Quantum Programming Interfaces for NP-complete problems

Lucas Saldyt

05-10-2017

Contents

1 Abstract

This report explores different programming interfaces for solving NP-complete problems. Different classical algorithms are compared to the interfaces for both the DWave quantum annealer and the IBM Quantum Experience Computer.

Because of the rich software toolchains surrounding the DWave, it is easy to describe problems for the annealer to solve. However, DWave has limitations in the size of problems it can run and how quickly it can run them. DWave (and quantum annealing in general), despite having a useable interface, still needs many improvements before it will be useful on real problems.

IBM does not allow the same level of abstraction as DWave, but has been proven asymptotically faster than classical computing. However, it is limited to five qubits, effectively making it limited to a five variable problem if the problem is run in full. If the interface for programming a Universal Gate Quantum Computer improves, IBM will be a strong option for solving NP-complete problems quickly.

While Quantum Computation has proved useful on fabricated problems, there is still a long time before it has a true impact on real world problem solving.

2 Introduction

The Knapsack problem and other NP-complete problems have immediate real-world applications. However, as the size of an NP-complete problem increases, it becomes very difficult to solve. Quantum Computation can allow NP-complete problems to be solved more quickly. Quantum Annealing may eventually allow for the same. This paper explores different interfaces in the hope of improving the availability of quantum problem solving, and demonstrat-

ing the ease of use of existing interfaces.

3 Methods

3.1 Classical Knapsack Implementations

Firstly, this paper shows classical implementations of the Knapsack problem, each written in Python. The purpose of this is to make it clear how classical interfaces look compared to quantum interfaces.

3.1.1 Naive Implementation

The simplest interface for solving the knapsack problem is a naive python implementation that iterates through every combination of items, and returns the combination with the highest value that satisfies the problem's constraints.

Listing 1: Naive Knapsack

```
from itertools import chain, combinations

def powerset(iterable):
    "powerset([1,2,3]) --> () (1,) (2,) (3,) (1,2) (1,3) (2,3)
        (1,2,3)"
    s = list(iterable)
    return chain.from_iterable(combinations(s, r) for r in range(len
        (s)+1))

def setvals(items):
    assert(len(items) > 0)
    setvals = (0,) * len(items[0])
    for item in items:
        setvals = tuple(a + b for a, b in zip(setvals, item))
    return setvals
```

```
def naive(items, capacities):
    selection = []
    bestval   = 0
    for comb in powerset(items):
        if len(comb) > 0:
            setcaps = setvals(comb)
            if bestval < setcaps[0]:
                for j, capacity in enumerate(capacities):
                    if setcaps[j + 1] > capacity:
                        break
                    if j == len(capacities) - 1: # All constraints
                        satisfied for this set
                        selection = comb
    choices = [items.index(choice) for choice in selection]
    return bestval, selection, choices
```

However, the effectiveness of this implementation is limited, and only useful for showing how the code scales with different solvers.

3.1.2 Dynamic Programming Implementation

Traditionally, the optimal way to solve the knapsack is through dynamic programming. However, the code necessary is more difficult to write from scratch:

Listing 2: Dynamic Programming

```
import functools


def memoize(obj):
    '''
    Decorator that caches function calls, allowing dynamic
        programming.
    If a function is called twice with the same arguments, the
        cached result is used.
```

```
    obj: function to be cached
    '''
    cache = obj.cache = {}
    @functools.wraps(obj)
    def memoizer(*args, **kwargs):
        key = str(args) + str(kwargs)
        if key not in cache:
            cache[key] = obj(*args, **kwargs)
        return cache[key]
    return memoizer

def dynamic_knapsack(items, outerConstraints):
    '''
    Solve the knapsack problem

    'items': a sequence of pairs '(value, weight, volume)', where '
        value' is
      a number and 'weight' is a non-negative integer.

    'outerConstraints': a list of numbers representing the maximum
        values
      for each respective constraint

    'return': a pair whose first element is the sum of values in the
         most
    valuable subsequence, and whose second element is the
        subsequence.
    '''

    @memoize
    def bestvalue(i, constraints):
        '''
        Return the value of the most valuable subsequence of the
            first i
        elements in items whose constraints are satisfied
```

```
        '''
        if i == 0: return 0
        value, *limiters = items[i - 1]
        tests = [l > v for l, v in zip(limiters, constraints)]
        if any(tests): # constraint checking
            return bestvalue(i - 1, constraints)
        else:          # maximizing
            modifications = [v - l for l, v in zip(limiters,
                constraints)]
            return max(bestvalue(i - 1, constraints),
                       bestvalue(i - 1, modifications) + value)

    k = outerConstraints
    result = []
    for i in range(len(items), 0, -1):
        if bestvalue(i, k) != bestvalue(i - 1, k):
            result.append(items[i - 1])
            k = [
                    c - items[i - 1][n + 1] for n, c in enumerate(k)
                    ]
    result.reverse()
    choices = [items.index(choice) for choice in result]
    return bestvalue(len(items), outerConstraints), result, choices
```

3.1.3 Fully Polynomial Approximation Scheme Implementation

However, a fully polynomial approximation scheme is more accurately comparable to the problem being run on the DWave. A FPTAS simplifies the problem given, solving it to within a margin of error. After simplification, FPTAS uses dynamic programming. For problems with few variables, the results of FPTAS are very similar to the results of dynamic programming, even if the margin of error is allowed to be high. Notice that the fptas solver calls the

dynamic programming solver (with a simpler version of the given problem). Even though its implementation appears to be short, it is actually much larger because it includes the dynamic programming code.

Listing 3: Fully Polynomial Time Approximation Scheme

```
from operator  import itemgetter

from .dynamic import dynamic_knapsack

def fptas(items, capacities, e=0.1):
    maxvalue = max(items, key=itemgetter(0))[0]
    k = e * (maxvalue / len(items))
    items = [(v/k, *rest) for v, *rest in items]
    return dynamic_knapsack(items, capacities)
```

3.1.4 Greedy hueristic solver

The current implementation of the greedy algorithm only works on single constraint problems, but is very fast:

Listing 4: Greedy Knapsack Algorithm

```
from operator  import mul
from functools import reduce

def greedy(items, capacities):
    '''
    Currently undefined on problems with more than two constraints
    (ie a problem with value, weight, and volume)
    '''
    rank   = lambda item : item[0] / max(1, (reduce(mul, item[1:],
        1)))
    sitems = sorted(items, key=rank, reverse=True)
```

```
selection = []
setcaps = (0,) * len(capacities)
breakOuter = False

for i, item in enumerate(sitems):
    if breakOuter:
        break
    for j, capacity in enumerate(capacities):
        if j != 0:
            if setcaps[j] + item[j] >= capacity:
                breakOuter=True
                break
        if j == len(capacities) - 1:
            setcaps = tuple(a + b for a, b in zip(setcaps, item)
                )

bestval = 0
if all(a <= b for a, b in zip(sitems[i][1:], capacities[1:])):
    if sitems[i][0] > bestval:
        selection = [sitems[i]]
value = sum(item[0] for item in sitems[:1])
if value > bestval:
    selection = sitems[:i]

choices = [items.index(choice) for choice in selection]
return bestval, selection, choices
```

3.2 Quantum Knapsack Implementations

While there are many ways of providing problems to the DWave quantum computer (Such as APIs for Python, C++, or Haskell, or an array of Domain Specific Languages), This paper investigates the highest-level implementations possible for the DWave.

3.2.1 DWave Knapsack Implementations : Verilog

The first way to solve an NP-complete problem on the DWave is to provide a Verilog implementation of a classical "oracle" circuit that checks if a solution is correct. This provides enough information to generate a Quantum Binary Optimization Problem, which the DWave can perform annealing on. Below is a Verilog circuit, its corresponding image , and its corresponding image after problem reduction (to boolean satisfiability).

Listing 5: Single Constraint Knapsack Problem

```
module single (A, B, C, D, E, valid);
    input A, B, C, D, E;
    output valid;

    wire [4:0] min_value = 5'd15;
    wire [4:0] max_weight = 5'd16;

    wire [4:0]  total_value =
        A * 5'd4
      + B * 5'd2
      + C * 5'd2
      + D * 5'd1
      + E * 5'd10;

    wire [4:0]  total_weight =
        A * 5'd12
      + B * 5'd1
      + C * 5'd2
      + D * 5'd1
      + E * 5'd4;
```

```
    assign valid = ((total_value >= min_value) && (total_weight <=
        max_weight));
    endmodule
```

To simplify the construction of the Verilog file, one can use additional software:

1. QA-Prolog, a compiler from Prolog to Verilog (shown below)
2. QNP, which includes a script that generates Verilog from a csv file describing the knapsack problem

3.2.2 DWave Knapsack Implementations : Prolog

Using the QA-Prolog software, Prolog can produce similar code to the Verilog shown above. The single constraint problem, written in Prolog:

Listing 6: Single Constraint Knapsack Problem

```
knapsack(A, B, C, D, E) :-
    A * 4 + B * 2 + C * 2 + D + E * 10 >= 15,
    A * 12 + B + C * 2 + D + E * 4 =< 16.
```

3.2.3 DWave Knapsack Implementations : CSV

A csv file, which also compiles to the Verilog shown previously:

names	min value 15	max weight 16
A	4	12
B	2	1
C	2	2
D	1	1
E	10	4

3.2.4 Theoretical IBM Quantum Experience Knapsack Implementations

Quantum Annealing is not the only alternate method of solving the knapsack problem. Instead, Universal Gate Quantum Computing offers a polynomial

speedup based on Grover's search.

However, the interface for using a Universal Gate Quantum Computer is much different than using a classical computer or the DWave. By comparison, the DWave interface may as well be classical, since, as this paper has showed, classical languages can be compiled to annealing problems runnable on the DWave.

The language of Universal Gate Quantum computers usually is Quantum Gates, which are expressed in qasm files. While a qasm file describing Grover's search is potentially short, it requires expert knowledge to write, as each line of code represents multiplying a vector of complex numbers by a matrix of complex numbers. Like with the DWave interface, the code will look different for each problem, as well as scaling with the problem itself. An advantage to the classical interface is that the code can stay the same even when the problem changes.

Note that this code shown solves a two variable version of the knapsack problem.

Listing 7: Grover's search on a two-variable problem

```
h q[1];
h q[2];
s q[2];
h q[2];
cx q[1], q[2];
h q[2];
s q[2];
h q[1];
h q[2];
x q[1];
x q[2];
h q[2];
cx q[1], q[2];
h q[2];
```

```
x q[1];
x q[2];
h q[1];
h q[2];
```

3.2.5 Alternate Gate Model Languages

In addition to qasm, QML or QCL can be used to program a gate model quantum computer. However, these languages do not offer much abstraction from the hardware, and are far less developed than qasm is.

4 Results

A description of the five variable knapsack problem run below:

```
Given the following items:
Name : [value, weight, volume]
A    : [4,     28,      27]
B    : [8,      8,      27]
C    : [1,     27,       4]
D    : [20,    18,       4]
E    : [10,    27,       1]
Choose a set, such that:
value is greater than or equal to 30
weight is less than or equal to 50
volume is less than or equal to 50
```

Below is output from the classical solvers on a five variable knapsack problem: (Run with 1,000,000 iterations)

```
lucas@qed:~/projects/lanl/qnp$ ./qnp solve csvs/var5_multi.csv 1000000
```

```
['csvs/var5_multi.csv', '1000000']
fptas : 120.80939865000255
(1000000 iterations, timing=time.perf_time())
{'E', 'D'}
value is satisfied (30)
weight is satisfied (45)
volume is satisfied (5)
```

```
naive : 107.68942248700478
(1000000 iterations, timing=time.perf_time())
{'E', 'D'}
value is satisfied (30)
weight is satisfied (45)
volume is satisfied (5)
```

```
dynamic_knapsack : 114.03447730200423
(1000000 iterations, timing=time.perf_time())
{'E', 'D'}
value is satisfied (30)
weight is satisfied (45)
volume is satisfied (5)
```

DWave's annealing successfully solves the 5 variable knapsack problem as well:

```
Submitting the problem to the DW2X solver.
```

```
[...]

Timing information:

    Measurement                    Value (us)
    ------------------------------ ----------
    total_real_time                    199315
    anneal_time_per_run                    20
    post_processing_overhead_time        4406
    qpu_sampling_time                  181780
    readout_time_per_run                  141
    qpu_delay_time_per_sample              21
    qpu_anneal_time_per_sample             20
    total_post_processing_time           4406
    qpu_programming_time                15160
    run_time_chip                      181780
    qpu_access_time                    199315
    qpu_readout_time_per_sample           141

Number of solutions found:

        75 total
        33 with no broken chains or broken pins
         1 at minimal energy
         1 excluding duplicate variable assignments
```

```
Solution #1 (energy = -26.00, tally = 56):

    Name                Spin  Boolean
    ------------------  ----  --------
    harder_multi.A        -1  False
    harder_multi.B        -1  False
    harder_multi.C        -1  False
    harder_multi.D        +1  True
    harder_multi.E        +1  True
    harder_multi.valid    +1  True

{'D', 'E'}
```

This report does not include a five variable problem being run on IBM's topology.

5 Discussion

Quantum Computers currently do not help solve non trivial real world problems in an advantageous way. DWave currently has a superior interface, but has not proven an asymptotic advantage. The IBM interface is less usable for layprogrammers, but the gate model has been theoretically proven faster than the classical model of computation.

In the future, IBM and Google plan on building 50-qubit gate model computers, which will hopefully be able to demonstrate experimental quantum speedups for the gate model. D-Wave also plans on extending its hardware, and will continue releasing more advanced Quantum Annealers. Hopefully, D-Wave can demonstrate an asymptotic advantage both theoretically and experimentally.

Because it has been theoretically proven, the Gate Model seems to likeliest to be successful in the future. However, it needs advanced methods for error correction, which need to be developed further for gate model quantum computation to become a reality. Also, it would be greatly improved with an additional layer of abstraction in the programming interface.

DWave needs to prove an asymptotic speedup, as well as developing its own fault-tolerance. Currently, there do not exist robust, realizable schemes for fault-tolerance in annealing systems. If these are developed, D-Wave also be a potential option for quantum computation in the future.

Until these improvements happen, however, Quantum Computing will remain nothing more than an interesting topic for discussion.

6 Appendix

References

[1] Arvind, V., Schuler, Rainer *The Quantum Query Complexity of 0-1 Knapsack and Associated Claw Problems* Archiv, Accessed 2017

[2] Dahl, E. D. *Programming With the D-Wave: Map Coloring Problem* D-Wave systems, 2013

[3] Jordan, Stephen *Quantum Algorithm Zoo* math.nist.gov/quantum/zoo, Accessed 2017

[4] Booth, Michael, Reinhardt, Steven P., Roy, Aidan *Partitioning Optimization Problems for Hybrid Classical/Quantum Execution* D-Wave systems, 2017

[5] Hen, Itay, Young, A. P. *Exponential Complexity of the Quantum Adiabatic Algorithm for certain Satisfiability Problems* Department of Physics, UC Santa Cruz, 2011

[6] Mandra, Salvatore, Guerreschi, Gian Giacomo, Aspuru-Guzik, Alan *Faster than Classical Algorithm for Dense Formulas of Exact Satisfiability and Occupation Problems* Harvard University, 2016

[7] Pakin, Scott *Performing Fully Parallel Constraint Logic Programming on a Quantum Annealer* Cambridge University Press, 2017

[8] Ines Dutra *Constraint Logic Programming: a short tutorial* University of Porto, 2010

[9] Grover, Lok K. *A fast quantum mechanical algorithm for database search* Proceedings of 28th Symp Theory of Computing, 1996

[10] John Wright *Lecture 4: Grover's Algorithm* Carnegie Mellon University, 2015

[11] Ambainis, Andris *Quantum Search Algorithms* Archiv, Accessed 2017