**QOSF Fall 2024 Application - Research Document**

**Ashwin Kaliyaperumal**

**Part 1 – Integer Linear Programming (ILP) to Quadradic Unconstrained Binary Optimization (QUBO) model:**

**Define the ILP formulation of the Bin Packing Problem.**

A close-up of a paper with writing

Description automatically generated

There are three main parts of the Integer Linear Programming (ILP) model of the Bin Packing Problem (BPP).

The objective function is the main goal of the bin packing problem and it attempts to minimize the number of bins used. N is the number of bins and j is the index of the current box. The binary variable Y\_j is et to 1 if the current box is being used (or in other words, if it has at least one item in it), and 0 otherwise.

The constraints allow us to have valid solutions to the bin packing problem. The first constraint is the assignment constraint. The goal of this constraint is to ensure that every item can only go in one box. You want to make sure every item has a home, and you want to make sure you’re not duplicating items. M is the maximum number of boxes. This is a helpful variable to have to limit the bounds of possible solutions and it is always set to the total number of items – this matches the worst-case scenario that each item needs its only one box. I is the index of the current item being iterated on. The binary variable X\_ij is set to one if item I is in box J, and is set to 0 otherwise.

The capacity constraint makes sure that all of the items in a box fits. More specifically, this constraint makes sure that the capacity of the box doesn’t get overfilled by the weights of all of the items inside. W\_i is the weight of item I. You can think of this as the number of units that a specific item takes. C is the capacity of the bins. In this problem, I assume that all of the bins have the same capacity.

**Transform the ILP into a QUBO model.**

A close up of a paper

Description automatically generated

To convert the ILP into a Quadratic Unconstrained Binary Optimization (QUBO) model, I needed to 1) rewrite the y\_j binary variable only in terms of x\_ij and 2) I needed to combine the objective function and constraints into one function.

I tried to make sure the purpose of each function and constraint in the ILP definition of the BPP was preserved when converting it over into a QUBO model. The ILP definition of the objective functions punishes overuse of boxes and, in turn, rewards the use of fewer boxes. I needed a way of approximating this goal with just x\_ij**.** I ended up approximating this goal by counting the number of items in each box over every boxThis would mean that if there weren’t any items in a box j, the function would add to 0, which would be a reward. Looking at it in another way, **i**f there were items in a box, the function would add to a number greater than 0, which would be a punishment. Just as a side note, the objective function should technically be squared, but a squared binary variable reduces down to just that binary variablesince(x\_ij)^2 = x\_ij

For the assignment constraint function, I just squared the ILP assignment constraint function to fit the quadratic part of the QUBO functionJust like the ILP definition, the QUBO assignment constraint punishes any configurations that allow for either 1) no items, since the squared terms leaves a positive 1, adding an extra cost to the cost functionor 2) the same item in multiple boxes. Again, just like the no item case, this will lead to a positive integer and would add additional cost to the function. The best possible scenario (in other words, the lowest cost) for this function would be a 0. This case would only happen if an item I was assigned to one, and only one, box. This constraint is checked with every item.

The capacity constraint function follows the assignment constraint in process. It punishes any configurations of items in a box j if it 1) doesn’t utilize box space efficiently or if 2) it exceeds box capacity. This constraint makes solutions strive to be as close to meeting capacity or meet box capacity exactly to reduce the cost incurred in the function.

The constants (alpha, beta, gamma) in front of each function are there for scaling reasons. This is to make sure that the constraints are being met using actual data (and not just variables).

**Test the QUBO with specific instances (size – small, medium, and big)**

I used a random number generator to generate random capacities for each object, with each object taking from 1 – the bin capacity.

Initially, the specifications for my small, medium, and large test cases were:

* Small:
  + Bin Capacity – 100 units
  + Number of items – 30 items
* Medium:
  + Bin Capacity – 150 units
  + Number of items – 300 items
* Large:
  + Bin Capacity – 200 units
  + Number of items – 3000 items

However, I needed to scale my instance sizes down to allow for IBM’s quantum computers to handle it. These are the specifications that I ended up testing with:

* Small:
  + Bin Capacity – 100 units
  + Number of items – 2 items
* Medium:
  + Bin Capacity – 150 units
  + Number of items – 10 items
* Large:
  + Bin Capacity – 200 units
  + Number of items – 30 items

**Part 2 – Create a Brute-Force solver for the QUBO problem:**

I used D’Wave’s ExactSolver() to produce exact solutions to the QUBO model of the Bin Packing problem.

To do so, however, I needed to convert the equation form of my QUBO model into a matrix. I did this by extracting the coefficients of the diagonal terms and the off-diagonal terms by expanding out the QUBO equation for a small data set (with 2 items and a maximum of 2 boxes), simplifying down to the diagonal/off-diagonal components, and extrapolating that equation for a set of n items with a maximum of n boxes. The coefficients of the diagonal/off-diagonal binary variable components are what get put in the matrix.

Assuming n items with a maximum of n boxes, there are a total of n \* n variables that the QUBO model needs to keep track off.

To translate the objective function into a matrix form, I just add alpha to all of the diagonals.

Translating the assignment constraint function into a matrix is slightly more involved. There are both diagonal and off-diagonal terms to account for. For each of the diagonals, I subtracted beta. For all of the off-diagonals, where the items were the same, I added 2\*beta.

For the capacity constraint, I added gamma \* w\_i \* (w\_i – 2C) to all of the diagonals. For all of the off-diagonals where the binary variables referred to the same bin, I added a 2w\_0\*w\_1 term, where 0 and 1 refer to the weight of the two items that were in that same bin.

**A screenshot of a math test

Description automatically generated**

The picture above is a visualization of the QUBO array on the large dataset.

I set the alpha, beta, and gamma scalars through experimentation. I ended up learning alpha and gamma equal to one. I made beta equal to the bin capacity squared. This struck a good balance between returning solutions that didn’t include any items and between including duplicate items in my solution.

**Part 3 - Use a quantum annealing simulator (D-Wave)**

After creating the QUBO array, it was simple to use D Wave’s quantum annealing simulator. I passed in the QUBO array (represented as a numpy array for efficiency and speed in code), turned it into a binary quadratic model, and optimized it using D Wave’s SimulatedAnnealingSampler().

**Part 4 – Use a Quantum Variational Approach to solve the QUBO:**

**Create multiple Ansatz for testing**

To use a quantum variational approach to solve the QUBO model of the BPP, I needed to first convert the QUBO into an Ising Hamiltonian. This would allow me to use established variational approaches and would allow me to run the QUBO on IBM’s superconducting quantum computers, instead of D’Wave’s quantum annealing computers.

To do this, I had to implement a substitution of the binary variable, that was either a 0 or a 1, into a spin variable, that is either a -1 or +1. This conversion was done by plugging in the formula x\_ij = (1 + s\_i) / 2. From there, I could substitute the z gate for the spin variable and I could resolve for all of the coupling operators (the coefficient of the s\_i \* s\_j terms) and the energy (the coefficient of the s\_i terms) operators.

**Build a function with input being the QUBO and Ansatz. Using a hybrid approach, solve the QUBO.**

I tested out multiple of Qiskit’s built-in ansatz, using their EfficientSU2 and RealAmplitude ansatz. I settled on using the EfficientSU2 ansatz – it was the most effective and the one that produced valid solutions (without any duplicate items) most often.

To test the ansatz, I started off with random parameters and optimized, using Qiskit’s Powell optimizer, to find the parameters that minimized the cost. I then ran the optimized ansatz one last time to get the optimal item configuration for the specific bin packing problem instance I was solving.

**Part 5 – Using the Quantum Approximate Optimization Algorithm (QAOA) to solve the QUBO:**

**Create from scratch a QAOA function.**

I used the Ising Hamiltonian state and coefficients from the quantum variational approach to help create the rotational gates for the cost and mixer operations. I started off with a circuit of item \* item qubits and set all of the qubits in superposition.

For the cost function, there were two parts to it. For the energy operators, I applied a single rotational z gate to that qubit, with the rotational equal to the coefficient of that energy operator \* gamma, where gamma is a parameter to optimize over. For the coupled terms, I added a controlled rotational z gate to the two qubits, where the rotation was the coefficient of both of the qubits multiplied by each other and by gamma.

For the mixer circuit, I added a rotational x gate to each qubit, with the rotational equal to 2 \* beta, where beta is another parameter to optimize over.

I tested my QAOA algorithm with two runs, or two cycles of the cost and mixer layers. To test the QAOA algorithm, I followed the same procedure with the quantum variational approach. I started with a set of random parameters for each of the gamma and beta parameters. I then optimized to find the most optimal parameters that minimized the cost, and ran the QAOA algorithm one last time with the optimal parameters to get a solution to the instance of the bin packing problem.

**Part 6 – Compare and analyze the results:**

**What is the difference between the QAOA, Quantum Annealing, and Quantum Variational approaches with different Ansatz? How do the results compare with the brute-force approach?**

Testing all of the different solvers and the brute-force approach with the small set, I got the same, exact answer (and the most optimal answer) from all of the solvers. They all ran relatively fast too. This isn’t too surprising – with just two items, there is only four variables that these algorithms needed to keep track of and finding all of the possible item-box permutations is trivial.

The problem, however, arose with the medium data-set. Theoretically, with 100 binary variables or qubits, Qiskit’s quantum computers should have been able to run both the variational algorithm and QAOA. However, in practice, while I was trying to optimize my parameters, my algorithm kept timing out – it was taking too long between runs. I would like to do more testing to figure out the exact reason why, but I believe having a dedicated session with the quantum computer, instead of running my job and waiting in the public queue to run it again, would help. I also couldn’t use a simulator because the simulator doesn’t support 100 qubits. The brute force solver didn’t work either. There were too many binary variables in QUBO model for the medium bin packing problem instance for the solver to brute force.

Just like with the medium data set, only quantum annealing was able to find a solution for the large data set. It got a valid solution of 21 boxes. While I don’t have a concrete way to prove that, for the large instance, 21 boxes is the minimum, I got the same solution after running the quantum annealing algorithm over 50 runs.

The main difference between all of the different solvers is size, specifically the size of the problem they can solve. Quantum annealing computers, while specialized to only solve problems that can be formatted as a QUBO model, can use their 1000’s of qubits to solve these models at scale. Quantum computers, like the ones that IBM makes, are more general – these can use a variety of gates to perform more complex algorithms. However, as of now, these computers are not at a scale that they can solve problems outside of case-studies.