Overview

We introduce qbsolv, a tool that solves large quadratic unconstrained binary optimization (QUBO) problems by partitioning into subproblems targeted for execution on a D-Wave system. Using a classical subproblem solver rather than quantum annealing, qbsolv delivers state-of-the-art numerical results and executes in two-thirds of the time of the best previously known implementation. We have released qbsolv as open-source software to foster greater use and experimentation in such partitioning solvers and to establish the QUBO form as a target for higher-level optimization interfaces.
Notice and Disclaimer

D-Wave Systems Inc. (“D-Wave”) reserves its intellectual property rights in and to this document, any documents referenced herein, and its proprietary technology, including copyright, trademark rights, industrial design rights, and patent rights. D-Wave trademarks used herein include D-WAVE®, D-WAVE 2X™, D-WAVE 2000Q™, and the D-Wave logo (the “D-Wave Marks”). Other marks used in this document are the property of their respective owners. D-Wave does not grant any license, assignment, or other grant of interest in or to the copyright of this document or any referenced documents, the D-Wave Marks, any other marks used in this document, or any other intellectual property rights used or referred to herein, except as D-Wave may expressly provide in a written agreement.
Summary

Quantum annealing systems implemented by D-Wave Systems offer a very different computing substrate from classical computers, which requires new programming tools to enable widespread use. The quadratic unconstrained binary optimization (QUBO) problem serves as a useful intermediate problem representation, as it closely matches the hardware format and many real-world problems can be easily mapped to it. However, QUBO instances arising from real-world problems do not necessarily match the constrained size and connectivity of a given system. We describe a solver, qbsolv, that reads a QUBO instance in a general format, partitions it into subQUBOs, solves the subQUBOs, and combines the results to form a solution to the original instance. Individual subQUBOs may be solved via a D-Wave system or a classical tabu search solver. Using the classical subQUBO solver, our solver finds competitive solutions in about two-thirds of the time of the best known alternatives. Solving the subQUBOs on a D-Wave system, our solver finds competitive solutions though with no speed advantage. These results demonstrate that large QUBOs can be effectively solved by a solver using quantum annealing hardware of limited size and connectivity. We expect this proof of concept to spark further work both in such solvers and in tools and techniques to map real-world problems to the QUBO format.
1 Introduction

Quantum annealing (QA) systems such as those developed by D-Wave Systems will provide one path beyond the anticipated end of Moore’s Law performance improvements for classical computers. However, the programming model implemented by QA is very different from that of the classical von Neumann architecture and requires a much closer mapping of problem to computer (at this stage of QA development). This difference requires application developers to map to new problem formulations, and they require that the time spent on such fundamental rethinking be amortized over multiple machine generations. Thus, generality of programming abstractions is essential.

The quadratic unconstrained binary optimization problem, or QUBO [1], is a close match to the native (Ising model) machine instruction of D-Wave systems. Many problems have been shown to map to the QUBO form [2, 3], so it is an attractive intermediate representation. However, in order to be solved directly on a D-Wave platform, a given input instance in general QUBO form must be partitioned and mapped to a restricted format known as the Chimera graph [4]. The work of Glover et al. [5] and extensions in Wang et al. [6, 7] show that decomposing large QUBO instances into subQUBOs and combining the sub-solutions can be an efficient and effective solution approach. This work describes a new algorithm motivated by Glover’s algorithm, with the subQUBO size selected to fit on a given D-Wave system, for eventual quantum acceleration. With the current implementation the user may select to execute subQUBOs either on a D-Wave system or on qbsolv’s internal classical tabu search solver. Both choices deliver solutions that are competitive with the best published results. Further, purely classical execution delivers execution times that are about two-thirds of the best published results.

While these results are encouraging, we view them more as validating the approach of creating a general QUBO solver suitable for a variety of problem types rather than this particular implementation. Independent of the exact algorithm used, large-neighborhood (subQUBO) updates can improve the performance of local search algorithms, and solving the subQUBOs is achievable with current QA hardware. As expressed by Booth et al. [8], the availability of general QUBO solvers (especially those accelerated by quantum computers) may spur the development of higher-level tools and methods that target the QUBO form as a preferred intermediate representation.

2 The Quadratic Unconstrained Binary Optimization Problem

The QUBO problem is defined, for $Q$ an $n \times n$ upper-triangular matrix of real weights and $x$ a vector of $n$ binary variables, as minimizing the function

$$f(x) = \sum_i Q_{ii}x_i + \sum_{i<j} Q_{ij}x_ix_j$$

or, more concisely,

$$\min_{x \in \{0,1\}^n} x^T Q x.$$
With trivial transformations, the QUBO form is equivalent to the unconstrained binary quadratic program (UBQP) [5], Ising model [9], and undirected graphical model [10] forms.

### 3 Algorithm and Implementation

The hybrid classical/quantum algorithm described in this work uses a two-level approach — the first level being the full problem QUBO instance and the second level being the sub-QUBO sized to fit on the underlying D-Wave system. The algorithm iterates through trials, with each trial consisting of a set of calls to a subQUBO solver (ideally the D-Wave system) for global minimization and a call to tabu search for local minimization. (For ease of application development without access to a quantum system, the subQUBO may be executed either on a D-Wave system or on an internal classical solver. We refer to this as “subQUBO execution.”) The hybrid nature of the algorithm exploits the complementary strengths of the D-Wave quantum-annealing system: the D-Wave quantum-annealing system is very good at exploring diverse regions of the state space but its limited precision constrains its ability to get to an exact minimum quickly [13, 14]. By contrast, the tabu search can quickly find an exact minimum within a neighborhood but sometimes struggles to escape from that neighborhood. Alternatively, the algorithm can be viewed as a large-neighborhood local search [15], with tabu improvements after every iteration. Similar approaches were studied by Rosenberg et al. [16] and Albash et al. [17], with many suggested variations and refinements.

The algorithm itself is documented in Algorithm 1. The key data structure maintained is the index vector, which orders the variables by decreasing impact on the value of the solution. The impact of a variable is the increase in objective-function value (i.e., less optimal) that occurs when the variable is negated in the current solution. (Assuming the current solution is a local minimum, the value will not decrease.) We expect variables of larger impact to be more strongly determined.

After finding an initial value (best_energy), solution (best_solution), and index (lines 8-10), the main loop (lines 13-33) repeats the two phases of the computation. The first phase splits the input QUBO instance into subQUBOs that fit on a given system (currently about 45 variables on a 1000-qubit D-Wave 2X system) that are then executed. The fraction constant denotes the fraction of the QUBO instance that is partitioned into subQUBOs; it is typically in a range between 0.05 and 0.15. Each subQUBO’s variables are contiguous in the index vector, partly as a vestige of Glover’s algorithm and partly to solve portions of the problem whose variables have roughly equal impact on the solution. To optimize over a set of variables $S$, we clamp the variables outside of $S$. That is, we consider the QUBO instance on variables $S$ that results from fixing the variables not in $S$ to their values in the current best solution. More precisely, if the current solution is $x^*$, then the new subQUBO on $S$ is

$$f_S(x_S) = \sum_{i \in S} (Q_{ii} + d_i)x_i + \sum_{i,j \in S, i < j} Q_{ij}x_ix_j$$

where $d_i$ is the contribution to the linear term in $x_i$ provided by the variables that are clamped:

$$d_i = \sum_{j \not\in S} (Q_{ij} + Q_{ji})x^*_j.$$
After the subQUBO is solved, the current solution is updated with the appropriate bits from the subQUBO solution vector. The goal is that the new candidate solution, built from the prior candidate solution and updates from the subQUBO results (line 19), will jump out of a local minimum. If none of the subQUBO updates result in a change, those bits of the solution are set to 0 or 1 randomly. This new solution is then input to a tabu search that will find a new local minimum. Every time a new best solution is found, the main loop’s passCount iteration counter is reset (lines 23-25); the algorithm will make up to numRepeats passes to discover a new best solution.

The D-Wave and tabu phases are complementary in numerical precision. D-Wave systems have limited precision and are affected by integrated control error; for these and other reasons (including the heuristic nature of QA), solutions to the subQUBOs returned may not be global or even local optima. However, the classical portion of the algorithm, including the main loop and the tabu search, is calculated with the same precision with which the original QUBO format is specified: IEEE double-precision (64-bit) floating-point values.

The current method of mapping the logical subQUBO to the physical topology, via complete-graph embedding, is inefficient in terms of the number of variables that can be embedded. See Section 5 for a more detailed discussion of this effect.

In this description, we focus on the important concepts in the algorithm. In practice, an optimized implementation of the classical portions of the algorithm is also important. We omit many details of that optimization in the interests of clarity of exposition; e.g., the variables that hold the expected change in objective-function value from flipping each bit are not shown.

This algorithm builds on the concept from Glover’s algorithm [11] of strongly determined variables, which are those that frequently receive particular values in the best solutions previously discovered. This is related to the backbone concept from the constraint-satisfaction world, which denotes the set of variables that are fixed in all satisfying solutions. The algorithm tries to identify strongly determined variables constructively, because the analogous set of optimal assignments is not known in advance. The algorithm executes a number of trials, each consisting of a phase of tabu search and a phase of fixing or freeing selected variables, selecting a successively smaller subQUBO of the most impactful variables, solving a coarsened problem, and projecting those results into the next larger problem. (See also [12] for related work.)

4 Performance

To measure the performance of this new algorithm, we have focused on the UBQPs collected by Beasley and residing in the ORlib repository [18, 19]. (A UBQP instance can be trivially transformed to a QUBO instance.) These problems range in size up to 2500 variables with significant density; we focus on the ten 2500-variable problems. The results from Wang et al. [6] are the best results previously published. Wang et al. obtained their results on a PC running Windows XP with a Pentium 2.83 GHz CPU and 8 GB RAM. The qbsolv results below were obtained executing on a single core of a MacBook Pro with an Intel Core™ i7 2.3 GHz CPU and 16 GB RAM.
Algorithm 1 Partitioning algorithm implemented by qbsolv

1: Input: QUBO instance
2: # best_energy is the lowest value found to date
3: # best_solution is the solution bit vector corresponding to the lowest value so far
4: # index is the indices of the bits in the solution, sorted from
5: # most to least impact on value
6: # Get initial estimate of minimum value and backbone
7: solution ← random 0/1 vector
8: (best_energy, best_solution) ← TabuSearch(QUBO, solution)
9: index ← OrderByImpact(QUBO, best_solution)
10: passCount ← 0
11: solution ← best_solution
12: while passCount < numRepeats do
13:   change ← false
14:   for i = 0; i < fraction ∗ Size(QUBO); i += subQUBOsize do
15:     # select subQUBO with other variables clamped
16:     sub_index ← i + subQUBOsize − 1
17:     subQUBO ← Clamp(QUBO, solution, index[sub_index])
18:     (sub_energy, sub_solution) ← DWaveSearch(subQUBO)
19:     # project onto full solution
20:     if (solution[sub_index] ≠ sub_solution) then
21:       solution[sub_index] ← sub_solution
22:       change ← true
23:   end if
24: end for
25: if not change then
26:   Randomize(solution[0 : i − 1])
27: end if
28: (energy, solution) ← TabuSearch(QUBO, solution)
29: if energy < best_energy then
30:   best_energy ← energy
31:   best_solution ← solution
32:   passCount ← 0
33: else
34:   passCount ++
35: end if
36: index ← OrderByImpact(QUBO, solution)
37: end while
38: Output: best_energy, best_solution
Table 1 compares the results from Wang et al. to the results produced by qbsolv. The columns contain the instance number, the best previous value of the objective function (which [6] finds for all instances), the time for the PR1 algorithm from [6] (since PR1 is on average faster than PR2) in seconds, the best qbsolv value of the objective function in non-target mode (see below), the gain (or loss, if negative) in value compared to [6], the time in seconds for qbsolv in non-target mode, the non-target-mode speed-up of qbsolv over PR1, the time in seconds for qbsolv in target mode, and the target-mode speed-up of qbsolv over PR1. The qbsolv timings reported in Table 1 include the time spent partitioning the problem into subQUBOs and the time spent solving those subQUBOs. The qbsolv version is the most recent one in the qbsolv Github repository on October 3, 2017, also released as part of qOp 2.5. In these results all subQUBO-solution time is spent executing purely classically by qbsolv’s built-in tabu solver. To simplify comparison of successive results, qbsolv was run with a fixed random seed for each execution, yielding deterministic results when executed classically. Thus the timings shown are for a single execution for each instance.

By default, the qbsolv algorithm continues looking for a better answer after it achieves a new best-to-date value of the objective function. The “non-target” times here include this time spent after the final answer is found. When the user knows the best possible value (analytically or by construction), qbsolv’s target option allows that value to be specified, causing execution to halt when that best answer is found and saving any time that would otherwise be spent looking for a better answer. Both non-target and target times are reported in Table 1, as it is not clear that Wang et al.’s algorithm can also save that time.

The qbsolv numerical results are the same as the best from the previous literature, though consuming 53% of the total wall-clock time for the non-target case and 17% of the wall-clock time for the target case.

Rosenberg et al. [16] report objective-function values and execution time for their partitioning solver, also intended to solve subQUBOs on quantum annealing hardware when that
execution delivers a performance advantage. That work also reports objective-function values and execution time, with the number of executed system-sized subQUBOs measured and wall-clock time calculated. That algorithm is implemented in Python while qbsolv is implemented in optimized C. With this implementation difference besides the calculated D-Wave execution time, it is not clear that the Rosenberg timings, which are more than 8 times slower than the qbsolv timings reported here (1731 seconds for all 10 Beasley 2500-variable problems total compared to 201 seconds), indicate anything about the relative merits of the two algorithms.

5 Discussion

A partitioning solver accepting a standard form of a widely relevant intermediate representation appears to be a useful implementation target for higher-level tools. The potential usefulness of qbsolv as a back end in such a tool stack has been demonstrated via a prototype constraint-satisfaction tool, which generates the QUBO format described in Appendix A and solves it via qbsolv. Connections to widely used optimization interfaces such as the General Algebraic Modeling System (GAMS) [20] or AMPL [21] appear attractive.

This algorithm can reasonably be applied to very large optimization problems. The largest problem we have attempted to solve with qbsolv is map-coloring the 3108 continental US counties, with a color-encoding that creates a QUBO instance of 12,432 variables. The qbsolv algorithm solves that problem in 202 seconds using the target option. We have anecdotal information from early users that they have solved problems of more than 30,000 variables on a D-Wave 2X system.

At many junctures, we chose one of a set of plausible approaches for implementation; many not-chosen approaches may yield better results than the current implementation. Some of the most important topics we see for experimentation are:

- **Methods for embedding subQUBOs.** The inner loop of the algorithm, which extracts subQUBOs and executes them, will often execute a subQUBO that has not executed previously. A mandatory step in executing such a subQUBO instance on a D-Wave system is mapping the variables and pairwise interactions of the logical problem to physical qubits and couplers. For real-world problems, this is often done via a heuristic embedding approach [22], but that can be unacceptably slow for qbsolv’s needs. Instead, at initialization time the algorithm embeds a complete graph of the maximum size supported by the targeted system, re-using that same embedding for each call to DWaveSearch. The simple mapping of subQUBO variables and influences to clique chains is very fast but resource-inefficient in that early anecdotal evidence is that many subQUBOs will be much more sparsely connected than a complete graph. Improved embedding schemes will enable the algorithm to use a bigger subQUBO size and also improve the performance of the D-Wave system. As discussed by Albash et al. [17] and Rosenberg et al. [16], the emergence of fast, resource-efficient techniques for mapping subproblems to QA hardware is essential for the success of an iterative solver.

- **Methods for partitioning the input QUBO instance.** qbsolv uses a backbone-based method inspired by Glover et al. [7], which may result in getting stuck at a local
minima despite the use of large-neighborhood moves. Other partitioning strategies may yield better results.

6 Open-Source: Use and Collaboration

With our primary goals being to enable application use of D-Wave systems and find better partitioning solvers that support acceleration via quantum annealers, we chose to release qbsolv via an open-source license to promote its use, serve as a viable, sustainable code base that will foster such further work, and enable ready comparisons between different algorithms implemented within it. The qbsolv project resides at http://github.com/dwavesystems/qbsolv. Besides the code itself, there is a man page and an example. Questions or problems may be communicated via the Github site.

The source code may be built and executed purely classically via the built-in tabu solver on OS X and Linux systems. Users wishing to execute on a quantum-processing unit (QPU) (or simulated) D-Wave system will require additional software; contact the authors or D-Wave for details.

We welcome collaborators wanting to experiment with better versions of the qbsolv algorithms. We will administer the project balancing the two goals of fostering diverse experimentation and delivering usable, robust software.

7 Conclusion

In this paper, we describe a new implementation of a large-neighborhood local search algorithm that combines tabu search with partitioning a large QUBO instance into pieces that fit on current quantum annealing systems. We show that the performance of qbsolv, without acceleration from quantum annealing, is competitive in numerical results and superior in wall-clock performance compared with other QUBO solvers. The input QUBO instance is expressed in a file format derived from the DIMACS CNF format [23], which could be used as a common format by all tools writing and reading QUBO instances. We offer qbsolv as a proof of the concept of a large-QUBO solver that executes subQUBOs using quantum annealing, expecting that improved algorithms will likely arise. With the emergence of robust large-QUBO solvers, application developers and upper-level tools can map problems of arbitrary size and connectivity to the QUBO form and be confident there will be effective means of solution.

8 Acknowledgements

Our D-Wave colleagues Mark Furtney and Denny Dahl helped develop the tools architecture in which an arbitrary QUBO abstraction and qbsolv play essential roles. Mark Furtney implemented the initial shell of the qbsolv program and the prototype tool for solving constraint-satisfaction problems. Fred Glover and Mark Lewis improved the comparison
in Section 3 between the algorithm in [7] and this algorithm.
A QUBO Input File Format

The input file format is modeled on the DIMACS Satisfiability File Format [23]. QUBO files, typically named with a .qubo file extension, are ASCII, line-based, and may contain four types of lines:

- Comments: denoted by a "c" in the first column.
- Program: denoted by a "p" in the first column, the single program line per file must be the first non-comment line in the file, and must contain the following space-separated fields in the following order:
  - p: the problem-line sentinel
  - qubo: file type; qubo is the only value currently supported
  - target: topology; currently supported values include 0 or unconstrained
  - maxDiagonals: the number of diagonal elements in the problem
  - nDiagonals: the number of nonzero diagonal elements in the problem
  - nElements: the number of nonzero off-diagonal elements in the problem
- Diagonal: the $Q_{ii}$ from Section 2. I.e., a triple consisting of the variable number in both the first and second positions and the variable’s weight, which can be any integer or float value, in the third position. Variable numbers, for both Diagonal and Element lines, must be between 0 and maxDiagonals-1 inclusive.
- Element: the $Q_{ij}$ from Section 2. I.e., a triple consisting of the distinct variable numbers of the two variables the element connects in the first and second positions and the connection’s strength, which can be any integer or float value (excluding zero), in the third position.

The number of Diagonal lines must equal the nDiagonal value in the program line and the number of Element lines must equal the nElements value. While conventionally all Diagonal lines appear before any Element lines, Diagonal and Element lines may be freely intermixed in the file.

Listing 1: Example QUBO File

```
c  start with a comment
2 p  qubo  0 4 4 6
3 c  diagonals follow
4  0 0 3.4
5  1 1 4.5
6  2 2 2.1
7  3 3 -2.4
8 c  elements follow
9  0 1 2.2
10  0 2 -3.4
11  1 2 4.5
12 c any comment you want
13  0 3 -3.2
14  1 3 4.5678
15  2 3 1
```
References


