimization problems.

1.5 Randomness

As the case for all randomized algorithms. We have observed that Simulated Annealing sometimes achieves better results by perturbing the random number generator:

```
p_threshold = drand48() * 0.999.
```

This could be explained by the fact that thermal fluctuations in physical systems are highly random. To mimics thermal fluctuations, high degree of randomness is required to favor skipping suboptimal configurations. This observation motivated the experimentation of the other random number generators such as the standard `C++ random`, `TRNG4`, and `MKL`. Furthermore, it was also observed that different seeds can produce diverse probabilistic distributions, which can lead to varying outcomes across independent runs. This strongly suggests that we could benefit from utilizing parallelization to achieve superior outcomes without the need for manually altering the seed. The algorithm can explore multiple probabilistic distributions simultaneously by assigning each worker a different seed, enhancing the variety of explored solutions and enhancing the likelihood of locating optimal or near-optimal solutions.

1.6 Parallel algorithm

In this section, we present an embarrassingly parallel CPU implementation of the Simulated Annealing algorithm based on the optimized version. In a typical implementation, a master worker retrieves the problem data and broadcast it to other workers, and then collects the results at the end of the execution. This parallel version employs the methodology of requiring each CPU to execute the optimized SA version by employing a distinct probability distribution. For this purpose, we used a function based on the C++ STD hasher to generate a seed for each worker:

```
int generate_seed(int rank)
{
    std::hash<int> hasher;
    std::mt19937 gen(hasher(rank));
    std::uniform_int_distribution<int> seed(0, rank + 1);

    return hasher(rank + (8 - rank)*seed(gen));
}
```

Regarding $p_{threshold}$ generation, we experimentally found that the random number generator provided by **Intel MKL** is performing better than the standard C++ ones

as well the ones provided by in the **TRNG4** library. The scheme of the code can be understood by the following scheme

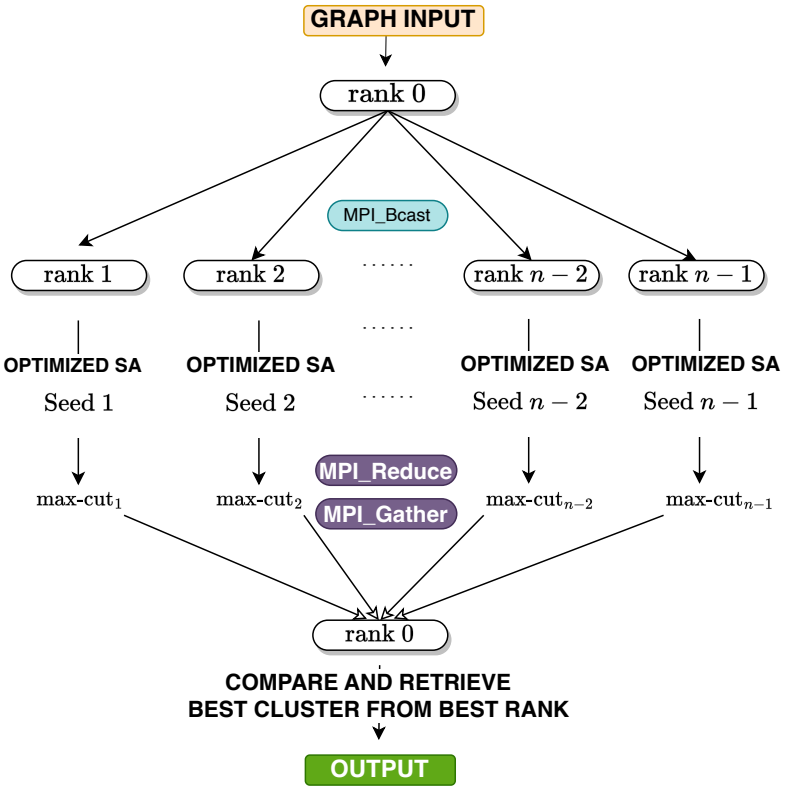


Figure 1.3: A scheme of the proposed parallel Simulated Annealing.

As can be seen from the above scheme, all CPUs run the Simulated Annealing solver with a different seed generated by the function `generate_seed`. This ensures that each CPU follows different exploration paths from the same starting point. At the end of the execution, the solver returns the results as a tuple:

```
results = SimulatedAnealing(rank, n, change, cluster, edges);
```

The tuple contains the best objective value found by the worker (`bObjRank`), the temperature at which this value was found (`rTemp`), and the configuration that yielded the objective value (`bConfigRank`). The returned values are unpacked as follows:

```
    double rTemp, reachedTemp_best;
    bObjRank = std::get<0>(results);
    rTemp = std::get<1>(results);
    bConfigRank = std::get<2>(results);
```

In the next step, the master worker collects all the objective values found by the other workers using an `MPI_Gather`:

```
MPI_Gather(&bObjRank, 1, MPI_DOUBLE, allBObjs.data(), 1, MPI_DOUBLE, 0, MPI_comm);
```

The master worker stores the objective values in the same order of the workers using their ranks. To determine best rank, the master worker compares the objective values in `allBObjs` on-to-one and finds the best index of the best one. In the final phase, once the rank holding the best result (`bFank`) is identified, a conditional check is performed. If the rank of the current process matches `bRank`, it sends its best configuration (`bConfigRank`) and the corresponding temperature (`rTemp`) to the master process using `MPI_Send`.

```
1  if (rank == bRank)
2  {
3      MPI_Send(bConfigRank.data(), n_vertices, MPI_INT, 0, 0, MPI_comm);
4      MPI_Send(&rTemp, 1, MPI_DOUBLE, 0, 0, MPI_comm);
5  }
6
7  if (rank == 0)
8  {
9      MPI_Recv(bConfig.data(), n, MPI_INT, brank, 0, MPI_comm, MPI_status);
10     MPI_Recv(&BTemp, 1, MPI_DOUBLE, bRank, 0, MPI_comm, MPI_status);
11 }
```

Concurrently, the master worker (rank 0) waits to receive this data using `MPI_Recv`. It receives the best configuration into `bConfig` and the temperature into `rTemp_best`, ensuring it collects the optimal results found during the parallel computation. Finally, the master worker handles the rest of the tasks, such as printing the solution on the screen or saving it in a file.

Chapter 2

Scatter Search Algorithm

Scatter Search (SS) is an advanced metaheuristic optimization method, originally developed by Glover in 1977 [10]. Overtime, it gained significant attention for its efficiency in solving certain complex optimization problems. The method combines elements of evolutionary algorithms, such as population-based search, with strategic solution recombination, allowing it to explore large solution spaces effectively and avoid local optima. In this chapter, we will discuss our GPU-accelerated Scatter Search implementation.

2.1 Algorithm

The basic Scatter Search algorithm structure [2] is defined as follows:

Algorithm 4 Scatter Search Algorithm

1: **Generate** an initial set of trial solutions S, ensuring that all solutions are distinct:

$$s_l \neq s_m, \forall s_l, s_m \in S \text{ where } l \neq m.$$

2: **Apply** a local refinement method $\mathcal{L}_R$ to each solution $s_i \in S$, producing refined solutions s_k:

$$s_k \leftarrow \mathcal{L}_R(s_i) \quad \text{such that} \quad \text{cost}(s_k) \quad \text{better than} \quad \text{cost}(s_i).$$

3: **Create** a reference set R_S from the refined solutions, selecting based on quality and diversity.
4: **while (stopping condition not met) do**
5: **Generate** subsets of solution pairs from R_S for combination.
6: **Combine** the selected pairs to produce new solutions.
7: **Apply** the local refinement method $\mathcal{L}_R$ to improve the combined solutions.
8: **Update** the reference set R_S with improved and diverse solutions.
9: **Sort** R_S by quality and diversity criteria.
10: **end while**
11: **return** the best solution: $R_S[0]$.

A graphical scheme of SS can be presented as follows:

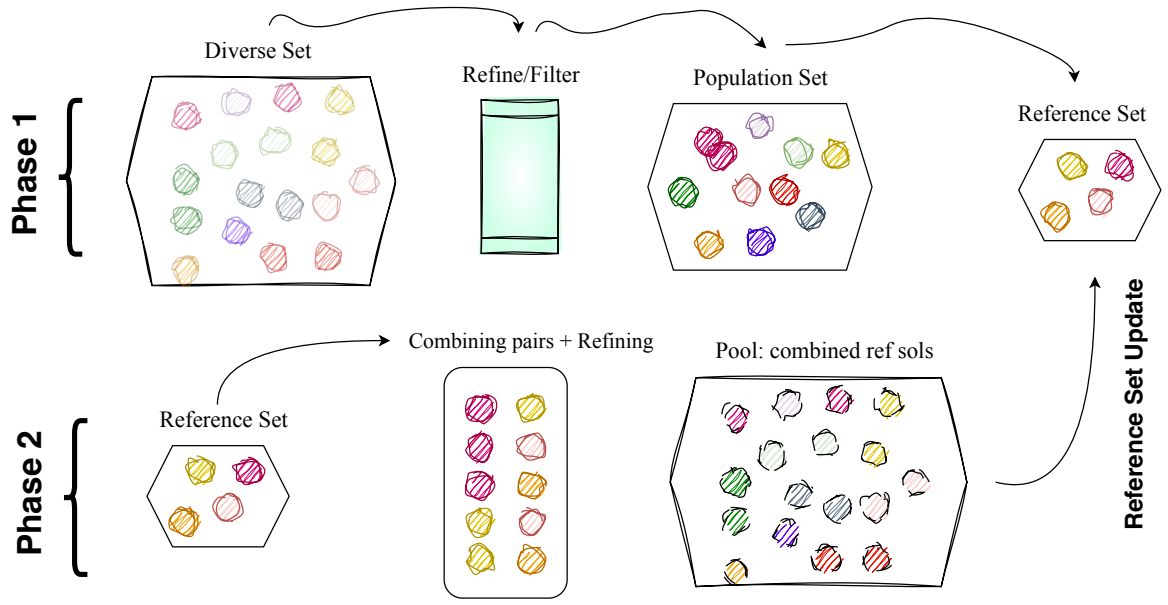


Figure 2.1: Schematic view of Scatter Search steps.

The process continues to operate continuously until a stopping condition is reached. Subsequently, the optimal solution is obtained by sorting the reference set. Indeed, advanced Scatter Search implementations frequently surpass the conventional steps outlined in Alg. 4.

2.2 Our SS implementation

It can be challenging to develop an efficient Scatter Search implementation that consistently yields the desired outcome, especially when adjusting it to address specific optimization problems. We would like to thank Marti et al. [2] for sharing their implementation, which helped to provide significant insights and inspiration and allowed a faster development of our implementation. From their work, we were inspired to improve the SS implementation by adding more methods and improvements. In our CUDA implementation, the general structure of Scatter Search solver is given in the following code snippet:

```
1     /* SCATTER SEARCH */
2     void ScatterSearch(SS& pb)
3     {
4         /* BUILD POP AND REF SET */
5         SSCreate_P(pb);
6         SSCreate_RefSet(pb);
7
8         /* MAIN LOOP */
9         for(int step = 0; step < pb.MaxSteps; step++)
10        {
```

```
                if(foundNewSol)
                {
                    SSUpdate_RefSet(pb);
                }
                else
                {
                    SSRebuild_RefSet(pb);
                }
        }
    };
```

2.2.1 POP/REF sets

The `ScatterSearch()` solver operates on the back-end class `SS`, which is responsible for managing all key aspects of the algorithm, from reading and storing problem data, initialization of parameters to managing population (POP), reference (REF), and pool set. The function `SSCreate_P()` is responsible for generating a population of solutions and refining them. Based on the choice parameter `opConst`, this method calls one of our GPU implemented construction methods as shown bellow:

```
    switch (pb.opConst)
    {
        case 0:
            RANDHam_GPU(pb,  pb.numSols, pb.parConst);
            break;
        case 1:
            SSRand_GPU(pb, pb.numSols, pb.parConst);
            break;
        case 2:
            SSGRASP_GPU(pb, pb.numSols, pb.parConst);
            break;
        default:
            std::cerr << "ERROR: opConst not valid" << std::endl;
    }
```

These methods are specifically designed to produce a high-quality initial set of solutions, ensuring a robust starting point for further exploration and optimization. Following the generation, a GPU (`BESTLS_GPU`) and a recursive CPU (`SSLS_ECB`) local refinement method are employed to further refine the solutions. The best **PSize** solutions are then added to the population. Interestingly, it has been observed that our `SSCreate_P()` method excels in uncovering superior solutions during the generation of the population set, achieving best-known outcomes for certain (MaxCut) problem instances. The function `SSCreate_RefSet()` is a CPU method used to filter solutions in the population

set based on quality and diversity. The filtered solutions are added to the reference set, resulting in a proportion of $b/2$ of high-quality solutions, while the remaining $b/2$ are diverse solutions.

2.2.2 Update/Rebuild REF set

The funtion `SSUpdate_RefSet()` calls highly optimized CPU methods for combining the reference set solutions to produce newer solutions. The combined solutions are refined again and the worst solutions are replaced by the new discovered highly quality ones. The combination of solution is performed using one of the following functions:

- `SSCombineB`: Generates new solutions by blending two parent solutions, `sol1` and `sol2`, based on their objective values (`ObjVal`). The goal is to produce offspring solutions that inherit characteristics from both parents, guided by a probabilistic scoring mechanism.
- `SSCombine2B`: Generates two new solutions by combining `sol1` and `sol2`. This method uses a combination of deterministic and probabilistic techniques, relying on mixing operations to generate new solutions.
- `SSCombinePR2B`: Generates two new solutions by employing path relinking strategy between `sol1` and `sol2`. This function combines two solutions to produce two new solutions.

These methods as well as their mathematical aspects are described in detail by Marti et al [18]. When no solutions are being admitted, the method `SSRebuild_RefSet()` is used to rebuild the reference set by removing less promising solutions and replacing them with higher-quality or more diverse solutions.

2.3 CPU/GPU methods

Our SS implementation overlaps CPU and GPU computation for solving the MaxCut optimization problem. The expensive calls are parallelized on the GPU to minimize the computational burden. Moreover, non-trivial data handling were employed to reduce computational complexity while producing similar results in a shorter time. To hit better result than those reported in the literature [2], we introduced novel generation and refinement methods that benefit from GPU capabilities.

2.3.1 Generation

We introduced two new GPU generation kernels:

- `SSRand_GPU`: This GPU method uses bias to stimulate the random sampling of initial solutions.

```
1       for (int i = 0; i < n_vertices; i++)
2       {
```

```
3           rand_p = curand_uniform(&state);
4           int bitctr = (rand_p < bias);
5           d_cluster[i] =  1 - bitctr;
6       }
```

This trick allows threads to explore the solution space more by influencing the random bit generator (control randomness). This exploration technique was shown to significantly enhance the efficiency of the search process.

- RANDHam_GPU: This GPU method uses a bias in addition to a minimum Hamming distance constraint that enforces diversity within the generated solutions (the solutions are distant enough from each other). When a solution does not respect the constraint, the kernel attempts to replace it with a valid solution up to a maximum of 100,000 times, or until a valid solution is discovered. If validation fails repeatedly, it will increment the required hamming distance during retries to adapt the diversity requirement.

The GPU kernels adhere to the strategy depicted in Fig. 2.2:

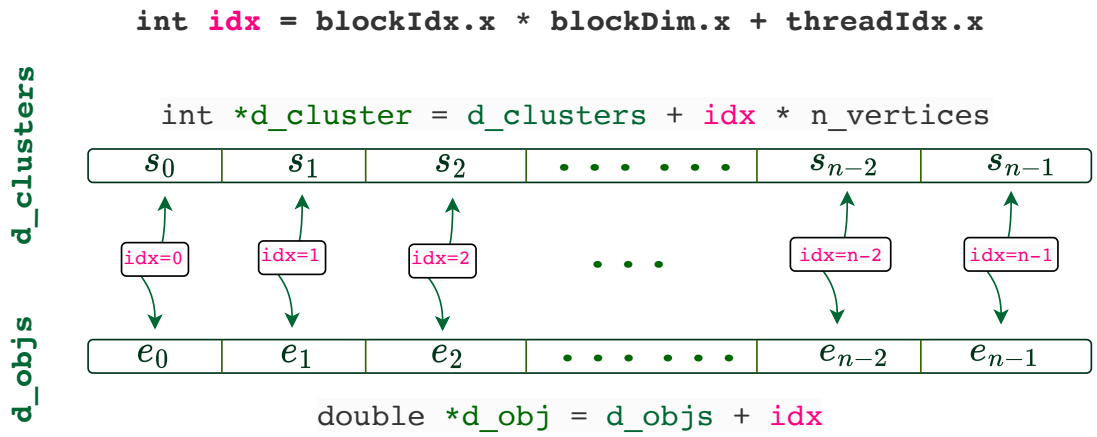


Figure 2.2: CUDA parallelization scheme.

At generation stage, two contiguous blocks of memory are allocated on the GPU, namely `d_clusters` and `d_objs`. `d_clusters` is a single large array of size `n` × `sizeof(d_cluster)`, containing all non-initialized clusters, where each cluster is accessed by offsetting (moving the pointer) with `idx` × `sizeof(d_cluster)`. `d_objs` is a single array of objective values, where each objective is accessed by moving the pointer by `idx`. The proposed flattened memory strategy facilitates coalesced memory access, allowing each CUDA thread to process a single cluster and its corresponding objective. This ensures that threads within the same warp can access contiguous memory locations while avoiding race condition. Additionally, it optimizes memory transfer between the CPU and GPU, resulting in a reduction in memory transactions and consequently leading to enhanced performance.

2.3.2 Refinement

In this work, we invented two new local search algorithms that uses the parallel capability of GPUs:

- Algorithm 1: Iterative refinement with bit-flip tolerance threshold.

Algorithm 5 Generalized parallel local refinement algorithm

Input: `Clusters, Costs, Edges, Offsets, NSols, minDelta, maxDepth, csize`
Parallel execution for each solution:
for $idx = 0$ **to** $NSols - 1$ **do**
    $cluster \leftarrow Clusters[idx]$
    $obj \leftarrow Costs[idx]$
    for $iter = 1$ **to** $maxDepth$ **do**
        for $v = 0$ **to** $csize - 1$ **do**
            Flip Vertex State:
            $old\_state \leftarrow cluster[v]$
            $cluster[v] \leftarrow 1 - old\_state$
            Compute Delta:
            $delta \leftarrow 0$
            for each edge e **connected to** v **do**
                Retrieve edge weight w and neighbor state
                Compute contribution to $delta$
            end for
            Decision to Revert Flip:
            if $delta < minDelta$ **then**
                $cluster[v] \leftarrow old\_state$
            end if
        end for
    end for
    Update cost:
    $obj \leftarrow$ ObjectiveFunction($cluster$)
end for

- Algorithm 2: Randomized iterative refinement with bit-flip tolerance threshold.

Algorithm 6 Generalized parallel randomized local refinement algorithm

Input: `Clusters, Costs, Edges, Offsets, NSols, minDelta, maxDepth, csize`
Parallel execution for each solution:
for $idx = 0$ **to** $NSols - 1$ **do**
    $cluster \leftarrow Clusters[idx]$
    $obj \leftarrow Costs[idx]$
    Setup RNG state for idx
    for $iter = 1$ **to** $maxDepth$ **do**
        Randomly select vertex index:
        if $vertex\_idx \geq csize$ **then**
            $vertex\_idx \leftarrow RNG()\%csize$
        end if
        Flip Vertex State:
        $old\_state \leftarrow cluster[vertex\_idx]$
        $cluster[vertex\_idx] \leftarrow 1 - old\_state$
        Compute Delta:
        $delta \leftarrow 0$
        for each edge e **connected to** $vertex\_idx$ **do**
            Retrieve edge weight w and neighbor state
            Compute contribution to $delta$
        end for
        Decision to Revert Flip:
        if $delta < minDelta$ **then**
            $cluster[vertex\_idx] \leftarrow old\_state$
        end if
    end for
    Update cost:
    $obj \leftarrow$ ObjectiveFunction($cluster$)
end for

Both algorithms perform iterative refinement, with a fixed number of iterations defined by `maxDepth`. In each iteration, the first algorithm traverses all vertices in the solution, performs a bit-flip, and calculates the change in the objective function, referred to as `delta`, by evaluating the contributions of all edges connected to the flipped vertex. The second algorithm introduces randomization in vertex selection. It randomly selects a vertex, flips its state, and then computes `delta`. The decision to retain or revert the flip is based on a threshold condition (`minDelta`). The tolerance for accepting or discarding a bit-flip prevents the algorithms from becoming trapped and enables them to explore diverse search paths.

In the following, we report the performance of these two algorithms for some Gset instances [16]:

Gset	Algorithm	Max	Min	Avg.	Kernel time (ms)
g1	SSRand_GPU	9791	9312	9585.67	N/A
	LS ALG 1	11613	11468	11541.2	477.54
	LS ALG 2	11603	11433	11513	457.43
g2	SSRand_GPU	9782	9332	9585.37	N/A
	LS ALG 1	11606	11469	11544.6	461.57
	LS ALG 2	11602	11431	11517.3	435.19
g12	SSRand_GPU	60	-62	-2.026	N/A
	LS ALG 1	554	532	543.42	43.74
	LS ALG 2	556	524	540.6	55.95
g22	SSRand_GPU	10292	9758	9997.92	N/A
	LS ALG 1	13257	13097	13171.2	531.55
	LS ALG 2	13254	13014	13117.3	570.34

Table 2.1: Performance comparison of the two algorithms.

The above data is collected by generating 1000 trials, for each instance, using our `SSRand_GPU` method discussed earlier. It is evident that these algorithms are highly effective in improving upon the initially generated solutions. In certain instances, such as `g12`, the second algorithm was able to reach the best known cost following only the initial refinement. In fact, both algorithms excel at exploring a broader solution space at lower computational costs, making them efficient for optimization tasks. It is worth mentioning that our CUDA implementation of these algorithms follows the same strategy described in Fig. 2.2.

On the CPU, we employ a recursive local refinement method based on the SS implementation provided by Marti et al. [2]. This method adds another layer of refinement, which has been refactored and further optimized by using a different approach to data access. In the following sections, we will discuss the tricks used to optimize both the CPU and GPU code.

2.3.3 CPU / GPU optimization

Data access

A new method based on edge list adjacency and offsets between edges is used to reduce the time complexity of loops and the computation of relevant quantities. The method involves representing the graph with an edge list and offset array:

- `edges[]`: A list of tuples containing edges and their corresponding weights.
- `offset[]`: An array containing the starting and ending indexes of each edge.

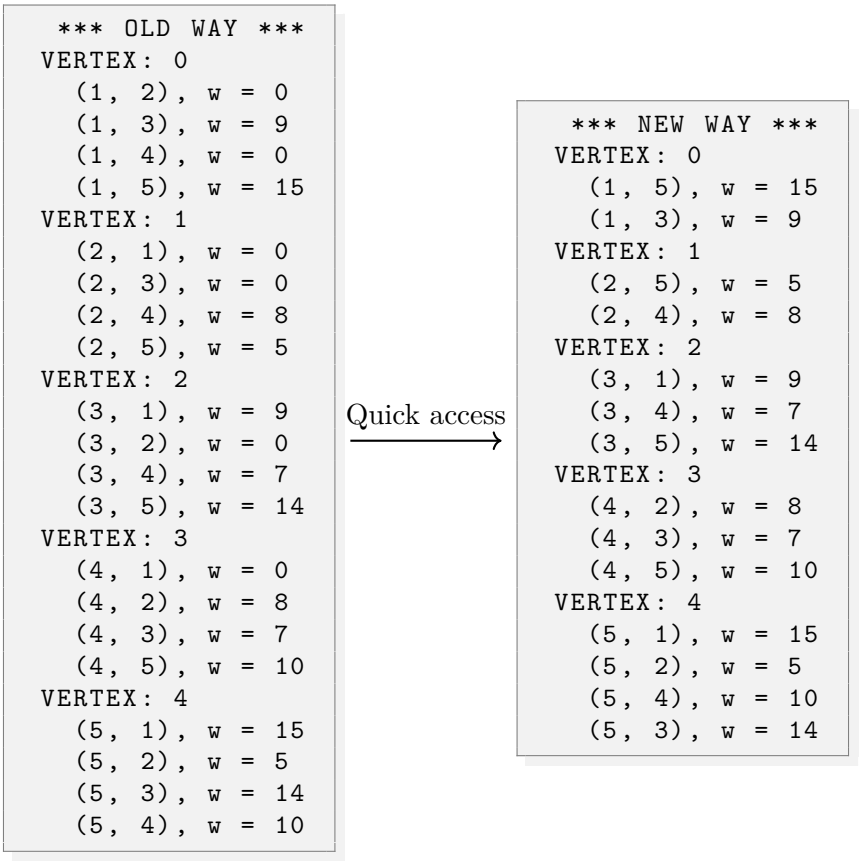
Using this technique, we were able to drop time complexity while producing the same results:

$$O(N^2) \longrightarrow 2 \times O(E), \tag{2.1}$$

where E is the number of edges. We tested the implementation with a bigger graph, like `g55` ($N = 5000$ and $E = 12498$). With compiler optimizations, the time required for evaluating the cost function of a randomly generated trial was reduced from:

$$t_{O(N^2)} = 1.7458 \times 10^{-2} \longrightarrow t_{2\times O(E)} = 8.9590 \times 10^{-5}, \tag{2.2}$$

which is ≈ 194.86 times faster, and ≈ 968 times faster without compiler optimizations. In the following example, we demonstrate how the edge list and offset strategy not only enables faster data access but also significantly reduces memory usage, particularly when dealing with sparse graphs.



The edgelist and offset of this example are:

$$\texttt{edgeList} = [\{1,\ 5,\ 15\},\ \{1,\ 3,\ 9\},\ \cdots\ ,\ \{5,\ 4,\ 10\},\ \{5,\ 3,\ 14\}]. \tag{2.3}$$

$$\texttt{offset} = [0,\ 2,\ 4,\ 7,\ 10,\ 14]. \tag{2.4}$$

This highly optimized data access pattern allows for fast computing by minimizing memory transactions, avoiding unnecessary computations on zero-weighted edges, and focuses only on the non-zero weighted edges.

CPU/GPU code

The CPU and GPU code is written in a highly optimized way to minimize branching (reduce cache-misses) and warp divergence (GPU-related), which are of critical importance for maximizing CPU/GPU performance. Warp divergence happens when the threads in the same warp (32 threads per warp) follow different execution paths, such as when an `if-else` branch is encountered. This causes the GPU to serialize the execution, leading to reduction in parallel efficiency and performance degradation. To avoid this issue, the code was written using several techniques to avoid critical branches as much as possible. For instance, the following code snippet:

```
1 if (vec[i] != vec[j]) { delta += edge.w; } else { delta -= edge.w; }
```

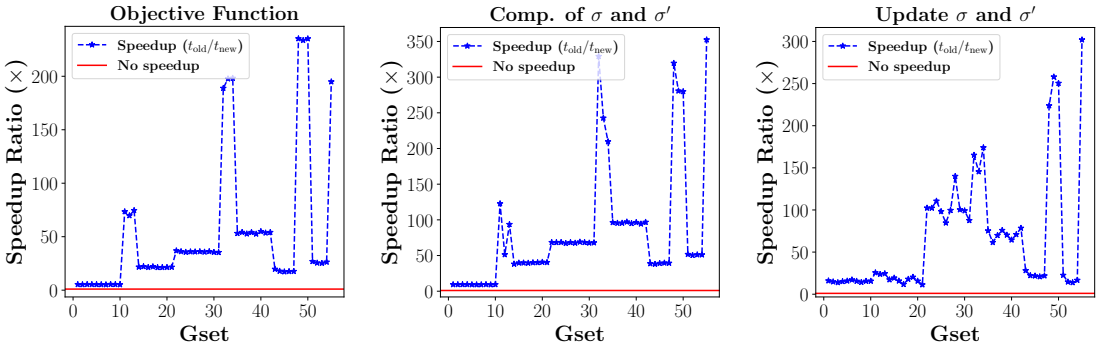
introduces a branch based on the condition (`vec[i] != vec[j]`). If threads in the same warp evaluate this condition differently, some threads will follow the `if` path while others will follow the `else` path, causing warp divergence. Since, in our case, the vectors contains either 0 or 1. We can avoid this, by rewriting the snippet as

```
1 int ctrl = vec[i] - vec[j];
2 delta += (2 * ctrl * ctrl - 1) * edge.w;
```

This trick entirely removes branching by using arithmetic operations. The quantity `2 * ctrl * ctrl - 1` evaluates to 1 when `vec[i] != vec[j]` and -1 otherwise, reproducing the same logic as the original `if-else` statement. This ensures that the GPU executes the entire warp without divergence, leading to more efficient utilization of GPU resources. Additionally, the usage of edge list techniques reduces the dependence on critical branches, both on the device and host. As Scatter Search is CPU/GPU hybrid, we used CUDA streams to overlap CPU and GPU computations and to optimize memory transfer by using pinned memory on the host.

2.3.4 CPU speedups

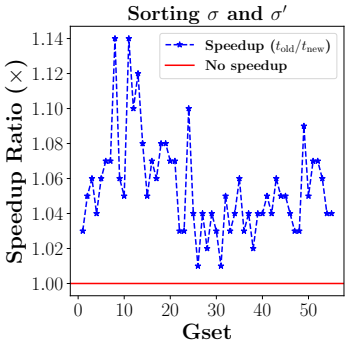
The optimizations we implemented resulted in significant accelerations for several core CPU functions. The following list of figures highlights and reaffirms the achieved speedups for these functions:



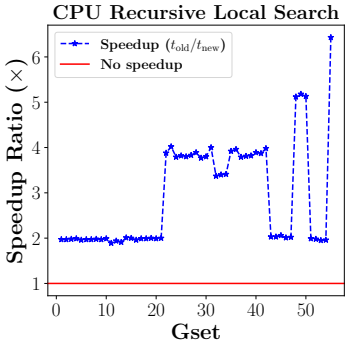
(a)

(b)

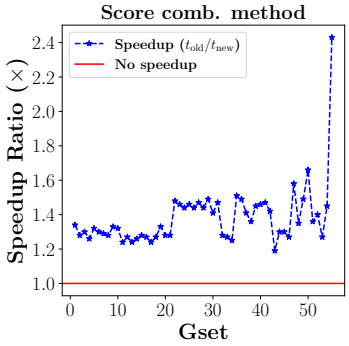
(c)



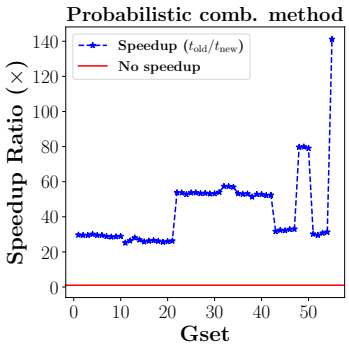
(d)



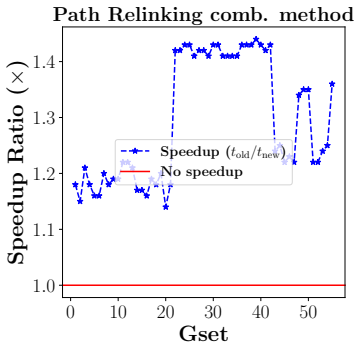
(e)



(f)



(g)



(h)

Here, t_{old} refers to the reference time (taken by previous implementations by Martí et *al.* [2]), and t_{new} is the time taken by our implementations to achieve better or similar results. σ and σ' are two greedy functions computing the contribution of each vertex to the objective function [2].

Chapter 3

Quantum Annealing

In this chapter, we will discuss quantum annealing in the D-Wave machine and briefly highlight the underlying quantum physics behind it. Quantum Annealing (QA) is a heuristic optimization technique [19] that uses the principles of quantum mechanics to solve hard combinatorial problems. Similar to its classical counterpart, Simulated Annealing, QA aims at finding the global optimum of a defined objective function describing a problem. However, instead of relying on thermal fluctuations to escape local optimum, quantum annealing uses quantum tunneling [20] to traverse energy barriers that could trap classical methods.

3.1 D-Wave quantum annealer

Quantum annealing is heuristic quantum algorithm inspired by classical Simulated Annealing [5]. QA was introduced to find the ground state of a hardly solvable Hamiltonian. In contrast to Simulated Annealing (SA), QA make use of quantum fluctuations (quantum tunneling) to bypass local minima as depicted in Fig. 3.1.

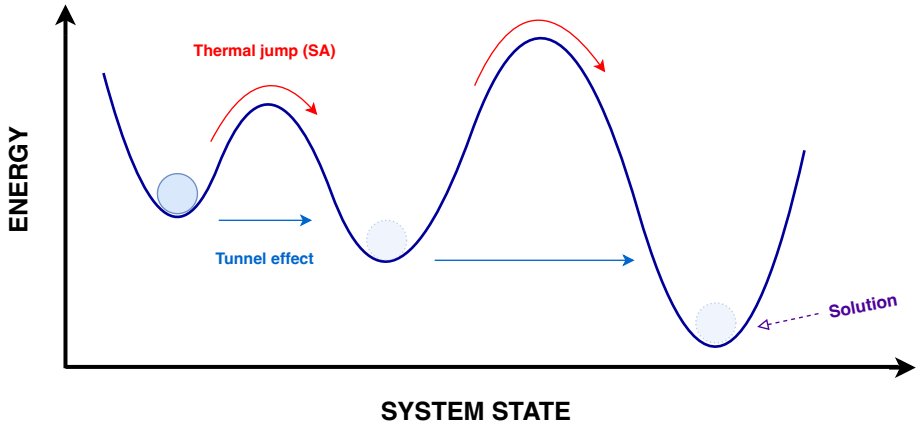


Figure 3.1: Illustrative scheme of classical and paths of SA and QA.

Quantum tunneling, also known as the tunnel effect, is a quantum mechanical phenomenon where a particle has a finite probability of passing through a potential

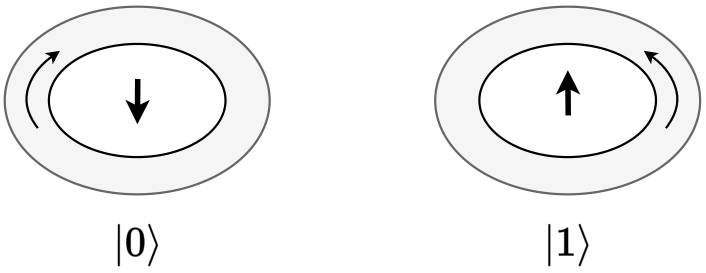


Figure 3.2: Qubit representation using a circulating current.

energy barrier, even when its energy is less than the height of the barrier. This occurs because particles at the quantum scale are described by wavefunctions, which represent the probability of their presence at different positions. These wavefunctions extend beyond the boundaries of the barrier, allowing a non-zero probability of the particle being found on the other side, despite the barrier being insurmountable in classical physics [21]. Recently, many studies arguing whether QA outperforms SA in time-to-solution, many of them confirmed the QA speedup over SA [22, 23] while few studies showed the inverse [24]. In 2012, D-Wave Systems Inc. released the first hardware implementation of QA algorithm, usually referred to a Quantum Processing Unit (QPU). Unlike general gate-based quantum computing, D-Wave QPU executes a very specific type of quantum computing and can handle solving very specific problems that can be embedded in their QPU topology.

D-Wave QPU

The D-Wave systems (QPU) operates with thousands of qubits specialized to run quantum annealing tasks. A qubit is the most trivial two-level state, representing the fundamental unit for encoding quantum information. Unlike a classical bit that can be in one state at a time (0 or 1), a qubit can be in one of them or in a superposition of both at same time. This ability of qubits to occupy multiple states simultaneously enables parallelism which give an advantage to quantum algorithms for processing several possibilities at the same time. Additionally, thanks to superposition, a qubit can be entangled with another, creating correlations that are of great importance for quantum communication and computation. In Dirac notation, a qubit $|\text{qubit}\rangle$ is represented as:

$$|\text{qubit}\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle), \quad \textbf{s.t.} \quad |0\rangle = \begin{bmatrix}1\\0\end{bmatrix}, \quad |1\rangle = \begin{bmatrix}0\\1\end{bmatrix}. \tag{3.1}$$

The physical representation of a qubit is flexible, it can be realized using several different systems, including the spin of an electron, superconducting circuits, quantum dots, trapped ions, photons and so on. D-Wave QPU qubits are based on superconducting qubits, which represent ground states of superconducting loops. These superconducting loops are characterized by a circulating current and a corresponding magnetic field (see Fig. 3.2). Before the annealing process starts, the qubit is initialized in a superposed state as in Eq. (3.1), and its final state is only revealed after the annealing process

when the qubit collapses to ones of the classical states. This process can be explained using the energy diagram in Fig. 3.3.

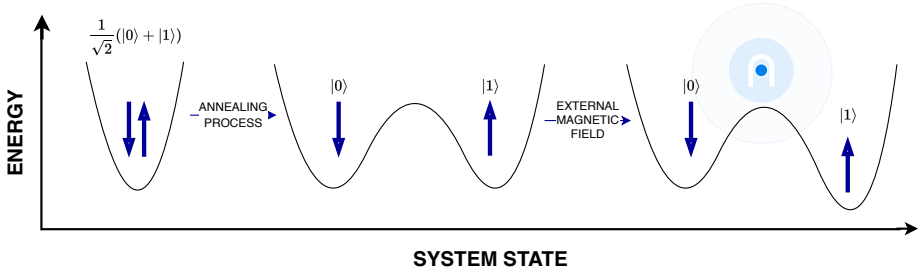


Figure 3.3: Annealing process energy diagram of qubit.

The energy state of the qubit is represented by a single valley with a single minimum energy state. When the annealing process starts, an energy barrier starts to form which splits the single minimum energy state into two valleys of equal energy minimas. The final energy state of the qubit has an equal probability of occupying either valley, with a 50% chance of being in the left valley and a 50% chance of being in the right valley. In fact, the equiprobability can be altered by applying an external stimulus (a magnetic field) to end the qubit in a preferred state.

3.2 Quantum annealing process

In the formulation stage, the optimization problem is encoded into a QUBO form or an Ising model representation [13]. The annealing process begins by initializing the system in the ground state of a driver Hamiltonian:

$$H_0 = -\sum_i \sigma_x^i. \tag{3.2}$$

typically corresponding to a uniform superposition state of equal $|0\rangle$ and $|1\rangle$, which is easily solvable. The problem to be solved is encoded into an Ising Hamiltonian (Target):

$$H_{\texttt{Ising}} = -\sum_i h_i \sigma_z^i - \sum_{i<j} J_{ij} \sigma_z^i \sigma_z^j \tag{3.3}$$

where σ_z^i is the Pauli matrix along z-axis, J_{ij} represents interaction strengths, and h_i are external fields. Then the system starts to evolve according the time-dependent Hamiltonian:

$$H(t) = A(t)H_0 + B(t)H_{\texttt{Ising}}, \tag{3.4}$$

where $A(t)$ is and $B(t)$ are time-dependent functions that are respectively monotonically decreasing and monotonically increasing. The two functions have the following properties:

$$t \to \begin{cases} 0: & A(0) >> B(0) \quad \longrightarrow \quad H(0) \approx H_0, \\ T: & A(T) << B(T) \quad \longrightarrow \quad H(T) \approx H_{\texttt{Ising}}. \end{cases} \tag{3.5}$$

where T refers to the annealing time. The system starts in the ground state of H_0 and ideally remains in the ground state throughout the annealing process, a concept known as adiabatic evolution [25, 26]. If the annealing is slow enough and there are no significant thermal or decoherence effects, the system transitions into the ground state of $H_{\texttt{Ising}}$, corresponding to the optimal solution. However, non-adiabatic effects, thermal excitations, or an insufficiently slow schedule can cause the system to end in an excited state rather than the true ground state.

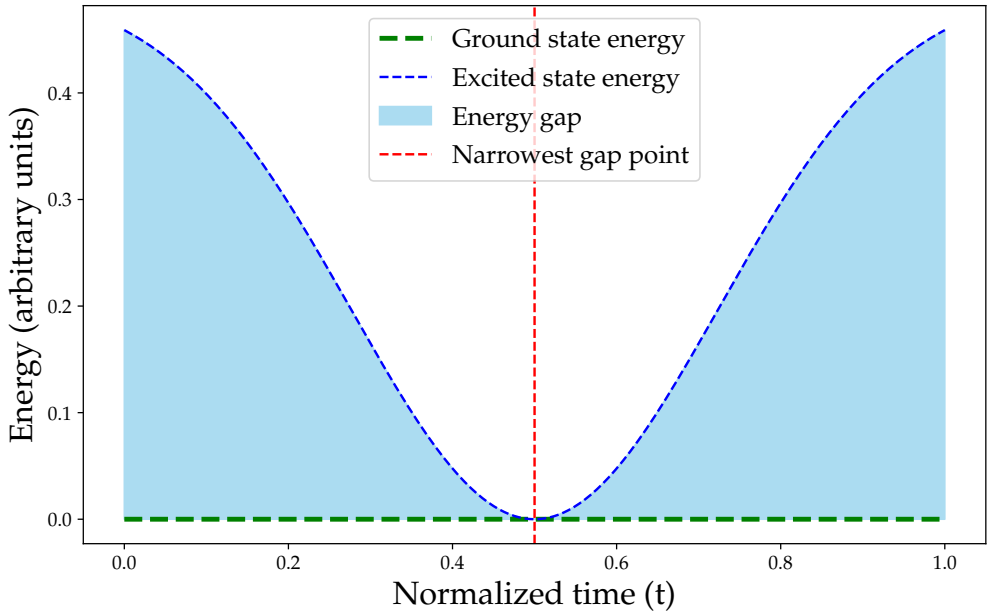


Figure 3.4: Illustrative energy gap narrowing during quantum annealing.

To ensure adiabaticity, the annealing time must be long enough relative to the energy gap between the ground and excited states, particularly at points where the gap narrows (See Fig. 3.4). This is critical as a narrow gap increases the likelihood of non-adiabatic transitions. Quantum annealers attempt to mitigate these challenges through techniques such as error correction [14] and careful tuning of the annealing schedule. For more details on quantum annealing, we recommend the reader read Refs. [13, 25].

3.3 Encoding and embedding

On D-Wave machine, the optimization problem is encoded using the Ising model or the QUBO form, similar to quantum annealing procedure, but involves a process known as embedding. Embedding is a technique used to map the variable of a given problem into a subgraph of the quantum annealer physical qubits.

3.3.1 Problem encoding

The first step is to encode the optimization problem into a form supported by the annealer hardware:

- QUBO: The problem is represented using a quadratic polynomial with binary variables (0 or 1).

$$\min_{x\in\{0,1\}^n}\left(\sum_i Q_{i,i}x_i + \sum_{i<j} Q_{i,j}x_i x_j\right) \rightarrow \min_{x\in\{0,1\}^n} x^T Q x, \tag{3.6}$$

  where Q is the QUBO matrix with entries Q_{ij}.

- Ising model: The problem is represented using spin variables ($+1$ or -1). The energy function to be minimized is:

$$\min_{s\in\{-1,1\}^n} \mathrm{E}_{\texttt{Ising}}(\boldsymbol{s}) = \min_{s\in\{-1,1\}^n}\left(\sum_{i=1}^{N} h_i s_i + \sum_{i=1}^{N}\sum_{j=i+1}^{N} J_{ij}s_i s_j\right), \tag{3.7}$$

  where h_i are local fields strengths, and J_{ij} are nearest-neighbor coupling strenghts.

3.3.2 Problem embedding

D-Wave QPU processors have limited connectivity topology among qubits. They are connected by programmable inductive couplers. For instance, the qubits can be arranged in a Chimera topology or Advantage topologies with restricted connections between qubits. These topologies can be visualized in the D-Wave documentation [1]. When the problem has variables that are not directly mapped to the hardware topology, D-Wave uses a technique called minor-embedding to map the problem onto the qubits that can be connected. Minor-embedding consists of finding a subgraph within the hardware that can represent the same problem. In Fig. 3.5, we illustrate how a triangular graph (top) can be embedded onto the topology of a hardware graph with four qubits. The triangular graph is a representation of a problem with three vertices (a, b, and c) and weighted edges connecting them. As the hardware graph does not provide direct connections between all pairs of qubits required by the problem graph, minor embedding is employed. This entails mapping vertices onto physical qubits and constructing chains of multiple qubits to represent singular vertices or edges. Two potential embeddings are shown. In the right embedding, vertex b is mapped to a chain of qubits 0 and 3, while a and c are mapped to qubits 2 and 1, respectively. The left embedding represents the typical configuration, where chains are rearranged to achieve the same logical connections.

[1] D-Wave documentation

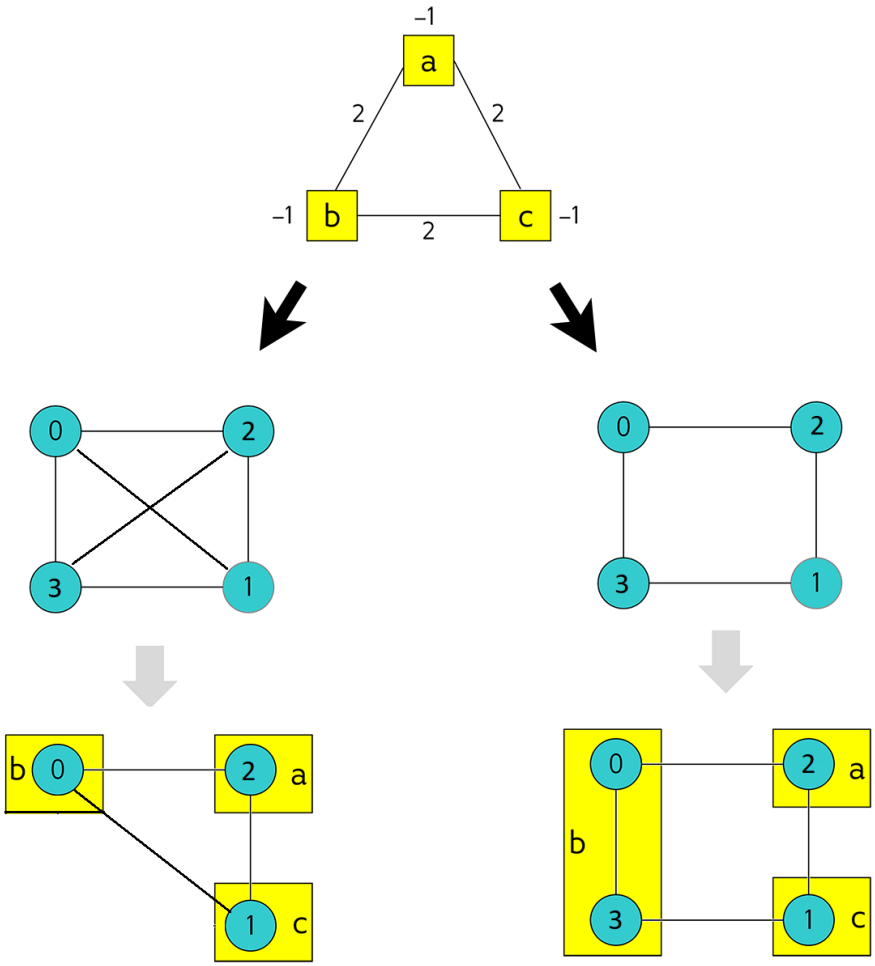


Figure 3.5: Example of minor-embedding (source D-Wave).

3.4 QPU jobs

To explore quantum annealing machines, CINECA Quantum Computing Lab provided premium access to the D-Wave QPUs through their CINECA work. We prepared a python jobs script to directly submit jobs to the available QPU Solvers:

```
    def QPUJob(Q, num_reads, label):
        # INIT QPU SOLVER
        bqm = BinaryQuadraticModel.from_qubo(Q)
        sampler = EmbeddingComposite(DWaveSampler())
        self.response = sampler.sample(bqm, num_reads, label=label)

        # GET RESULT
        response = sampler.sample(bqm, num_reads)
        solution = list(response.first.sample.values())
```

```
10          energy = response.first.energy
11          ........
```

To solve a QUBO problem on using QPUs. The QUBO matrix Q of the problem much be constructed as in Eq. (3.6). To show how to construct the QUBO matrix, lets consider the problem of finding the MaxCut of the following undirected 5 vertices graph:

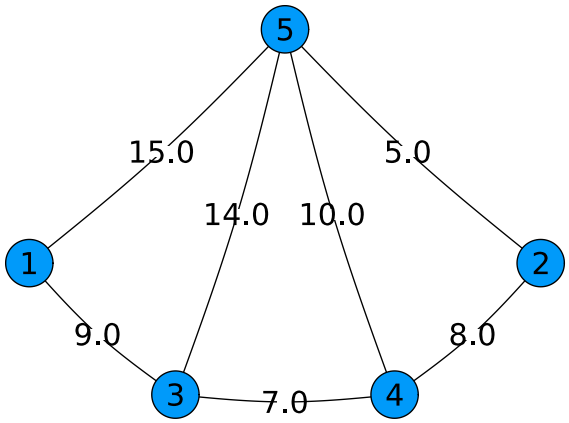


Figure 3.6: Weighted five vertices graph.

To find MaxCut, we need to minimize the energy function:

$$\min_{x\in\{0,1\}^5} E^5_{\texttt{maxcut}} = \min_{x\in\{0,1\}^n} \left(- \sum_{(i,j)\in E} w_{ij}(x_i + x_j - 2x_i x_j) \right). \tag{3.8}$$

For the 5 vertices graph in 3.6, the expanded energy function writes as:

$$\begin{aligned} E^5_{\texttt{maxcut}} = & -w_{15}x_1 - (w_{24}+w_{25})\,x_2 - (w_{34}+w_{35})\,x_3 \\ & + w_{13}\,(x_1\,(2x_3-1) - x_3) + x_4\,(w_{24}\,(2x_2-1) + w_{34}\,(2x_3-1) - w_{45}) \\ & + x_5\,(w_{15}\,(2x_1-1) + w_{25}\,(2x_2-1) + w_{35}\,(2x_3-1) + w_{45}\,(2x_4-1))\,. \end{aligned} \tag{3.9}$$

The QUBO matrix from $E^5_{\texttt{maxcut}}$ can be represented as:

$$Q_{5\times 5} = \begin{pmatrix} -24 & 0 & 18 & 0 & 30 \\ 0 & -13 & 0 & 16 & 10 \\ 18 & 0 & -30 & 14 & 28 \\ 0 & 16 & 14 & -25 & 20 \\ 30 & 10 & 28 & 20 & -44 \end{pmatrix}. \tag{3.10}$$

By submitting a job to the QPU with matrix Q and 100 reads:

```
QPUJob(Q, 100, "MaxCutQPU"),
```

the QPU returns the results:

```
Initializing QPU solver...
Submitting job to the QPU...

=== QPU Job Completed ===
Label: MaxCutQPU
Total time (seconds): 0.0789
Number of reads: 100

=== Detailed Response ===
Best configuration: [0, 1, 1, 0, 1]
Best energy: -49.0
All samples:
x    1  2  3  4  5  energy   num_oc. chain_.
0    0  1  1  0  1  -49.0      12     0.0
1    1  1  1  0  0  -49.0       8     0.0
2    1  0  0  1  0  -49.0      45     0.0
3    0  0  0  1  1  -49.0      16     0.0
4    1  0  1  1  0  -47.0       5     0.0
5    0  1  0  0  1  -47.0       5     0.0
6    0  0  1  0  1  -46.0       1     0.0
7    1  1  0  1  0  -46.0       3     0.0
8    0  0  0  0  1  -44.0       1     0.0
9    1  1  1  1  0  -44.0       1     0.0
10   0  1  1  0  0  -43.0       1     0.0
11   0  0  1  1  0  -41.0       1     0.0
12   1  0  1  0  0  -36.0       1     0.0
['BINARY', 13 rows, 100 samples, 5 variables]
Number of solutions: 13
Occurrences of best solution: 12
```

The results contains multiple candidate solutions. Each solution includes:

- Sample: $\{x_1 = 0/1,\ x_2 = 0/1,\ x_3 = 0/1,\ x_4 = 0/1,\ x_5 = 0/1\}$
- Energy: Energy of the sample.
- Occurrences: Number of times the solution appeared.

The results above shows that the QPU were able to find 4 solutions with minimum energy -49.0. In fact, for MaxCut only two solutions out of 4 are non-trivial since the other 2 solutions can be deduced using symmetry. The QPU returns also a `SampleSet` with QPU timing. In the following table, we provide the timing information:

Table 3.1: Timing information for QPU execution

Timing metric	**Value (µs)**
QPU Sampling time	11050.0
QPU Anneal time per sample	20.0
QPU Readout time per sample	69.92
QPU Access time	26810.76
QPU Access overhead time	809.24
QPU Programming time	15760.76
QPU Delay time per sample	20.58
Post-processing overhead time	1.0

It is observed that the QPU access time and programming time are the main contributors to the overall execution time, reflecting the classical overheads involved in managing the quantum hardware. The annealing time per sample is relatively short, indicating that the QPU is performing the quantum annealing process efficiently. Furthermore, the post-processing time is short, which suggests that the readout and result handling are not bottlenecks in the execution. It is worth mentioning, current QPUs can solve only problems that fit their topologies. In other words, the problem variables should not be more than the number of available qubits within the QPU.

Chapter 4

AlmoQUBO Library

In this chapter, we introduce `AlmoQUBO`, a robust and versatile C++ library designed for solving Quadratic Unconstrained Binary Optimization (QUBO) problems that can be translated into graph problems. `AlmoQUBO` combines state-of-the-art algorithms with the performance benefits of high-performance computing. The library integrates two high-performance optimization solvers: Simulated Annealing and Scatter Search, both of which have been meticulously developed and optimized for efficiency and scalability. We sought to simplify the user experience by enabling easy control of library parameters and solvers. In the following sections, we provide further details about the design and structure of our proposed library.

4.1 Design

The design and architecture of AlmoQUBO have been meticulously constructed to guarantee both flexibility and efficiency. In this section, we provide an overview of the fundamental components and structural choices that underpin the library. We emphasize the modular design that enables users to seamlessly adjust parameters and solvers during runtime. Furthermore, we discuss design considerations that ensure scalability, and maintenance.

The library is organized into a modular structure, with separate headers and source files as shown in Fig. 4.1. This modular approach does not only speedup compilation time, but also enhances readability, maintainability, scalability, and allows the integrability of new solvers.

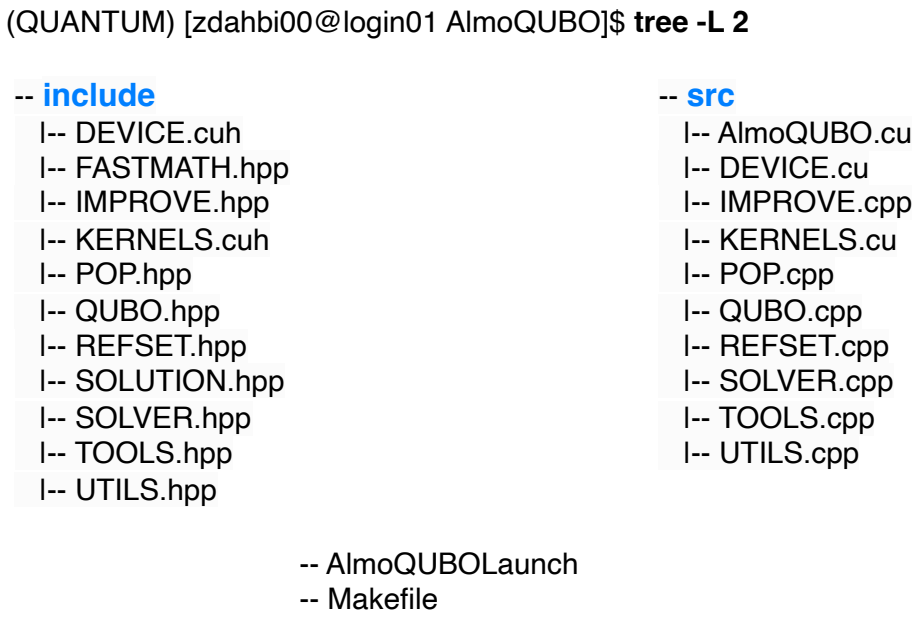


Figure 4.1: Library codebase structure.

In what follows, we explain the main parts of our library. The header files, in `include/`, define the functions, so the implemented functions can be recognized and used. In the `src/` directory, the corresponding source files implement the functionality declared in the headers:

- `DEVICE.cu`, `KERNELS.cu`: These files contain the CUDA kernels and device functions needed by some GPU kernels.
- `QUBO.cpp`: This source file implement the methods and data structures to read and maintain the QUBO problem.
- `POP.cpp`: This source file implement the core CPU code handling the creating of the population set for the QUBO problem.
- `REFSET.cpp`: This source file defines the logic for reference set creation and management.
- `IMPROVE.cpp`: This source file contains methods for local refinements on the CPU.
- `SOLVER.cpp`: This contains the definitions of all implemented solvers. Any new implemented solvers will be defined in this source file.
- `TOOLS.cpp`, `UTILS.cpp`: These provide the implementation for auxiliary tools, utility functions, and general C++ headers.
- `SOLUTION.cpp`: This source file defines a class that stores the final solution and all information about it.

This structured organization allows for efficient debugging, easier code updates, and the addition of new features without disrupting existing functionality.

4.2 Control

The AlmoQUBO library is made up of thousands of lines of code that are further optimized with aggressive compiler flags. Therefore, it becomes impractical to recompile for every modification in the problem parameters. To avoid the overhead of recompilation whenever a parameter is modified, we introduced a text-based launch file, which allowed QUBO problems and their parameters to be defined at runtime. This approach enables parameter modifications without the necessity of recompilation, thereby reducing the necessity for repetitive builds.

The launch file, **AlmoQUBOLaunch**, serves as a configuration file for the **AlmoQUBO Library**, which allows users to configure various parameters for optimization tasks. The file defines settings for two implemented solvers: Simulated Annealing (SA) and Scatter Search (SS), providing flexibility in choosing how to solve problems. A typical launch file for our proposed library must look like:

```
================================================================================
                          AlmoQUBO Library Config File
================================================================================

--------------------------------------------------------------------------------
                              LAUNCHING PARAMETERS
--------------------------------------------------------------------------------

    QUBO File           : path_to_qubo
    Solver Choice       : "SA" or "SS"
    File Type           : "Gset" or "UpperQ"
    Verbose             : 1 or 0
    UseTarget           : 1 or 0
    TargetCost          : xxxxx

--------------------------------------------------------------------------------
                      SIMULATED ANNEALING PARAMETERS
--------------------------------------------------------------------------------

    Maximum Temperature         : xxxxxx
    Heating Step                : xxxxxx

--------------------------------------------------------------------------------
                        SCATTER SEARCH PARAMETERS
--------------------------------------------------------------------------------

    Construction Option         : 0, 1, 2, or 3
    Local Search Option         : 0 or 1
```

```
Alpha Parameter             : 0 <= x <=1
CPU Recursive Depth         : xx
Number of Solutions         : xxxx
Number of Iterations        : xxxx
Minimum Hamming Distance    : xxxx
SS Seed                     : xxxx
Local Search GPU Max Depth  : xxxx
Local Search GPU Min Delta  : xxxx
Diverse Set Size            : xxxx
Population Set Size         : xxxx
Reference Set Size          : xxxx

=========================================================================
```

The configuration file contains three important sections:

1. LAUNCHING PARAMETERS
   This section enables the user to establish general settings to address the QUBO problem.

   - `QUBO File`: The path to the definition of the QUBO problem.
   - `Solver Choice`: This is used to specify the solver to be used, such as `SA` for Simulated Annealing or `SS` for Scatter Search.
   - `File Type`: Used to specify the type of QUBO input file. The library is able to understand two input types:
     - "`Gset`" if QUBO input is an edge list where the data is organized as follows:

$$\begin{array}{ccc} |V| & |E| & \\ v_1 & v_2 & w_{12} \\ v_1 & v_3 & w_{13} \\ \vdots & \vdots & \vdots \\ v_k & v_l & w_{kl} \\ \vdots & \vdots & \vdots \end{array}$$

     - "`UpperQ`" if the QUBO input is a $n \times n$ upper triangular matrix stored as follows:

$$\begin{array}{ccccccc} Q_{11} & Q_{12} & Q_{13} & Q_{14} & Q_{15} & \cdots & Q_{1n} \\ Q_{22} & Q_{23} & Q_{24} & Q_{25} & \cdots & Q_{2n} & \\ Q_{33} & Q_{34} & Q_{35} & \cdots & Q_{3n} & & \\ Q_{44} & Q_{45} & \cdots & Q_{4n} & & & \\ Q_{55} & \cdots & Q_{5n} & & & & \\ \ddots & \vdots & \vdots & & & & \\ Q_{nn} & & & & & & \end{array}$$

- `Verbose`: To enable monitoring by printing the objective value discovered at each step.
- `UseTarget`, `TargetCost`: These two options are used to activate the feature of stopping the solver when the target objective is reached.

2. SIMULATED ANNEALING PARAMETERS This section defines the parameters of Simulated Annealing.
   - `Maximum Temperature`: To set a suitable maximum temperature for the problem being solved.
   - `Heating Step`: To set a suitable heating schedule for the problem being solved.
3. SCATTER SEARCH PARAMETERS
   - `Construction Option`: Specifies the desired GPU method for generating initial trials.

     ```
     0: RANDHam_GPU | 1: SSRand_GPU | 2: SSGRASP_GPU
     ```

   - `Local Search Option` : Specifies the GPU method for local refinement of the trials:

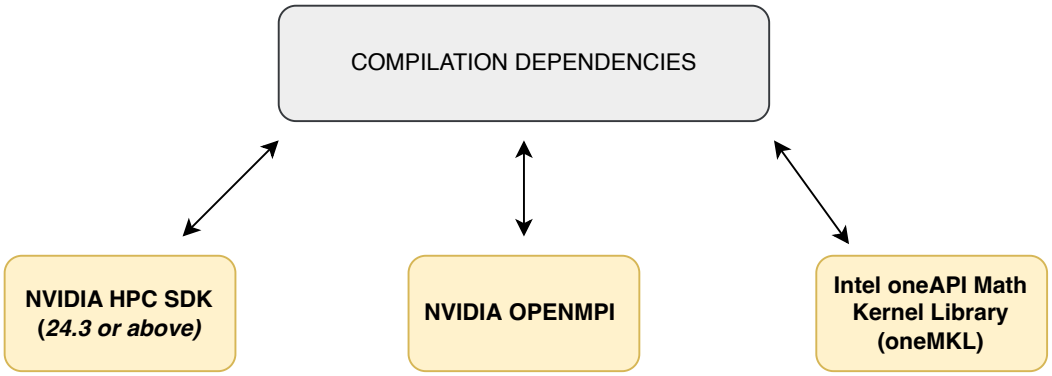
     ```
     0: BESTLS_GPU | 1: LSRAND_GPU
     ```

   - `CPU Recursive Depth`: Defines the number of recursive calls for the CPU-based local refinement method.
   - `Number of Solutions`: For specifying the number of initial trials to be generated by the GPU.
   - `Number of Iterations`: Sets the maximum number of iterations for the solver.
   - `Minimum Hamming Distance`: Specifies the minimum Hamming distance between generated trials when using the Hamming distance GPU generation method.
   - `SS Seed` : Sets the seed value for cuRAND to ensure reproducibility, and to try different explorations.
   - `Local Search GPU Max Depth`: Determines the maximum number of refining iterations performed on the GPU during local search.
   - `Local Search GPU Min Delta`: Sets the minimum tolerance value to accept a bit flip during local search.
   - `Diverse Set Size`: Specifies the number of solutions to include in the diverse set.
   - `Population Set Size`: Defines the total number of solutions in the population set.

- `Reference Set Size`: Specifies the number of solutions in the reference set.

4.3 Compilation

The library comes with a builtin main function (AlmoQUBO.cu) located in src/ folder. It allows compiling and using the library as it is. To compile AlmoQUBO, a Makefile is provided. A successful compilation requires loading the following dependencies:



On the CINECA Leonardo supercomputer [27] (the computational environment where this library was developed) the modules can be loaded as:

```
module load nvhpc/24.3 openmpi/4.1.6--nvhpc--24.3 intel-oneapi-mkl/
```

To provide greater flexibility, we distribute a `CMakeLists.txt` file that enables the AlmoQUBO codebase to be compiled as either a shared or static library. This modular approach enhances the usability of AlmoQUBO, and promotes its adoption and integration across diverse workflows and deployment scenarios.

Chapter 5

Benchmarks: The NP-hard MaxCut problem

The algorithms implemented in this work are benchmarked against the MaxCut, a well know NP-hard problem widely used for benchmarking both classical and quantum optimization algorithms. The MaxCut is a graph theoretic problem that rises in many practical applications in combinatorial optimization [28], statistical physics [29], circuit design [30], finance [31], network optimization [32] and so on. Solving a MaxCut problem is equivalent to partitioning a graph into two partitions such that the number of interconnects between them is maximized. The MaxCut can be interpreted as a Unconstrained Binary Optimization (QUBO) problem, where a solution can be represented using binary entries to indicate that a given vertex belongs to a specific partition. In this work, we are particularly interested in undirected graphs.

5.1 Search space complexity

As mentioned earlier, the MaxCut problem is NP-hard, meaning there is no known algorithm that can solve it in polynomial time. The proof of MaxCut NP-hardness has been proven by Karp [33]. Other proofs can typically follow by making a reduction from other problems almost known to be NP-hard problems, such as 3-SAT [34] for instance. In this section, we use the curse of dimensionality of the search space to show how complex is this optimization problem.

Definition 1
Given a directed or an undirected graph $G = (E,V)$, let H_G be the space of solutions (the search space). For a general partition (a cut solution), we choose k vertices from $|V|$ to put into a set $\mathcal{A}_0$ while the rest ($|V|-k$) automatically go to set $\mathcal{A}_1$. The number of possible choices is given by the binomial coefficient $\binom{|V|}{k}$. The dimension of H_G is the total number of all the possible choices (bipartitions), and is given by:

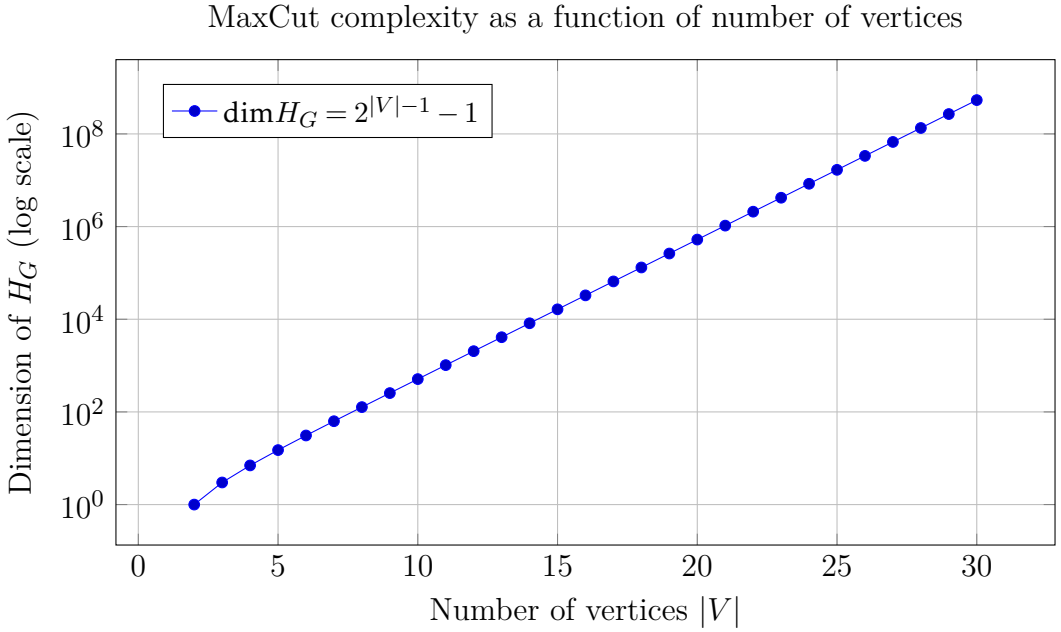
$$\dim H_G = \sum_{k=0}^{|V|} \binom{|V|}{k} = 2^{|V|}. \tag{5.1}$$

Definition 2
Given two bipartions, $\mathcal{A}_0$ and $\mathcal{A}_1$ of a graph G. Swapping bipartitions

$$cut(\mathcal{A}_0 \cup \mathcal{A}_1) = cut(\bar{\mathcal{A}}_0 \cup \bar{\mathcal{A}}_1) = (\mathcal{A}_1 \cup \mathcal{A}_0), \tag{5.2}$$

leaves cut value invariant. By eliminating trivial solutions such as $\emptyset \cup V$ or $V \cup \emptyset$, the total number of valid partitions reduces:

$$\dim H_G = 2^{|V|-1} - 1. \tag{5.3}$$



It is evident that, as the number of vertices $|V|$ increases, the dimension of the search space increases exponentially, following a super-linear growth in logarithmic scale. This demonstrates the escalating computational complexity of MaxCut, as the size of the search space significantly expands with even a modest increase in the number of vertices.

5.2 Benchmark dataset

To objectively assess the efficiency of our implemented algorithms, we utilize the famous Gset benchmark collection [16, 15]. The Gset comprises various graphs with different search space sizes and graph topologies, which allows testing a range of structural complexities.

5.2.1 Gset format

A Gset instance describes a weighted undirected graph in a text format. The text file contains specific information about the graph, including the number of vertices and edges, followed by an edge list telling the endpoint of each edge and its corresponding weight (see Table 5.1).

Table 5.1: Gset instance structure

Number of vertices	**Number of edges**
$\|V\|$	$\|E\|$

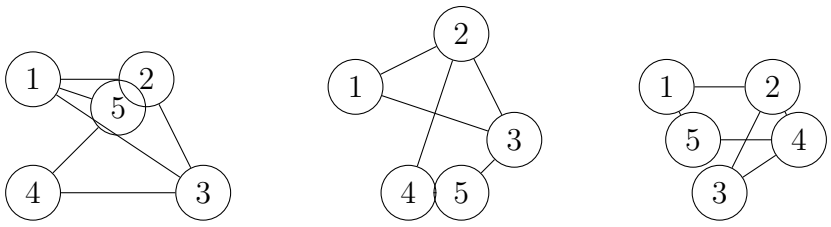
Vertex i	**Vertex** j	**Weight** w_{ij}
i_0	j_0	$w_{i_0 j_0}$
i_1	j_1	$w_{i_1 j_1}$
i_2	j_2	$w_{i_2 j_2}$
$\vdots$	$\vdots$	$\vdots$
$i_{\|E\|-1}$	$j_{\|E\|-1}$	$w_{i_{\|E\|-1} j_{\|E\|-1}}$

We mainly target the instances of `Set1` in the Gset data [16], which is widely used for experimentations, see for instance Myklebust [1], Marti et al. [2], Festa et al. [4], and Burer et al. [3]. By conducting these tests, we gain valuable insights into the adaptability of the solvers to varying problem characteristics by examining factors such as convergence behavior, scalability, and sensitivity to graph structure.

5.2.2 Graph topology

The graphs in `Set1` are undirected and have weighted edges with ± 1 weights. Their geometric structure can be classified into three distinct categories:

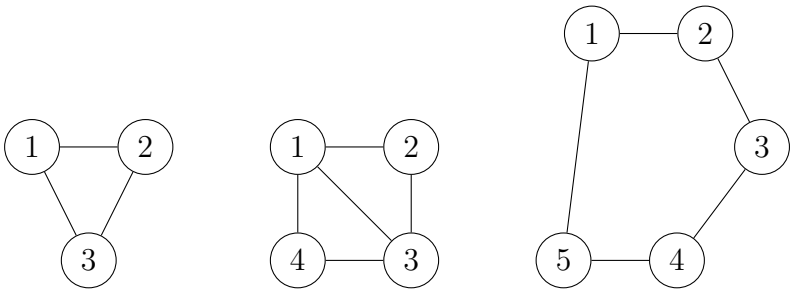
- *Random graphs:* The edges are formed randomly rather than based on a specific structural rule. These graphs follows the Erdős–Rényi (ER) model [35], where each edge exists with a probability p.



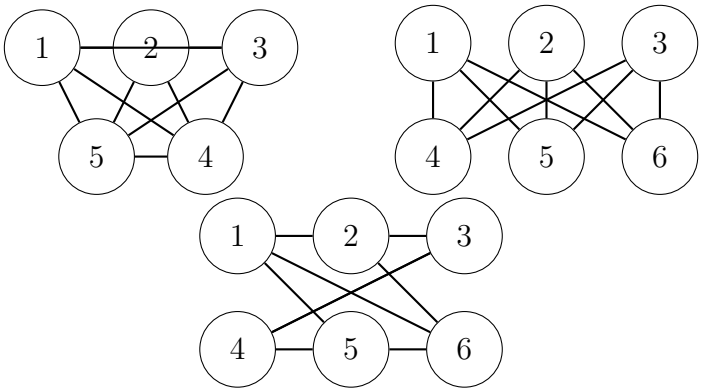
- *Planar graphs:* These graphs can be embedded in the plane, meaning they can be drawn on flat surfaces while avoiding a crossover between the edges. For connected planar graphs, the Euler's Formula holds true:

$$|V| - |E| + |F| = 2, \tag{5.4}$$

where $|V|$, $|E|$, and $|F|$ represent the number of vertices, edges, and faces, respectively. It worth stating that according to the *Kuratowski's Theorem* [36], any graph that contains a subgraph homeomorphic to the complete graph K_5 or the complete bipartite graph $K_{3,3}$ is not a planar graph.



- *Toroidal graphs:* These graphs can be embedded on a torus, without an edge crossing another. Unlike the plane, the torus provides extra dimension allowing some non-planar graphs to be embedded on a torus (for instance, the complete graph K_7).



5.3 Simulated Annealing results

In this section, we present benchmarks collected using our parallel implementation of the Simulated Annealing algorithm. We show by results, how our implemented SA solver performs across various graph sizes and graph topologies. As demonstrated in Section 5.1, the goal of these benchmarks is to ascertain and quantify the solver's efficiency in finding high quality solutions for the Gset instances in a short runtime.

We conducted two experiments. In the first experiment, we chose a temperature sufficiently high and a small heating step. In the second experiment, we applied fine-tuned parameters to each specific problem. During the first experiment, we used the following SA parameters:

$$T_{\max} = 9.0 \times 10^4, \quad \textbf{and} \quad \alpha = 8.0 \times 10^{-5}. \tag{5.5}$$

First experiment: Same parameters for all instances

We tested the solver on 2, 4, 8, 16, 32 compute nodes with 32 CPUs per node, using the parameters in Eq. (5.5) for all instances. In this experiment, we want to check the scalability of the solver in terms of solutions quality while the runtime remains more or less constant. In other word, the parallelization serves for discovering new paths

that may lead to new solutions. In Table 5.2, we present the results obtained across 8 compute nodes:

Table 5.2: 8 nodes: Comparison of MaxCut solutions found by Ref. [1] (see also the data in the Appendix) and our SA code (8 nodes).

Gset	**Our SA**	**SA in [1]**	**±%**	**×**	**G**	**Our SA**	**SA in [1]**	**±%**	**×**
g1	11624	11621	0.03	5.52	g28	3298	3298	0.0	4.24
g2	11620	11612	0.07	4.8	g29	3405	3394	0.32	4.83
g3	11622	11618	0.03	5.56	g30	3413	3412	0.03	4.82
g4	11646	11644	0.02	5.53	g31	3309	3309	0.0	4.82
g5	11631	11628	0.03	5.55	g32	1410	1410	0.0	4.34
g6	2178	2178	0.0	5.68	g33	1380	1376	0.29	4.33
g7	2006	2006	0.0	4.67	g34	1384	1382	0.14	4.36
g8	2005	2005	0.0	4.71	g35	7683	7485	2.65	4.84
g9	2054	2054	0.0	4.71	g36	7677	7473	2.73	3.86
g10	2000	1999	0.05	3.52	g37	7685	7484	2.69	5.0
g11	564	564	0.0	4.22	g38	7687	7479	2.78	4.89
g12	556	554	0.36	3.12	g39	2408	2405	0.12	3.39
g13	582	580	0.34	4.34	g40	2398	2378	0.84	4.5
g14	3063	3063	0.0	5.34	g41	2405	2405	0.0	3.38
g15	3050	3049	0.03	3.53	g42	2477	2465	0.49	4.55
g16	3052	3050	0.07	3.49	g43	6660	6658	0.03	5.06
g17	3047	3045	0.07	3.52	g44	6650	6646	0.06	4.98
g18	992	990	0.2	3.89	g45	6654	6652	0.03	4.96
g19	906	904	0.22	3.22	g46	6649	6647	0.03	5.08
g20	941	941	0.0	4.32	g47	6657	6652	0.08	5.01
g21	931	927	0.43	3.24	g48	6000	6000	0.0	17.52
g22	13359	13158	1.53	4.62	g49	6000	6000	0.0	13.65
g23	13344	13116	1.74	4.09	g50	5874	5858	0.27	13.43
g24	13337	13125	1.62	4.04	g51	3848	3841	0.18	4.4
g25	13340	13119	1.68	4.48	g52	3851	3845	0.16	3.58
g26	13328	13098	1.76	4.07	g53	3849	3845	0.1	3.68
g27	3341	3341	0.0	4.85	g54	3852	3845	0.18	3.61

The ±% column indicates the percentage improvement in solution quality achieved by our method compared with the reference algorithm [1] for each instance. It can be seen that several instances exhibit modest but noteworthy enhancements, such as `g22` (+1.53%) and `g23` (1.74%), indicating that our SA algorithm yields superior solutions in numerous scenarios. Moreover, the column × demonstrates the speedup of our AlmoQUBO library over the reference method in obtaining comparable or superior outcomes in a shorter runtime. For example, problem instances like `g48` and `g49` exhibit speedups of 17.52× ann 13.65×, respectively, demonstrating the efficiency of our implementation in obtaining optimal or near-optimal solutions more quickly. In general, while certain instances, such as `g6`, `g7`, and `g8`, demonstrate no enhancement in solution quality (0%), the majority of the results demonstrate that our SA algorithm enhances solution quality and offers significant computational advantages over the

reference method, particularly for larger problem instances.

In a second run, we attempted to initiate the solver with additional nodes to evaluate its ability to find other better solutions compared to those obtained with 8 nodes. In Table 5.3, we present the outcomes obtained across 32 compute nodes.

Table 5.3: Comparison of MaxCut solutions found in Ref. [1] (see also the data in the Appendix) and our SA code (32 nodes).

Gset	**Our SA**	**SA in [1]**	±%	×	**Gset**	**Our SA**	**SA in [1]**	±%	×
g1	11624	11621	0.03	5.46	g28	3298	3298	0.0	4.23
g2	11620	11612	0.07	4.72	g29	3405	3394	0.32	3.61
g3	11622	11618	0.03	5.52	g30	3413	3412	0.03	3.6
g4	11646	11644	0.02	5.55	g31	3310	3309	0.03	3.62
g5	11631	11628	0.03	4.39	g32	1410	1410	0.0	3.8
g6	2178	2178	0.0	5.6	g33	1380	1376	0.29	3.21
g7	2006	2006	0.0	4.65	g34	1384	1382	0.14	4.32
g8	2005	2005	0.0	3.51	g35	7684	7485	2.66	4.86
g9	2054	2054	0.0	3.49	g36	7680	7473	2.77	3.82
g10	2000	1999	0.05	3.46	g37	7688	7484	2.73	4.17
g11	564	564	0.0	3.48	g38	7688	7479	2.79	4.08
g12	556	554	0.36	3.09	g39	2408	2405	0.12	3.36
g13	582	580	0.34	3.22	g40	2399	2378	0.88	3.41
g14	3064	3063	0.03	5.31	g41	2405	2405	0.0	3.39
g15	3050	3049	0.03	3.53	g42	2477	2465	0.49	3.42
g16	3052	3050	0.07	3.51	g43	6660	6658	0.03	5.08
g17	3047	3045	0.07	3.51	g44	6650	6646	0.06	3.8
g18	992	990	0.2	3.93	g45	6654	6652	0.03	3.82
g19	906	904	0.22	3.29	g46	6649	6647	0.03	4.99
g20	941	941	0.0	3.98	g47	6657	6652	0.08	5.0
g21	931	927	0.43	3.19	g48	6000	6000	0.0	17.38
g22	13359	13158	1.53	4.55	g49	6000	6000	0.0	11.94
g23	13344	13116	1.74	4.07	g50	5874	5858	0.27	12.26
g24	13337	13125	1.62	4.1	g51	3848	3841	0.18	3.67
g25	13340	13119	1.68	4.43	g52	3851	3845	0.16	3.64
g26	13328	13098	1.76	4.03	g53	3850	3845	0.13	3.62
g27	3341	3341	0.0	4.59	g54	3852	3845	0.18	3.57

We observed that by using 32 nodes led to finding better results, for some specific instances, compared to 8 nodes. This improvement is due to an increase in the number of CPUs available to generate new seeds, which are then used to explore potential paths toward a potential enhanced solution. The results suggest that the efficiency of probabilistic algorithms can be significantly enhanced using HPC resources, where parallelization helps in efficiently generating diverse starting points that may lead to a better optimization outcome. To clearly observe the gains from the first run and the second run, we plot the gain differences:

$$\Delta G = \pm\%(N = 32) \; - \; \pm\%(N = 8). \tag{5.6}$$

The following plot shows the gain difference achieved for some Gset instances:

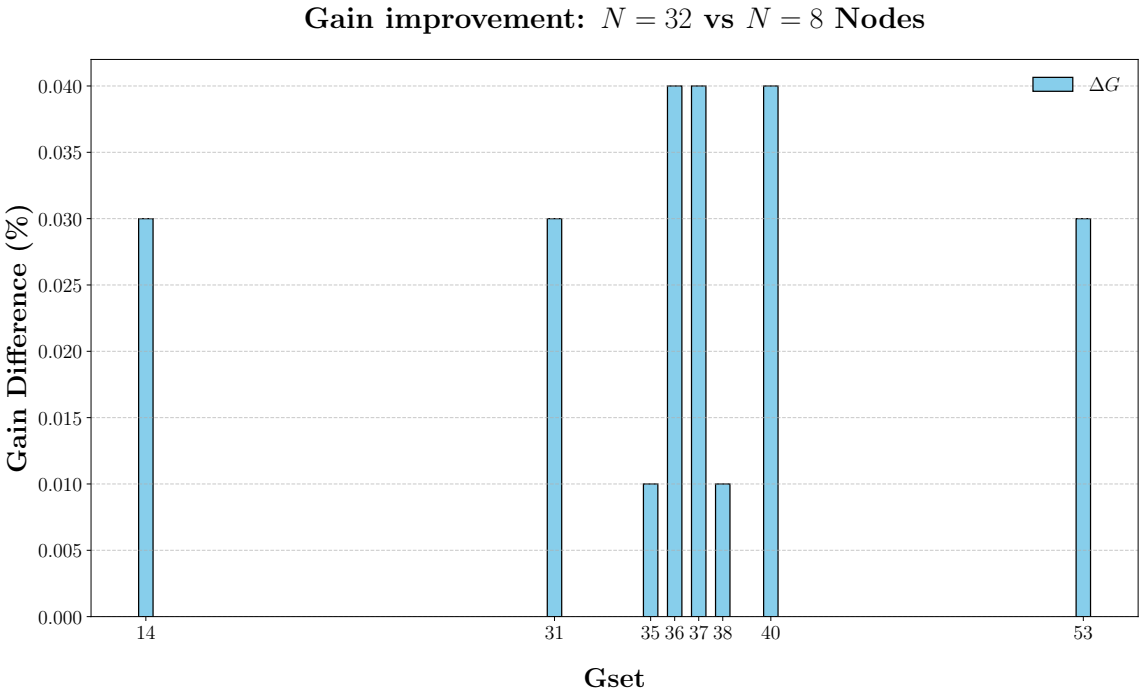


Figure 5.1: Improvement in gain between the first run ($N = 8$) and the second run $N = 32$)

Second experiment: Fine-tuning parameters for each instance

The objective of this second experiment is to demonstrate how using refined SA parameters (T_{max} and α) can help achieve an optimal balance between solution quality and time to solution. To choose the best parameters, we performed multiple runs by evaluating a range of parameter values, analyzing their effects on convergence speed, solution stability, and the final objective value. In Table 5.4, we report the best parameters that lead to the same solutions obtained in the first experiment in shorter time:

Table 5.4: Results of SA parameter tuning for problem instances

Gset	Obj	BRank	T_{max}	α	$t(s)$
1	11624	9	2.73×10^4	0.0015	1.29
2	11620	0	2.67×10^4	0.0014	1.65
3	11622	28	2.75×10^4	0.001	2.33
4	11646	31	2.69×10^4	0.001	2.36
5	11631	1	2.64×10^4	0.0015	1.56
6	2178	30	6.74×10^3	0.008	0.127
7	2006	7	6.65×10^3	0.008	0.201
8	2005	14	6.88×10^3	0.008	0.123
9	2054	7	7.18×10^3	0.006	0.15
10	2000	7	6.16×10^3	0.006	0.132
11	564	12	2.53×10^3	0.0007	0.255
12	556	6	2.29×10^3	0.0009	0.202
13	582	24	2.54×10^3	0.0009	0.229
14	3064	16	1.78×10^4	0.0002	4.97
15	3050	1	1.65×10^4	0.0004	2.45
16	3052	7	1.91×10^4	0.0004	2.75
17	3047	21	1.67×10^4	0.0002	5.73
18	992	20	4.31×10^3	0.0001	2.52
19	906	18	3.83×10^3	0.0001	2.75
20	941	8	4.15×10^3	0.0001	2.51
21	931	4	4.28×10^3	0.0001	2.55
22	13359	19	6.71×10^4	0.0009	4.17
23	13344	3	6.12×10^4	0.0009	3.83
24	13337	4	6.36×10^4	0.00085	4.11
25	13340	5	8.36×10^4	0.00085	5.32
26	13328	15	7.01×10^4	0.0007	5.5
27	3341	2	1.86×10^4	0.002	0.559
28	3298	31	1.73×10^4	0.0015	0.743
29	3405	26	1.76×10^4	0.001	1.01
30	3413	21	1.7×10^4	0.0005	1.91

In the table above, we collected the data for the trial problem instances in using only one compute node with 32 CPUs. In the table, we present the optimal MaxCut as well as the optimal rank, the achieved temperature T_{max}, the optimal heating step α, and the elapsed time in seconds during which the solution is obtained. In Fig. 5.2, we compare the results obtained with those obtained in Ref. [1].

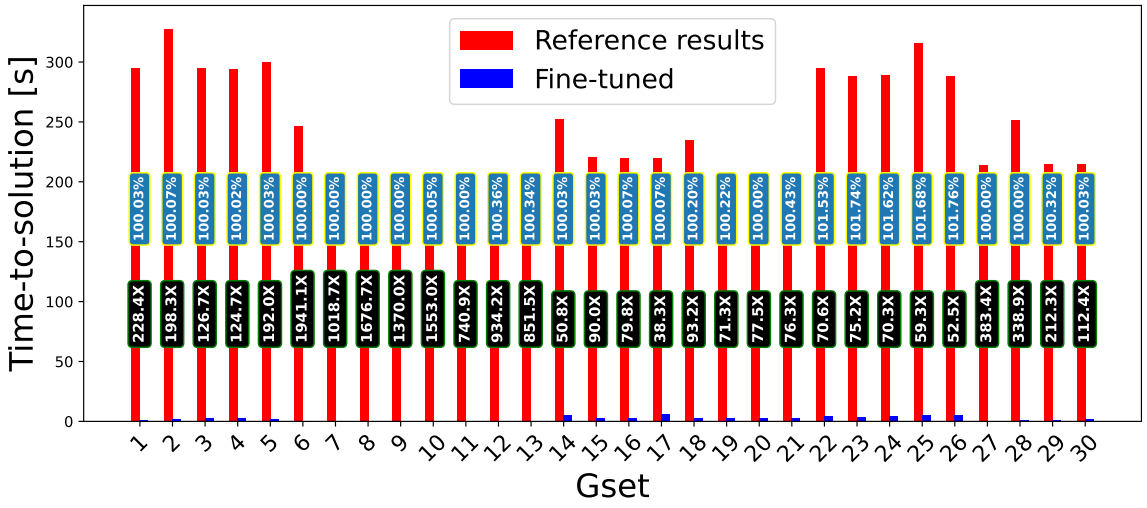
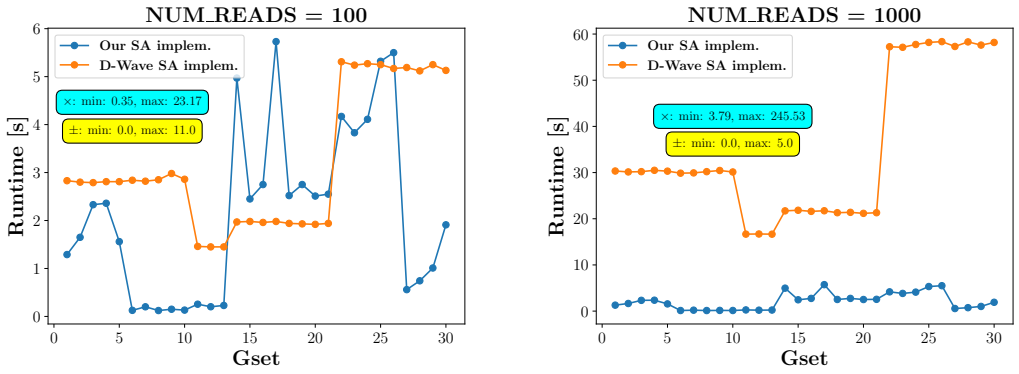


Figure 5.2: Comparison: fine-tuned SA results versus the results in Ref. [1].

As can be seen from the graph, we are getting massive speedups while finding either the same or better solutions. This results can be attributed to the following key factors:

- *Optimal exploration and exploitation balance:* Right parameters can help the solver avoid getting stuck in local minima and find the global optimum faster.
- *Reduced search space:* Good parameters allow the algorithm to find right paths and do right moves toward an optimal solution faster.

In a second comparison, we tried to compare the results achieved by our SA implementation (with fined-tuned parameters) versus the D-Wave SA implementation that comes in their Ocean SDK [37]. We gathered the outcomes and elapsed times of two runs of D-Wave SA with a count of 100 and 1000 reads. The results clearly demonstrate that our SA exceeds D-Wave SA in terms of solution quality and computational efficiency for most of the problem instances. In Fig. 5.3, we show the comparison of the objective values and the speedups versus the results of each run of the D-Wave Simulated Annealing:



(a) (b)

Figure 5.3: Comparison between results obtained my our SA implementation and the results obtained by D-Wave SA solver (for timing, see Appendix).

Figure 5.3(a) compares the results of our fine-tuned Simulated Annealing (SA) algorithm with those obtained using the D-Wave SA implementation with 100 reads. The results show that our SA either matches or outperforms the D-Wave SA, with improvements ranging from a minimum of +0 to a maximum of +11. In terms of runtime, the speedup ratio (×) between the D-Wave SA and our implementation indicates a significant acceleration for our SA, particularly in instances such as `g6` and `g8`. Figure 5.3(b) shows the D-Wave SA executed with 1000 reads. Here, we observe that D-Wave SA can find solutions better than those obtained with 100 reads. However, the runtime increases considerably, whereas our fine-tuned SA maintains constant runtime while still providing highly competitive results. These findings highlight the robustness and efficiency of our algorithm. It is also important to note that although the D-Wave SA may solve certain instances more quickly, particularly when the number of reads is around 10, it does not always achieve the objective values achieved by our SA implementation.

5.4 Scatter Search results

In this section, we present the outcomes obtained by our GPU-accelerated implementation of the Scatter Search algorithm. We will compare the performance and the results obtained with the results obtained using an advanced Scatter Search implementation by Martí el al. in [2]. It is noteworthy to mention that similar yet refined techniques, such as those employed in an advanced SS implementation in [2], were employed. Furthermore, new algorithmic techniques and strategies, such as randomized local search and iterative local search, were implemented to achieve better results with less computational time. Please refer to the chapter on Scatter Search for further details. In Table 5.5, we present the results obtained by our SS implementation and all the parameters employed to obtain them:

Table 5.5: Results obtained using SS for the first problem instances

Gset	Obj	$t(s)$	α	opC	opL	k_{CPU}	n_{sols}	n_{iters}	d_H	k_{GPU}	δ_{min}	$\lvert RSET \rvert$	$\lvert POP \rvert$
1	11624	0.86	0.55	0	1	2	700	10	400	140	-1	10	100
2	11620	1.68	0.55	0	1	2	700	10	400	140	-1	10	100
3	11622	0.82	0.55	0	1	2	700	10	400	140	-1	10	100
4	11646	0.85	0.55	0	1	2	700	10	400	140	-1	10	100
5	11631	0.86	0.55	0	1	2	700	10	400	140	-1	10	100
6	2178	0.82	0.55	0	1	2	700	10	400	140	-1	10	100
7	2006	0.87	0.55	0	1	2	1000	10	500	140	-1	10	100
8	2005	0.82	0.55	0	1	2	1000	10	500	140	-1	10	100
9	2054	0.82	0.55	0	1	2	1000	10	500	140	-1	10	100
10	2000	0.88	0.5	0	1	2	1000	10	500	140	-1	10	100
11	564	0.19	0.5	0	1	2	1000	10	500	140	-1	10	100
12	556	0.19	0.5	0	1	2	1000	10	500	140	-1	10	100
13	580	0.19	0.8	0	1	2	1000	10	500	140	-1	10	100
14	3060	27.81	0.55	0	1	2	600	15	350	140	-1	30	200
15	3049	23.05	0.5	0	1	2	600	15	350	140	-1	30	200
16	3047	29.82	0.55	0	1	2	600	15	350	140	-1	30	200
17	3043	31.44	0.7	1	1	2	600	15	250	140	-1	30	300
18	989	1.29	0.7	1	1	2	600	15	450	140	-1	10	100
19	904	2.38	0.5	0	0	2	600	15	350	140	-1	20	100
20	941	5.05	0.7	1	1	2	600	15	350	140	-1	20	100
21	931	12.20	0.5	0	0	2	600	15	350	140	-1	20	100
22	13356	3.64	0.8	1	0	2	600	15	450	140	-1	10	100
23	13341	11.43	0.8	1	0	2	600	3	350	140	-1	10	100
24	13333	11.20	0.7	1	0	2	600	5	350	140	-1	10	100
25	13335	7.59	0.7	1	0	2	600	5	350	140	-1	10	100
26	13324	7.51	0.7	1	0	2	600	5	350	140	-1	10	100
27	3340	11.07	0.7	1	0	2	600	5	350	140	-1	10	100
28	3297	16.72	0.7	1	0	2	600	5	350	140	-1	10	100
29	3402	15.74	0.7	1	0	2	600	5	350	140	-1	10	100
30	3412	15.39	0.7	1	0	2	600	5	350	140	-1	10	100

The significance of those parameters is as follows:

- α: a parameters used to stimulate randomness when generating solutions and combining them.
- *opC*: refers to the method of construction used when generating initial trial solutions.

- opL: refers to the method used to refine solutions, either using an iterative refinement or using a randomized refinement.
- k_{CPU}: refers to the depth of recursion for the CPU refinement method.
- n_{sols}: this refers to the number of generated trial solutions.
- d_H: refers to the minimum acceptable Hamming distance when using the random generation method based on Hamming distance.
- k_{GPU}: refers to the number of refining iterations on the GPU.
- δ_{min}: defines the tolerance for accepting a bit flip or not.
- $|RSET|$: refers to the size of the reference set.
- $|POP|$: refers to the size of the population set.

It is imperative to reiterate that within AlmoQUBO lib, the user has the ability to control these parameters via a text file, thereby enabling greater experimentation flexibility and greater freedom to explore diverse regions of the search space.

In Fig. 5.4, we present a comparison between the results obtained using our Scatter Search (SS) implementation and the results reported by Marti et al. [2].

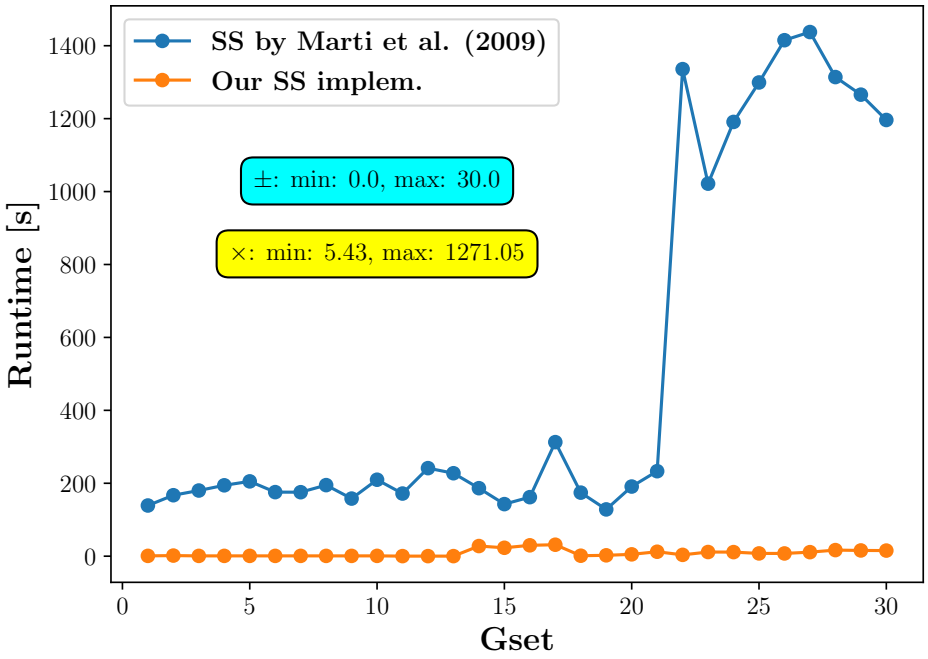


Figure 5.4: Comparison between results obtained my our SS implementation and the results obtained by the SS implementation in [2] (see also the data in the Appendix).

The minimum and maximum gain in solution quality are shown in the cyan-colored inset, and the maximum speedup is shown in the yellow-colored inset. In this experiment, the minimal gain achieved is +0, indicating that our implementation yields outcomes that are either identical to or superior to those reported in Ref. [2]. Furthermore, we achieved a considerable reduction in time-to-solution. The minimum speedup observed is 5.43×, and the maximum speedup reached 1271.05×. This wide range of speedups demonstrates the efficiency of our approach, particularly when handling larger problem instances. The fact that we consistently achieve speedups ranging from modest to remarkable highlights the scalability of our implementation across various instance sizes. In all instances, our SS outperforms the reference SS implementation in terms of solution quality and runtime, confirming the effectiveness of the optimizations and the improvements made in the implementation of AlmoQUBO library.

By comparing our SS approach with other existing algorithms, namely CirCut [3] and VNSPR [4]. Our SS method is still unbeatable and consistently offers the best results in terms of both high solution quality and reduced runtime.

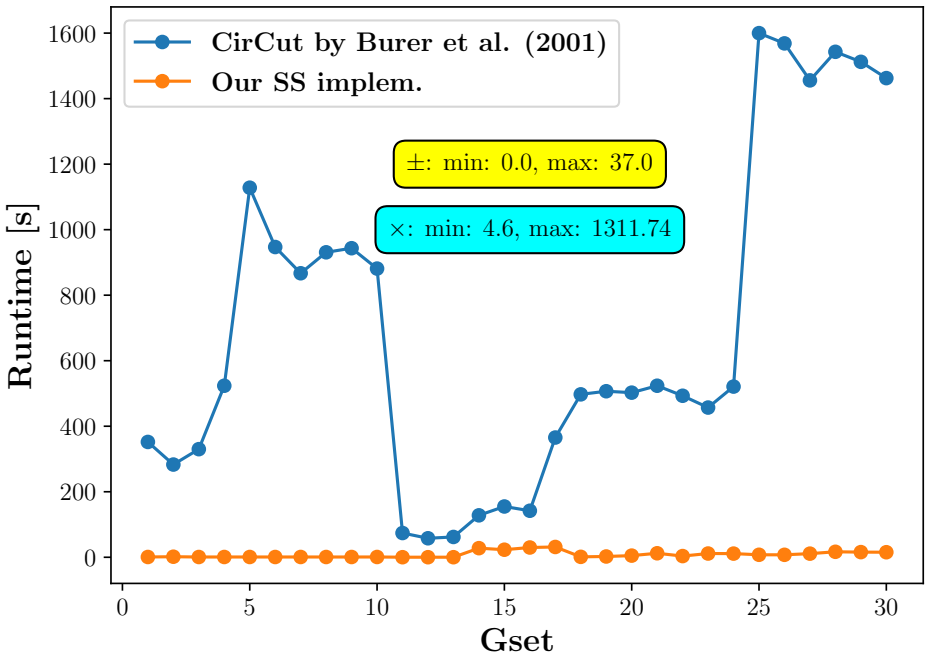


Figure 5.5: Comparison between results obtained my our SS implementation and the results obtained using CirCut [3] (see also the data in the Appendix).

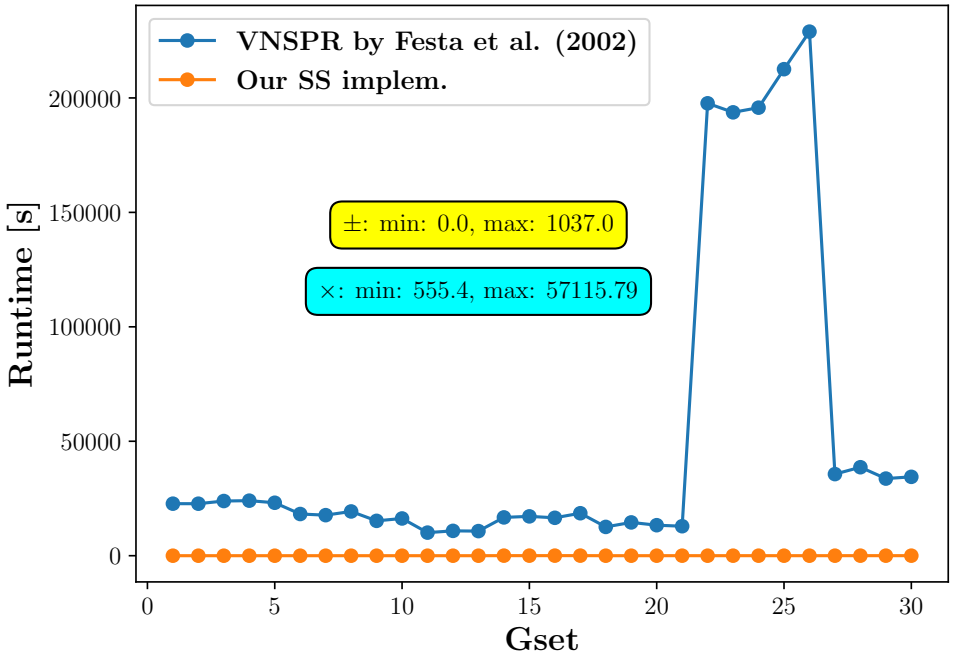


Figure 5.6: Comparison between results obtained my our SS implementation and the results obtained using VNSPR [4] (see also the data in the Appendix).

While our SS performs well for all problem instances, Circuit and VNSPR often require significantly more computation time to achieve comparable results, especially as the problem size increases. We can conclude that our algorithmic approach, in AlmoQUBO, is effective in achieving optimal or near optimal solutions in a highly reduced runtime. It is worth mentioning that one can still achieve better results with our SS by experimenting with other parameters, such as increasing the number of iterations.

5.5 SA & SS comparison

In this section, we perform a comparison of the results obtained using our implementations of Simulated Annealing and Scatter Search. In Fig. 5.7, we plot the runtime of our Simulated Annealing versus our Scatter Search. We observe from the results that, while the difference in runtime performance between the two algorithms is not substantial, the objective values obtained by SA consistently exhibit a slight edge over SS.

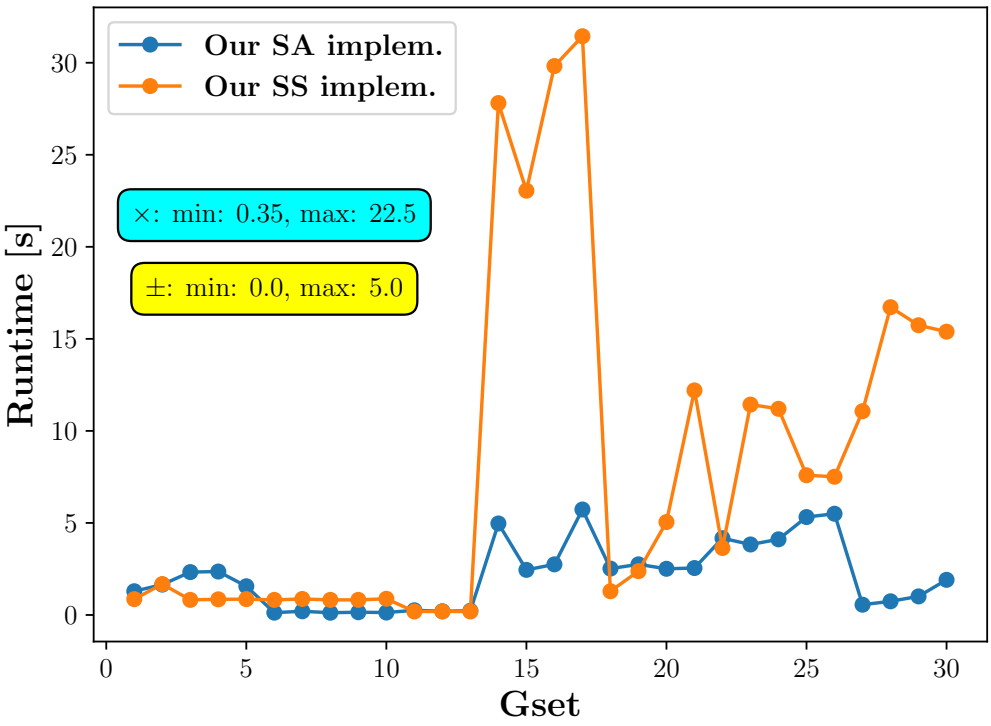


Figure 5.7: Comparison between the results obtained my our SA and SS implementations.

The SA algorithm achieves marginally higher solution quality in a shorter time frame for $\approx 99.99\%$ of the instances. This observation suggests that both algorithms are efficient in their own right, but SA holds a small advantage in balancing runtime and solution quality. Furthermore, it appears that the parallel seed strategy incorporated into our SA framework plays a pivotal role in achieving this enhanced performance without significantly increasing computational time. These improvements make SA a more adequate choice than SS.

Conclusion

In this work, we achieved significant milestones in constructing high-performance C++ implementations for QUBO optimization solvers for MaxCut problems, focusing on Simulated Annealing and Scatter Search. We proposed a heating-based parallel Simulated Annealing algorithm that was optimized for CPU execution, while Scatter Search was accelerated on GPUs using novel approaches. To achieve substantial performance gains, we employed techniques, such as edge list and edge offsets which reduced time complexity from $|V|^2$ to $2|E|$. Additionally, we developed few very efficient GPU algorithms for generating and refining a batch of trial solutions in a very competitive runtime.

We explored the potential of quantum annealing with D-Wave QPUs and investigated its strengths and limitations in solving QUBO problems. As a bigger accomplishment, we introduced AlmoQUBO, a versatile HPC-accelerated library supporting customizable solvers and input formats like Gset and upper triangular QUBO matrices. Finaly, we conducted benchmarks against the NP-hard MaxCut problem to demonstrate that our HPC solvers consistently outperformed reference implementations in both solution quality and time-to-solution. We observed, that using appropriate parameters (fine-tuned) proved crucial for optimizing performance, balancing runtime and the discovery of optimal or near-optimal solutions.

The future outlook includes adapting the heating-based Simulated Annealing algorithm for GPU implementation to leverage parallelism further. Expanding the AlmoQUBO library with additional solvers and advanced features to create a powerful tool for tackling a wide range of combinatorial optimization problems. The achievements of this work has the potential to pave the way for impactful applications in many areas that require solving QUBO problems.

Bibliography

[1] Tor GJ Myklebust. Solving maximum cut problems by simulated annealing. *arXiv preprint arXiv:1505.03068*, 2015. pages 4, 5, 7, 49, 51, 52, 54, 55, 69

[2] Rafael Martí, Abraham Duarte, and Manuel Laguna. Advanced scatter search for the max-cut problem. *INFORMS Journal on Computing*, 21(1):26–38, 2009. pages 4, 5, 7, 21, 22, 24, 28, 31, 49, 56, 58, 59, 70, 71

[3] Samuel Burer and Renato DC Monteiro. A projected gradient algorithm for solving the maxcut sdp relaxation. *Optimization methods and Software*, 15(3-4):175–200, 2001. pages 4, 49, 59

[4] Paola Festa, Panos M Pardalos, Mauricio GC Resende, and Celso C Ribeiro. Randomized heuristics for the max-cut problem. *Optimization methods and software*, 17(6):1033–1058, 2002. pages 4, 49, 59, 60

[5] Scott Kirkpatrick, C Daniel Gelatt Jr, and Mario P Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983. pages 7, 13, 32

[6] Sandip Sen. Simulated annealing approach to the max cut problem. In *Applications of Artificial Intelligence 1993: Knowledge-Based Systems in Aerospace and Industry*, volume 1963, pages 61–66. SPIE, 1993. pages 7

[7] Xueshi Dong, Qing Lin, Fanfan Shen, Qingteng Guo, and Qingshun Li. A novel hybrid simulated annealing algorithm for colored bottleneck traveling salesman problem. *Swarm and Evolutionary Computation*, 83:101406, 2023. pages 7

[8] Yan Zhang, Xiaoxia Han, Yingchao Dong, Jun Xie, Gang Xie, and Xinying Xu. A novel state transition simulated annealing algorithm for the multiple traveling salesmen problem. *The Journal of Supercomputing*, 77:11827–11852, 2021. pages 7

[9] Peter Grabusts, Jurijs Musatovs, and Vladimir Golenkov. The application of simulated annealing method for optimal route detection between objects. *Procedia Computer Science*, 149:95–101, 2019. pages 7

[10] Fred Glover. A template for scatter search and path relinking. In *European conference on artificial evolution*, pages 1–51. Springer, 1997. pages 7, 21

[11] Satoshi Morita and Hidetoshi Nishimori. Mathematical foundation of quantum annealing. *Journal of Mathematical Physics*, 49(12), 2008. pages 7

[12] Sheir Yarkoni, Elena Raponi, Thomas Bäck, and Sebastian Schmitt. Quantum annealing for industry applications: Introduction and review. *Reports on Progress in Physics*, 85(10):104001, 2022. pages 7

[13] Sergio Boixo, Tameem Albash, Federico M Spedalieri, Nicholas Chancellor, and Daniel A Lidar. Experimental signature of programmable quantum annealing. *Nature communications*, 4(1):2067, 2013. pages 7, 34, 35

[14] Kristen L Pudenz, Tameem Albash, and Daniel A Lidar. Error-corrected quantum annealing with hundreds of qubits. *Nature communications*, 5(1):3243, 2014. pages 7, 35

[15] OPTSICOM. Maxcut instances. `https://grafo.etsii.urjc.es/optsicom/maxcut.html`. Accessed: 2024-11-06. pages 8, 48

[16] Stanford University. Gset maxcut instances. `http://web.stanford.edu/~yyye/yyye/Gset/`. Accessed: 2024-11-06. pages 8, 12, 27, 48, 49

[17] Gyan Bhanot. The metropolis algorithm. *Reports on Progress in Physics*, 51(3):429, 1988. pages 13

[18] Daniel Delahaye, Supatcha Chaimatanan, and Marcel Mongeau. Simulated annealing: From basics to applications. *Handbook of metaheuristics*, pages 1–35, 2019. pages 14, 24

[19] Shuntaro Okada, Masayuki Ohzeki, Masayoshi Terabe, and Shinichiro Taguchi. Improving solutions by embedding larger subproblems in a d-wave quantum annealer. *Scientific reports*, 9(1):2098, 2019. pages 32

[20] Arnab Das and Bikas K Chakrabarti. Colloquium: Quantum annealing and analog quantum computation. *Reviews of Modern Physics*, 80(3):1061–1081, 2008. pages 32

[21] Richard P Feynman, Robert B Leighton, Matthew Sands, and Everett M Hafner. The feynman lectures on physics; vol. i. *American Journal of Physics*, 33(9):750–752, 1965. pages 33

[22] Giuseppe E Santoro, Roman Martonák, Erio Tosatti, and Roberto Car. Theory of quantum annealing of an ising spin glass. *Science*, 295(5564):2427–2430, 2002. pages 33

[23] Vasil S Denchev, Sergio Boixo, Sergei V Isakov, Nan Ding, Ryan Babbush, Vadim Smelyanskiy, John Martinis, and Hartmut Neven. What is the computational value of finite-range tunneling? *Physical Review X*, 6(3):031015, 2016. pages 33

[24] Demian A Battaglia, Giuseppe E Santoro, and Erio Tosatti. Optimization by quantum annealing: Lessons from hard satisfiability problems. *Physical Review E—Statistical, Nonlinear, and Soft Matter Physics*, 71(6):066707, 2005. pages 33

[25] Edward Farhi, Jeffrey Goldstone, Sam Gutmann, and Michael Sipser. Quantum computation by adiabatic evolution. *arXiv preprint quant-ph/0001106*, 2000. pages 35

[26] Edward Farhi, Jeffrey Goldstone, Sam Gutmann, Joshua Lapan, Andrew Lundgren, and Daniel Preda. A quantum adiabatic evolution algorithm applied to random instances of an np-complete problem. *Science*, 292(5516):472–475, 2001. pages 35

[27] Leonardo supercomputer. `https://leonardo-supercomputer.cineca.eu/`, 2024. Accessed: 2024-11-18. pages 46

[28] Thomas Hofmeister and Hanno Lefmann. A combinatorial design approach to maxcut. In *STACS 96: 13th Annual Symposium on Theoretical Aspects of Computer Science Grenoble, France, February 22–24, 1996 Proceedings 13*, pages 439–452. Springer, 1996. pages 47

[29] Lenka Zdeborová. Statistical physics of hard optimization problems. *arXiv preprint arXiv:0806.4112*, 2008. pages 47

[30] Stephan Held, Bernhard Korte, Dieter Rautenbach, and Jens Vygen. Combinatorial optimization in vlsi design. *Combinatorial Optimization*, pages 33–96, 2011. pages 47

[31] Gerard Cornuejols, Javier Peña, and Reha Tütüncü. *Optimization methods in finance*. Cambridge University Press, 2018. pages 47

[32] Maggie Xiaoyan Cheng, Yingshu Li, and Ding-Zhu Du. *Combinatorial optimization in communication networks*. Springer, 2006. pages 47

[33] Richard M Karp. *Reducibility among combinatorial problems*. Springer, 2010. pages 47

[34] David Steurer. Reduction from 3 sat to max cut. `https://www.cs.cornell.edu/courses/cs4820/2014sp/notes/reduction-maxcut.pdf`, 2014. CS4820 Course Notes, Cornell University, Spring 2014. pages 47

[35] Pál Erdős and Alfréd Rényi. On two problems of information theory. *A Magyar Tudományos Akadémia Matematikai Kutató Intézetének Közleményei*, 8(1-2):229–243, 1963. pages 49

[36] Kazimierz Kuratowski. *Introduction to set theory and topology*. Elsevier, 2014. pages 49

[37] D-Wave. Ocean software documentation. `https://dwave-meta-doc.readthedocs.io`. Accessed: 2024-11-06. pages 55

Appendices

MaxCut reference data

Table 6: Results obtained using D-Wave SA by performing 100 reads.

Gset	**ObjVal**	**Time**[s]
g1	11624	2.83
g2	11617	2.8
g3	11622	2.79
g4	11646	2.81
g5	11631	2.81
g6	2178	2.84
g7	2006	2.82
g8	2005	2.85
g9	2054	2.98
g10	1999	2.86
g11	562	1.46
g12	556	1.45
g13	580	1.45
g14	3057	1.97
g15	3043	1.98
g16	3044	1.96
g17	3036	1.98
g18	988	1.94
g19	904	1.93
g20	941	1.92
g21	927	1.94
g22	13357	5.31
g23	13338	5.24
g24	13332	5.27
g25	13329	5.25
g26	13321	5.17
g27	3333	5.19
g28	3295	5.12
g29	3402	5.25
g30	3412	5.13

Table 7: Results obtained using D-Wave SA by performing 1000 reads.

Gset	**ObjVal**	**Time**[s]
g1	11624	30.34
g2	11620	30.14
g3	11622	30.2
g4	11646	30.49
g5	11631	30.31
g6	2178	29.87
g7	2006	29.93
g8	2005	30.2
g9	2054	30.45
g10	2000	30.13
g11	564	16.67
g12	556	16.7
g13	582	16.66
g14	3059	21.71
g15	3046	21.84
g16	3049	21.6
g17	3043	21.73
g18	990	21.31
g19	906	21.39
g20	941	21.17
g21	931	21.31
g22	13358	57.25
g23	13343	57.13
g24	13335	57.74
g25	13338	58.19
g26	13326	58.36
g27	3341	57.31
g28	3298	58.3
g29	3404	57.61
g30	3413	58.18

Table 8: Simulated Annealing reference data [1].

Gset	ObjVal	Time[s]	G	ObjVal	Time[s]
g1	11621.0	228.34	g28	3298.0	214.312
g2	11612.0	294.68	g29	3394.0	251.784
g3	11618.0	327.224	g30	3412.0	214.44
g4	11644.0	295.132	g31	3309.0	214.628
g5	11628.0	294.28	g32	1410.0	214.044
g6	2178.0	299.572	g33	1376.0	193.964
g7	2006.0	246.516	g34	1382.0	194.092
g8	2005.0	204.756	g35	7485.0	194.116
g9	2054.0	206.232	g36	7473.0	262.772
g10	1999.0	205.5	g37	7484.0	265.46
g11	564.0	204.992	g38	7479.0	287.644
g12	554.0	188.928	g39	2405.0	263.84
g13	580.0	188.712	g40	2378.0	209.412
g14	3063.0	194.996	g41	2405.0	208.376
g15	3049.0	252.456	g42	2465.0	208.136
g16	3050.0	220.432	g43	6658.0	210.452
g17	3045.0	219.472	g44	6646.0	245.056
g18	990.0	219.464	g45	6652.0	241.36
g19	904.0	234.828	g46	6647.0	241.496
g20	941.0	196.02	g47	6652.0	244.896
g21	927.0	194.644	g48	6000.0	241.848
g22	13158.0	194.656	g49	6000.0	210.248
g23	13116.0	294.572	g50	5858.0	210.18
g24	13125.0	287.836	g51	3841.0	210.864
g25	13119.0	289.108	g52	3845.0	233.704
g26	13098.0	315.668	g53	3845.0	228.092
g27	3341.0	288.544	g54	3845.0	230.052

Table 9: Scatter Search reference data [2].

Gset	ObjVal	Time[s]	Gset	ObjVal	Time[s]
g1	11624.0	429.0	g28	3285.0	1437.5
g2	11620.0	139.0	g29	3389.0	1314.0
g3	11622.0	167.2	g30	3403.0	1265.8
g4	11646.0	180.1	g31	3288.0	1196.3
g5	11631.0	194.4	g32	1398.0	1336.2
g6	2165.0	205.2	g33	1362.0	900.6
g7	1982.0	175.7	g34	1364.0	925.6
g8	1986.0	175.5	g35	7668.0	950.2
g9	2040.0	194.9	g36	7660.0	1257.5
g10	1993.0	158.0	g37	7664.0	1391.9
g11	562.0	209.7	g38	7681.0	1386.8
g12	552.0	171.8	g39	2393.0	1011.5
g13	578.0	241.5	g40	2374.0	1310.9
g14	3060.0	227.5	g41	2386.0	1166.4
g15	3049.0	186.5	g42	2457.0	1016.5
g16	3045.0	142.8	g43	6656.0	1458.2
g17	3043.0	161.9	g44	6648.0	405.8
g18	988.0	312.8	g45	6642.0	355.9
g19	903.0	174.3	g46	6634.0	354.3
g20	941.0	128.3	g47	6649.0	498.1
g21	930.0	191.0	g48	6000.0	359.1
g22	13346.0	233.1	g49	6000.0	20.1
g23	13317.0	1335.8	g50	5880.0	35.1
g24	13303.0	1021.7	g51	3846.0	26.8
g25	13320.0	1191.0	g52	3849.0	513.0
g26	13294.0	1299.2	g53	3846.0	550.8
g27	3318.0	1415.1	g54	3846.0	423.8

Table 10: CirCut reference data [2].

Gset	ObjVal	Time[s]	Gset	ObjVal	Time[s]
g1	11624.0	523.6	g28	3260.0	1455.7
g2	11617.0	352.0	g29	3376.0	1542.9
g3	11622.0	283.0	g30	3385.0	1512.3
g4	11641.0	330.0	g31	3285.0	1462.6
g5	11627.0	523.6	g32	1390.0	1448.0
g6	2178.0	1128.1	g33	1360.0	221.0
g7	2003.0	947.0	g34	1368.0	198.0
g8	2003.0	866.9	g35	7670.0	237.0
g9	2048.0	930.9	g36	7660.0	440.0
g10	1994.0	943.4	g37	7666.0	400.0
g11	560.0	881.2	g38	7646.0	382.0
g12	552.0	74.0	g39	2395.0	1189.0
g13	574.0	58.0	g40	2387.0	852.4
g14	3058.0	62.0	g41	2398.0	900.6
g15	3049.0	128.0	g42	2469.0	941.6
g16	3045.0	155.0	g43	6656.0	875.4
g17	3037.0	142.0	g44	6643.0	213.0
g18	978.0	365.5	g45	6652.0	192.0
g19	888.0	497.3	g46	6645.0	210.0
g20	941.0	506.8	g47	6656.0	638.7
g21	931.0	502.5	g48	6000.0	632.9
g22	13346.0	523.6	g49	6000.0	119.0
g23	13317.0	493.0	g50	5880.0	134.0
g24	13314.0	457.0	g51	3837.0	231.0
g25	13326.0	521.0	g52	3833.0	497.3
g26	13314.0	1599.9	g53	3842.0	506.8
g27	3306.0	1568.6	g54	3842.0	502.5