Project report

Route Optimization using Quantum Annealing

Munawar Ali

supervised by
Dr. Abuzer YAKARYILMAZ

January 13, 2022
Route Optimization using Quantum Annealing

Quantum Annealing

Quantum annealing is a metaheuristic computing method for finding global minimum of a given objective function over a very large number of candidate solutions (local minima), by taking advantage of quantum mechanical properties like quantum superposition, tunneling, and entanglement. Quantum annealing computation outperform classical computing when it comes to solve NP hard optimization problems and probabilistic sampling problems. Physics can help to solve these sort of problems because we can frame them as energy minimization problem. The field has seen much progress of late and has been the subject of many research efforts. There have been many breakthroughs in the development of working quantum computers in recent years by companies such as IBM and D-wave Systems, the latter of which is a company known for its quantum annealing devices.

Quantum annealing, the type of quantum computation which is not capable of running the Shors algorithm and Grovers algorithm since the qubits used are not universal. It is not possible to apply the full range of gate operations on these qubits that would be necessary for a universal computer. However, quantum
annealing has great potential in the field of solving difficult optimization problems. The qubits can explore a vast energy landscape with many local minima quickly by utilizing quantum superposition, and thus find the global minimum of a complicated optimization problem.

Procedure for solving a problem on D-Waves quantum annealer

The basic unit of quantum computation is a quantum bit also known as qubit—analogous to the bit in classical computing. In D-wave QPU the qubits are the lower energy of the superconducting loops. The qubit is a quantum objects so it can be in state 0 or 1 and it can also be in the superposition state. In quantum annealing process initially all qbits are in superposition states as annealing process run, each qubit collapses from superposition state to the state 0 or in 1. The probability of falling state in 0 or 1 can be controlled by applying external magnetic field to the qubit. The programmable quantity which controls the external magnetic field is called bias. The real power of qubit comes when we link multiple qubit together so that one qubit influence the state of other qubit, the coupling of this qubits is done through a device known as coupler. In quantum annealing the programmable biases and coupling strength are the means by which problems are defined. The physics of this process can be visualized as a energy diagram shown bellow in digram 1. Initially qubit in superposition state correspond the state represented as one valley with single minima, after that annealing process runs and the barrier of the energy diagram raised and it turns into a double potential well with two minima, one valley correspond to the state 0 and one valley correspond to the state 1. The qubit will end up in one of these valleys at the end of the anneal. When we apply external magnetic filed to the qubit. This magnetic field tilts the well which increasing the probability of the qubit ending up in the lower state.
The coupling process links the qubit in such a way that the state of one qubit influence the state of other qubit here we basically exploiting the quantum entanglement phenomena. Quantum entanglement of two qubits can be thought as single object with four possible states.

At the end of the annealing process the minimum energy in this landscape correspond to the state (1, 1).

Another important phenomena which exploits the quantum tunneling feature. This feature corresponds to the instantaneous transition between the states. This means, transition between states does not require the qubit to climb the
barrier, they just pass through the barrier. This can be visualized as bellow in
the diagram 3.

Figure 3: A diagrammatic representation of quantum tunnelling. Credit: Cosmos Magazine

When we are formulating our problems we choose the value of biases and couplers which
defines the energy landscape, and quantum machines find the minimum energy of that
landscape, this is quantum annealing.

To solve problem with quantum annealing it is needed that the problem must be
formulated as discrete quadratic model so as it becomes amenable to quantum
annealing. The QPU is designed to solve quadratic unconstrained binary optimiz-
ation(QUBO) problems where each qubit represent the variable and coupler
between the qubit represent the cost associate between the qubit pairs. QUBO
is the problem of minimizing a quadratic polynomial over binary variables. It
can be represented as the following optimization problem:

\[ \text{Obj} = X^T Q X \]  

(1)

x is a vector containing N binary variables, and Q is an N×N matrix of real numbers describing the relationship between the variables. Given the matrix Q, finding binary variable assignments to minimize the objective function in this equation is equivalent to minimizing an Ising model, a well known problem in physics which is known to belong to NP.

**Problem Formulation**

My main motivation is to work on a multicommodity flow problem which is an optimization problem it lies in the NP-hard category and it is a linear flow problem. Linear multicommodity flow problems are linear programs (LPs) that can be characterized by a set of commodities and an underlying network. The objective is to flow the commodities through the network at minimum cost without exceeding arc capacities. It has many application in real world and this can be used to model and solve transportation, distribution and communication network problems. Mathematically we can represent this problem as bellow. Let \( G = (V, E) \) be a directed graph such that \( V \) is the set of nodes and \( E \) is the set of arcs. Each arc is assigned a capacity \( C_e \), an each arc has capacity \( d_{ij} \). A flow contains a binary decision variable where \( x_{ij}^k \) equals 1 if the entire quantity (denoted \( q^k \) ) of commodity \( k \) is assigned to arc \( ij \), and equals 0 otherwise. The cost of assigning commodity \( k \) in its entirety to arc \( ij \) equals \( q^k \) times the unit flow cost for arc \( ij \). Node \( i \) has supply of commodity \( k \), denoted \( b_i^k \), equal to 1 if \( i \) is the origin node for \( k \), equal to -1 if \( i \) is the destination node for \( k \), and equal to 0 otherwise.
minimize
\[
\sum_{\kappa \in K} \sum_{(i,j) \in A} c_{ij}^\kappa q^\kappa x_{ij}^\kappa
\]  

(2)
subjects to
\[
\sum_{(i,j) \in A} x_{ij}^k - \sum_{(j,i) \in A} x_{ji}^k = b_i^k \quad \forall i \in N \text{ and } \forall \kappa \in K
\]  

(3)
\[
\sum_{\kappa \in K} q^\kappa x_{ij}^\kappa \leq d_{ij} \quad \forall (i, j) \in A
\]  

(4)
\[
x_{ij}^\kappa \in \text{Integer and } \geq 0 \quad \forall (i, j) \in A \text{ and } \forall \kappa \in K
\]  

(5)

The first constraint represents that the flow must obey the capacity of each arc.
The second constraint ensures these three things: 1) the source vertex has one more edge leaving it than entering it; 2) the destination terminal has one more edge entering it than leaving it; 3) every other vertex has as many edges entering it as it has leaving it. And third constraint ensures that flow must be integer.

For QUBO formulation of the above equation we first convert the constraint term as a penalty
\[
H_s = \left( \sum_j x_{sj}^k - \sum_m x_{ms}^k - b_s^k \right)^2
\]  

(6)
\[
H_t = \left( \sum_j x_{tj}^k - \sum_m x_{mt}^k + b_t^k \right)^2
\]  

(7)
\[
H_i = \left( \sum_j x_{ij}^k - \sum_m x_{mi}^k \right)^2
\]  

(8)
above equations are the penalties which will be added to the original objective function. In this project I relaxed the capacity constraint (due to shortage of time) because it is an inequality term and to convert inequality as a penalty requires more detailed theoretical work. Adding all the penalty term together with main objective function our Hamilton looks like this
\[
H = \alpha_s H_s + \alpha_t H_t + \sum_{i \notin s,t} \alpha_i H_i + \alpha_c H_c
\]  

(9)
where \( H_c \) is our original objective function, so our objective is to minimize this Hamiltonian by choosing some suitable value of penalty constant (value of
alpha’s). I have construct this formulated quadratic unconstrained model into python with the help of D- Waves ocean Sdk. The Python program introduces a QM and submits it to the annealing solver to find the minimum energy value.

**Example**

I have taken an example of the above problem and minimize their energy to obtain the the required result. In this example I have generate a random graph of 10 nodes and 35 arcs with assigned costs to each arc, I have randomly chose the source and destination node and took one commodity and the demand of this commodity is 5, the problem is to find minimum cost at which we transport commodity from source point to the destination point. The result are shown bellow.

![Histogram of different energy](image)

This problem is numerated for 100 times, result obtained are probabilistic so this histogram of different energy shows that the probability of occurrence of each energy. 130 is a minimum energy and it is the our required result for this example.
This represent the energy occurrence for each numeration.

Our result can be represent as in this way, here variables which are active (equals to 1) are represents those arcs which are on the shortest path.
minimum energy: 130.0 The same result obtained from the D-wave quantum computer, and with the more times correct solution in 100 iteratins. More details comparison is not obtained although the code is given in Appendix(2) to study more in details.

0.1 Conclusions

Important real-world optimization problems can be formulated so as to be solvable using quantum annealing. QUBO provides a unifying framework for formulating such problems, we formulate our problem in Qubo formation and get the result as energy minimization problem by exploiting the quantum mechanical phenomena.
Appendices
Simulated Annealing Code

```python
from pprint import pprint
import matplotlib.pyplot as plt
import numpy as np
from pyqubo import *
import math
import random
import networkx as nx
from collections import Counter
import pandas as pd

G = nx.DiGraph()
T_commodities = 1  # Number of different types of commodities

#arcs = [(1, 0), (1, 2), (2, 0)]
arcs = [(0, 1), (0, 6), (0, 9), (1, 4), (1, 5), (1, 6), (1, 9), (6, 8), (9, 3), (9, 4), (8, 3), (8, 5), (8, 7), (3, 2), (3, 4), (3, 5), (3, 6), (4, 6), (4, 7), (4, 8), (5, 0), (5, 4), (5, 7), (5, 9), (7, 0), (7, 6), (2, 0), (2, 1), (2, 4), (2, 5), (2, 6), (2, 7), (2, 8), (2, 9)]

G.add_edges_from(i for i in arcs)

# Create random demand between node pairs
source_nodes = [[0], [0]]
dest_nodes = [[8], [9]]

# define arcs variables
x = [{(i, j): [Binary('x_{0}_{1}_{2}'.format(i, j, k)) for k in range(T_commodities)] for (i, j) in G.edges} for k in range(T_commodities)]
cost = [[math.ceil(10 * np.random.random()) for i in range(len(G.edges) + 1)] for k in range(T_commodities)]
```
#comm_dem = [math.ceil(10 * np.random.random()) for k in range(T_commodities)]

comm_dem = [5, 5]

#cost = [[15, 2, 1]]
cost = [[10, 2, 2, 12, 8, 5, 9, 7, 5, 9, 6, 3, 2, 6, 8, 3, 6, 46, 12, 13, 10, 10, 11, 2, 9, 21, 4, 10, 5, 7, 6, 6, 11], [10, 2, 2, 11, 2, 9, 10, 8, 9, 11, 5, 3, 4, 8, 5, 9, 3, 10, 5, 4, 9, 7, 3, 8, 12, 5, 3, 9, 5, 6, 3, 3, 8, 5]]

sum_cost_x = sum(np.matmul(list(sum(x[k][arc] for arc in G.edges)) for arc in G.edges), cost[k]) * comm_dem[k])

for k in range(T_commodities))

inn = [[[sum(x[k][(i, j)]) for (i, j) in G.in_edges(nodes)] for nodes in range(G.number_of_nodes())] for k in range(T_commodities)]

out = [[[sum(x[k][(i, j)]) for (i, j) in G.out_edges(nodes)] for nodes in range(G.number_of_nodes())] for k in range(T_commodities)]

# sum of out nodes - sum of in nodes = b_k... balanced constraint

def sum_out_in(T_commodities):
    # sum of intermediate nodes. or nodes without source or
destination points... sum((sum_out - sum_in) ** 2)

diff_k_out_in = []

for k in range(T_commodities):
    diff_out_in = []

    for nodes in range(G.number_of_nodes()):
        sum_in, sum_out = 0, 0

        if nodes not in list(set(dest_nodes[k])) and nodes not
        in list(set(source_nodes[k])):
            sum_in = sum(inn[k][nodes])

        if nodes not in list(set(source_nodes[k])) and nodes
        not in list(set(dest_nodes[k])):
            sum_out = sum(out[k][nodes])

        diff_out_in.append((sum_out - sum_in) ** 2)

    diff_k_out_in.append(sum(diff_out_in))
\[
\text{sum\_diff}_k = \sum(\text{diff\_k\_out\_in})
\]

# sum of supply and demand nodes, or nodes with source or
destination points. \( \sum((\text{sum\_out} - \text{sum\_in} - d)^2) \text{ or } \sum((\text{sum\_out} - \text{sum\_in} + d)^2) \)

\[
\text{diff\_k\_out\_in\_s}, \text{diff\_k\_out\_in\_d} = [], []
\]

for \( k \in \text{range}(T, \text{commodities}) \):
  \[
  \text{diff\_out\_in\_s}, \text{diff\_out\_in\_d} = [], []
  \]
  for \( \text{nodes} \) in \( \text{range}(G, \text{number\_of\_nodes}) \):
    \[
    \text{sum\_in}, \text{sum\_out} = 0, 0
    \]
    \[
    \text{sum\_in} = \sum(\text{inn}[k][\text{nodes}])
    \]
    \[
    \text{sum\_out} = \sum(\text{out}[k][\text{nodes}])
    \]
    if \( \text{nodes} \) in \( \text{source\_nodes}[k] \):
      \[
      \text{diff\_out\_in\_s}.\text{append}((\text{sum\_out} - \text{sum\_in} - 1)^2)
      \]
    if \( \text{nodes} \) in \( \text{dest\_nodes}[k] \):
      \[
      \text{diff\_out\_in\_d}.\text{append}((\text{sum\_out} - \text{sum\_in} + 1)^2)
      \]
  \[
  \text{diff\_k\_out\_in\_s}.\text{append}(\sum(\text{diff\_out\_in\_s})), \text{diff\_k\_out\_in\_d}
  \]
  \[
  .\text{append}(\sum(\text{diff\_out\_in\_d}))
  \]
\[
\text{sum\_diff\_k\_s} = \sum(\text{diff\_k\_out\_in\_s})
\]
\[
\text{sum\_diff\_k\_d} = \sum(\text{diff\_k\_out\_in\_d})
\]
return \( \text{sum\_diff\_k}, \text{sum\_diff\_k\_s}, \text{sum\_diff\_k\_d} \)

\[
\text{sum\_diff\_k}, \text{sum\_diff\_k\_s}, \text{sum\_diff\_k\_d} = \text{sum\_out\_in}(T, \text{commodities})
\]

print('start')

# Qubo formation

\[
\text{M} = \sum(\sum(\text{cost}[k]) \ast \text{comm\_dem}[k]) \text{ for } k \in \text{range}(T, \text{commodities})
\]

#penalty

\[
\text{H} = \text{sum\_cost\_x} + \text{M} \ast \text{sum\_diff\_k} + \text{M} \ast \text{sum\_diff\_k\_s} + \text{M} \ast \text{sum\_diff\_k\_d} \text{ # hamiltonian}
\]
model = H.compile()
qubo, offset = model.to_qubo()

#pprint(qubo)
bqm = model.to_bqm()
sa = neal.SimulatedAnnealingSampler()
sampleset = sa.sample(bqm, num_reads=100)
decoded_samples = model.decode_sampleset(sampleset)
best_sample = min(decoded_samples, key=lambda x: x.energy)
print(best_sample.energy)

model = dimod.BinaryQuadraticModel.from_qubo(qubo, offset=offset)

def plot.enumerate(results, title=None):
    plt.figure()

    energies = [datum.energy for datum in results.data(["energy"], sorted_by=None)]

    if results.vartype == 'Vartype.BINARY':
        samples = ["".join(c for c in str(datum.sample.values())).strip()
                        if c.isdigit()] for datum in results.data(["sample"], sorted_by=None)]
        plt.xlabel("bitstring for solution")
    else:
        samples = np.arange(len(energies))
        plt.xlabel("solution")

    plt.bar(samples, energies)
    plt.xticks(rotation=90)
    plt.ylabel("Energy")
    plt.title(str(title))
    print("minimum energy:", min(energies))
def plot_energies(results, title=None):
    energies = results.data_vectors['energy']
    occurrences = results.data_vectors['num_occurrences']
    counts = Counter(energies)
    total = sum(occurrences)
    counts = {}
    for index, energy in enumerate(energies):
        if energy in counts.keys():
            counts[energy] += occurrences[index]
        else:
            counts[energy] = occurrences[index]
    for key in counts:
        counts[key] /= total
    df = pd.DataFrame.from_dict(counts, orient='index').sort_index()
    df.plot(kind='bar', legend=None)
    plt.xlabel('Energy')
    plt.ylabel('Probabilities')
    plt.title(str(title))
    plt.show()
    print("minimum energy: ", min(energies))

simAnnSampler = neal.SimulatedAnnealingSampler()
simAnnSamples = simAnnSampler.sample(model, num_reads=100)
plot_enumerate(simAnnSamples, title='Simulated annealing in default parameters')
plot_energies(simAnnSamples, title='Simulated annealing in default parameters')

Listing 1: Simulated annealing code
Quantum annealing Code

```python
from pprint import pprint
import matplotlib.pyplot as plt
import numpy as np
from pyqubo import *
import math
import random
import networkx as nx
from collections import Counter
import pandas as pd
from dwave.system.samplers import DWaveSampler
from dwave.system.composites import EmbeddingComposite

sampler = EmbeddingComposite(DWaveSampler())

G = nx.DiGraph()
T_commodities = 1  # Number of different types of commodities

#arcs = [(1, 0), (1, 2), (2, 0)]
arcs = [(0, 1), (0, 6), (0, 9), (1, 4), (1, 5), (1, 6), (1, 9), (6, 8), (9, 3), (9, 4), (8, 3), (8, 5), (8, 7), (3, 2), (3, 4), (3, 5), (3, 6), (4, 6), (4, 7), (4, 8), (5, 0), (5, 4), (5, 7), (5, 9), (7, 0), (7, 6), (2, 0), (2, 1), (2, 4), (2, 5), (2, 6), (2, 7), (2, 8), (2, 9)]

G.add_edges_from(i for i in arcs)

# Create random demand between node pairs

source_nodes = [[0], [0]]
dest_nodes = [[8], [9]]
```
# define arcs variables

```python
x = {{(i, j): [Binary('x_{0,1,2}'.format(i, j, k))]
    for (i, j) in G.edges} for k in range(T_commodities)}
cost = {[math.ceil(10 * np.random.random()) for i in range(1, len(G.edges) + 1)] for k in range(T_commodities)}
#comm_dem = [math.ceil(10 * np.random.random()) for k in range(T_commodities)]
comm_dem = [5, 5]
#cost = [[15, 2, 1]]
cost = [[10, 2, 2, 12, 8, 5, 9, 7, 5, 9, 6, 3, 2, 6, 8, 3, 6, 46, 12, 13, 10, 10, 11, 2, 9, 21, 4, 10, 5, 7, 6, 6, 11], [10, 2, 2, 11, 2, 9, 10, 8, 9, 11, 5, 3, 4, 8, 5, 9, 3, 10, 5, 4, 9, 7, 3, 8, 12, 5, 3, 9, 5, 6, 3, 3, 8, 5]]
sum_cost_x = sum([np.matmul(list(sum(x[k][arc]) for arc in G.edges)), cost[k]) * comm_dem[k])
for k in range(T_commodities)]
inn = [[[sum(x[k][i, j]) for (i, j) in G.in_edges(nodes)] for nodes in range(G.number_of_nodes())] for k in range(T_commodities)]
out = [[[sum(x[k][i, j]) for (i, j) in G.out_edges(nodes)] for nodes in range(G.number_of_nodes())] for k in range(T_commodities)]
# sum of out nodes - sum of in nodes = b_k... balanced constraint

def sum_out_in(T_commodities):
    # sum of intermediate nodes, or nodes without source or
destination points... sum((sum_out - sum_in) ** 2)
diff_k_out_in = []
for k in range(T_commodities):
    diff_out_in = []
    for nodes in range(G.number_of_nodes()):
        sum_in, sum_out = 0, 0
        if nodes not in list(set(dest_nodes[k])) and nodes not in list(set(source_nodes[k])):
            ...
sum_in = sum(inn[k][nodes])
if nodes not in list(set(source_nodes[k])) and nodes not in list(set(dest_nodes[k])):
    sum_out = sum(out[k][nodes])
    diff_out_in.append((sum_out - sum_in) ** 2)
    diff_k_out_in.append(sum(diff_out_in))
    sum_diff_k = sum(diff_k_out_in)

# sum of supply and demand nodes. or nodes with source or destination points.. sum((sum_out - sum_in - d) ** 2) or sum((sum_out - sum_in + d) ** 2)
    diff_k_out_in_s, diff_k_out_in_d = [], []
for k in range(T_commodities):
    diff_out_in_s, diff_out_in_d = [], []
    for nodes in range(G.number_of_nodes):
        sum_in, sum_out = 0, 0
        sum_in = sum(inn[k][nodes])
        sum_out = sum(out[k][nodes])
        if nodes in source_nodes[k]:
            diff_out_in_s.append((sum_out - sum_in - 1) ** 2)
        if nodes in dest_nodes[k]:
            diff_out_in_d.append((sum_out - sum_in + 1) ** 2)
        diff_k_out_in_s.append(sum(diff_out_in_s)), diff_k_out_in_d.append(sum(diff_out_in_d))
    sum_diff_k_s = sum(diff_k_out_in_s)
    sum_diff_k_d = sum(diff_k_out_in_d)
return sum_diff_k, sum_diff_k_s, sum_diff_k_d

sum_diff_k, sum_diff_k_s, sum_diff_k_d = sum_out_in(T_commodities)

print('start ')

# Qubo formation
Listing 2: Quantum annealing Dwave sampler code