THE CLASSICAL BOIS

QUANTUM PROGRAMMING

Madhav Sankar Krishnakumar madhavsankar@ucla.edu

Parth Shettiwar parthshettiwar@g.ucla.edu **Rahul Kapur** rahulkapur@g.ucla.edu

February 28, 2022

Contents

ABSTRACT

We use Cirq to simulate two popular quantum algorithms (Shor's Algorithm and QAOA) on a classical computer. For each of the algorithms, we benchmark the performance of our simulations, measuring the scalability of compilation and execution with the number of qubits and dependence of the execution time on the the size of inputs. We then discuss our code organization and implementation of quantum circuits.

1 Design and Evaluation

1.1 Shor's algorithm

1.1.1 Low Level Design / Code Walkthrough:

Given a number n, we return the prime factorization of n. Shor's algorithm factors integers in polynomial time with high probability. The idea of Shor's algorithm is to guess a random integer that is less than the input and then check whether the guess leads to a factoring. The guess-and-check may well fail but it will succeed with probability greater than 1/2. So if we repeat the guess-and-check a few times, we are highly likely to succeed. The basic idea is given by:

```
Algorithm: Run Main Circuit
Input: A positive integer n >= 2.
Output: A prime factorization n = p1 ** k1 * ...  pk ** k_m.
Method:
while(1):
    (factor, power) = find_factor(n)

    if factor is None:
        break

    factors_with_power.append((factor, power))

    if (factor ** power) == n:
        answer_found = True
        break

    n = n // (factor ** power)

return factors_with_power
```

The process can be summarized as:

- First find a non-trivial factor of n. This code is explained below. Assume we do get back a non-trivial factor and the power of this factor in the prime factorization. If this return None, it means our algorithm failed.
- Add the factor to the prime factorization.
- If the value is this prime's power, then we have completed the prime factorization. Else continue factorizing $(n/factor^{power})$.

As can be seen, we have to find a non-trivial factor of n. Let us find how this is done:

```
Algorithm: Integer Factorization (find_factor).
Input: A positive integer n >= 2.
Output: A non-trivial factor of n, with its power -> (factor, power)
```

```
Method:
if (n is a prime) {
    return (n, 1)
}

if (n is even) {
    return (2, num_occurences(2))
}
else if (n = p ** k where p is prime and k >= 1) {
    return (p, k)
}
else {
    int d = Shor(N)
    return (d, num_occurences(d))
}
```

The process can be summarized as:

- First check if the number is prime, if yes just return it.
- First check if the number is even, then we know 2 is a non-trivial factor.
- Check if the given number is a power of prime, then we can solve it directly.
- If both the above cases do not satisfy, we try to find a non-trivial factor using Shor's Algorithm.
- We use a function $num\_occurrences(x)$ to find the power of x in the prime factorization of n.

Now let us look at the Shor's algorithm part of the above factorization code:

```
Algorithm: Shor's Algorithm.
Input: An odd, composite integer n that is not the power of a prime.
Output: A nontrivial factor of n, that is, an integer d such that 1 < d < N and d divides n.
Method:
repeat
    int a = random choice from {2, ..., n-1}
    int d = gcd(a,n)
    if (d > 1) {
        return d
    }
    else {
        int r = Find_Order(a, n)
        if (r is even) {
            int x = a ** (r // 2) - 1 (mod n)
            int d = gcd(x,n)
            if (d > 1) {
                return d
            }
        }
    }
until we give up
```

The process can be summarized as:

- First we take a random value from 2 to n-1 and if this happens to be a factor, then we are lucky and we can return it.
- Shor's algorithm calls the subroutine $Find\_Order$. The idea is that $Find\_Order(a, n)$ returns the smallest integer $r > 0$ such that a^r is 1(mod N). This number r is known as the order of a in Z_N^*.
- As n does not divide x $(x = a^{r/2} - 1(modn))$, we can hope that there might be some common factor between the two values that is non-trivial. So if the gcd between x and n is greater than 1, then we have found a non-trivial factor.

Now let us try to understand how $Find\_Order$ is implemented:

```
Algorithm: Order Finding.
Input: An integer n > 1 and an element x which belongs to Z*n.
Output: The smallest integer r > 0 such that a ** r = 1 (mod n).

Method:
if x < 2 or n <= x or math.gcd(x, n) > 1:
    raise ValueError()

circuit = create_orderfind_circuit(x, n)
measurement = cirq.sample(circuit)
return continued_fraction_algorithm(measurement, x, n)
```

The process can be summarized as:

- First ensure that x is an element of the multiplicative group modulo n.
- Now create the order finding circuit. This function computes an eigenvalue of the unitary function using Phase Estimation, which internally using Quantum Fourier Transform.
- Sample from the created circuit.
- Each run produces k/r, where we don't know either of k and r. This is where the Continued Fraction Algorithm comes in. The Continued Fraction Algorithm maps a list of samples of k/r to r. The Continued Fraction Algorithm runs in $\mathcal{O}((logn)^3)$ time, which is a major contributor to the time complexity of Shor's algorithm.

1.1.2 Code readability:

The whole code was divided into multiple functions. Have used the parameter of `verbose`, which prints all the intermediate steps of the program if set to True. The following are the main functions and their purpose:

- `class Modular_Exponential(target, exponent, base, n)`: This class defines modular exponential operation used in Shor's algorithm. This class represents the unitary which multiplies the target with the base raised to exponent and applies modulo operator over it. More precisely, it represents the unitary which computes:

```
if  0 <= target < modulus
ME|target>|exponent> = |target * (base ** exponent MOD modulus)> |exponent>
else
ME|target>|exponent> = |target> |exponent>
```

- `create_orderfind_circuit(x, n, verbose)`: This function returns quantum circuit which computes the order of x modulo n. This function computes an eigenvalue of the unitary using Phase Estimation, which internally using Quantum Fourier Transform.

- `continued_fraction_algorithm(result, x, n)`: The function interprets the output of the order finding circuit. Specifically, it determines k/r such that $e^{(2\pi ik/r)}$ is an eigenvalue of the unitary

```
U|y> = |xy mod n> if  0 <= y < n
U|y> = |y> if n <= y
```

  then computes r by continued fractions algorithms and returns it.

- `find_order(x, n, verbose)`: This function finds the smallest positive r such that $x^r mod n == 1$.

- `find_factor_of_prime_power(n)`: This function returns non-trivial factor of n if n is a prime power, else returns None.

- `num_occurrences(n, f)`: Returns the number of occurrences of the factor f of n, that is the power of the factor, f in the prime factorization of n.

- `find_factor(n, verbose, max_attempts)`: This function calls all the previous functions to find a non-trivial factor of n and returns this factor with its occurrence count.

- `runMainCircuit(n, verbose, max_attempts)`: This is the main function that runs the entire factorization algorithm. It repeatedly calls the $find\_factor$ function to get one factor at a time and creates the prime factorization of n.

1.1.3 Parametrizing the solution in n

We create the circuit using $create\_order find\_circuit$ which can take any given value of n as input. The number of bits in the representation of n can be assumed to be len. Then we need 'len' target qubits and '2 * len + 3' exponent qubits. Modular Exponentiation function is applied in a similar way as how Oracles were generated and all the other gates are applied to the exponent qubits, which are measured at the end.

1.1.4 Testing

The testing was done rigorously on the correctness of the solution by trying various values of n. The generated circuits were also printed. We tried running up to n of 30. For larger values of n, the memory and time requirement were too large to run multiple times and analyze (More values of n could be run once but not enough times to analyze the time).

1.1.5 Sample Circuit Generated for Find Order:

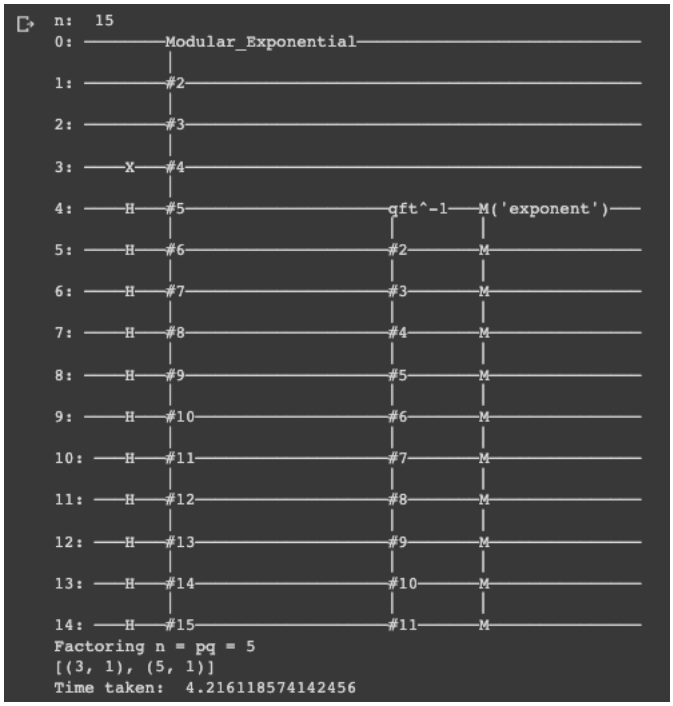

Figure 1: Find Order Circuit

1.1.6 Variation of Execution Time with n:

This is followed by plotting the amount of time taken to run the circuit. We ran this for various n and repeat the experiment for 5 times and take the average time.

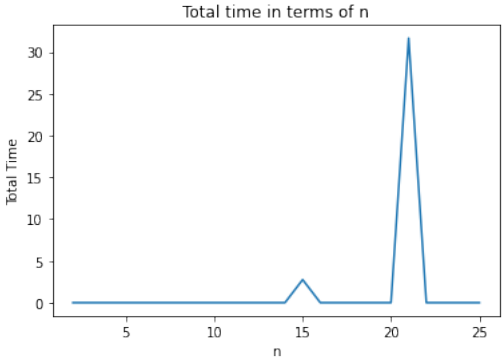

Figure 2: Execution Times

- We might observe that there are 2 peaks in the execution times at 15 and 21. This is because all the other numbers are either even or prime powers. Therefore the other number do not run the quantum find order code and as a result they are classical code paths. This explains their low execution times.
- In case of 15 and 21, they are not even or prime powers, therefore the find order generates the factorization through quantum fourier transform.

- We observe that the time for 21 is much higher than that of 15. Therefore the time will increase exponentially with n (when the number are not prime powers or even). This is due to the circuit generation and simulation times required for their execution.

1.1.7 Range of time taken for different runs of the same n:

In this experiment, we see the histograms for different runs for the same value of n = 15. We ran the code 100 times.

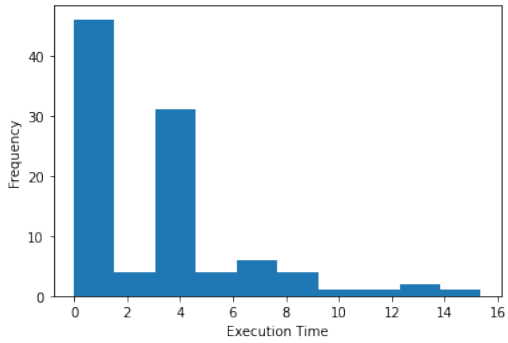

Figure 3: Histograms for Execution times for 100 iterations for n = 15

- First we observe that there is a huge range of time ranging from sub 1 seconds to close to 15 seconds. This is because the code uses a number of random numbers. So sometimes the factor is found using the random number itself, sometimes it is found using the Shor's algorithm in 1 run. In other times, it might have run for the complete 25 times as well! So there is a huge range of possibilities and this is reflected by the histogram.
- We observe over 40% of these times are sub 1, meaning that in most cases the solution was found without even going into the quantum find order.
- Around 35% instances took 1-5 seconds. This implies that in most cases the quantum find order found the solution in few iterations itself. Hardly few instances took a large number of iterations.
- We also observe from the code that there were 0 failure cases. So with 25 iterations, we had a 100% success rate.

1.1.8 Success Rate as a function of the maximum number of iterations:

In this experiment, wanted to check the success rates of the Algorithm by varying the maximum number of iterations for n = 15. We run each case 100 times and present the success rate as the percentage of successes.

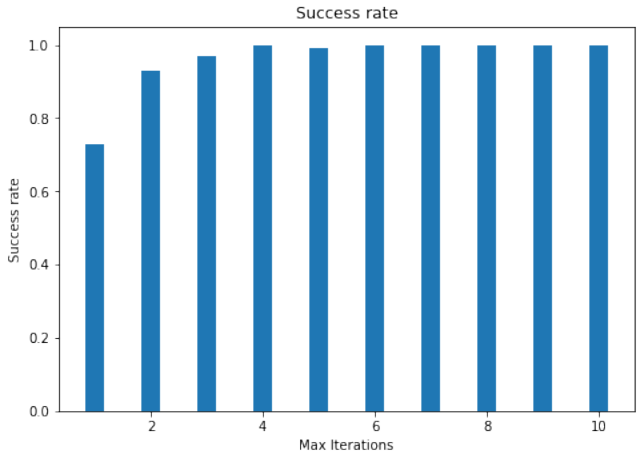

Figure 4: Success Rate as a function of maximum number of iterations.

- First we observe that the success rate increases as expected with increasing max iterations.
- It reaches an almost 100% success with 4 iterations maximum. This helps us understand that the default 25 iterations was not really needed for smaller n values.

1.1.9 Failure Rate as a function of n:

In this experiment, wanted to check the success/failure rates of the Algorithm by varying n for small max iteration values of 1 and 4. We chose small values as for larger values of max iteration success rate was almost always 100%. We run each case 20 times and present the failure rate as a percentage.

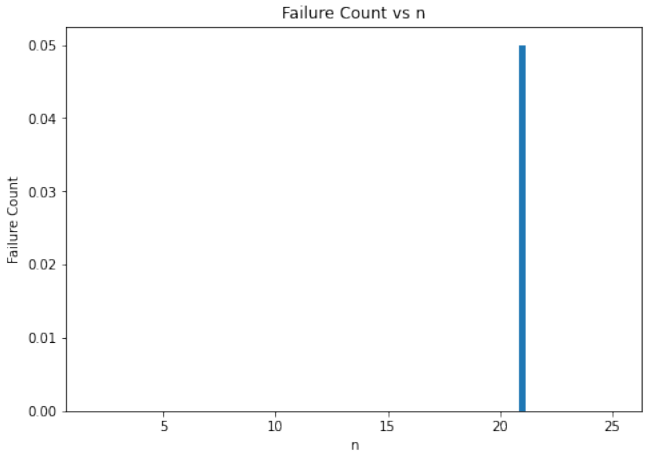

Figure 5: Failure Rate as a function of n for max iterations = 4.

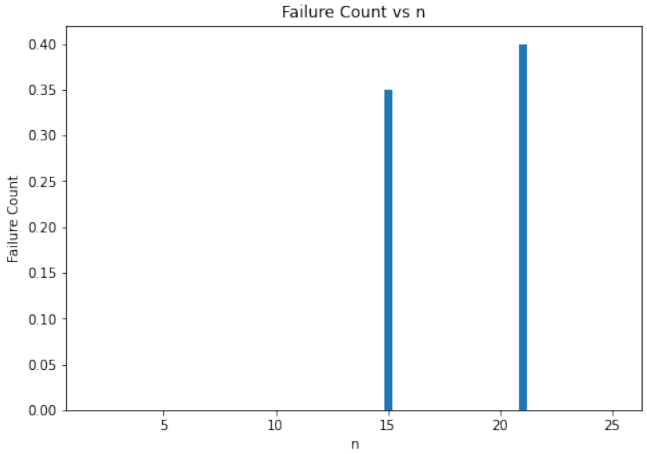

Figure 6: Failure Rate as a function of n for max iterations = 1.

- First, the success rate is close to 100% for almost all cases even for max iteration of 4. Only 21 shows failures and even that is close to 0. This is why max iteration on 1 becomes useful.
- We observe that the failure rate increases with n. We need to observe only n = 15 and n = 21 as in all other cases quantum circuits are not called. This is clearly visible with max iteration of 1.

1.2 QAOA Algorithm

1.2.1 Low Level Design / Code Walkthrough:

The objective of a Quantum Approximate Optimization Algorithm (QAOA) is to find approximate solution to a combinatorial optimization algorithm, with n variables and m clauses. [1]

More specifically,
Given fixed n, m, we take sufficiently varied number of $\gamma \in [0, 2\pi]$, $\beta \in [0, \pi]$, *such that when we measure:*
Mix(β*) Sep(*γ*)* $H^{\otimes n}$ $|0^n\rangle$ *we obtain a measurement z that maximizes count(z)*

Constraints:

- We have restricted to Max2Sat problem that comprises of all m clauses with 2 literals, each such literal can be any one of the n variables. Each clause can have repetition or negation of the same literals.
- **Problem statement**: Here n denotes the number of variables, m denotes the number of clauses and t which can be anything from 1,2,....m denotes the number provided by the user of how many clauses the algorithm must satisfy. The user asks if it does satisfy at least t clauses, then output 1, else output 0.
- Here, for current implementaion, p = 1, which denotes application of *Mix(*β*) Sep(*γ*)* $H^{\otimes n}$ in the above statement for 1 time. Theoretically, p can be varied to be sufficiently large to approximate the solution with higher probability.

The key to obtain the set of best solutions — by performing grid search of γ's and β's, is that for each such pair, QAOA satisfies as close to maximum clauses as possible — is dependent on designing apt implementation of the circuit.

Given a circuit with n qubits and 1 helper qubit, where each qubit denotes one variable, we divide our objectives in the following stages: For all different γ's and β's:

- i. Estabilishing Ground Truth using Classical Solver
- ii. Devising and Implementing Separator Circuit
- iii. Desigining Mixer circuit
- iv. Measure z and Evaluate Count(z). If $Count(z) \geq t$, then it's a good solution.

All the information can be summarized in the figure 7:

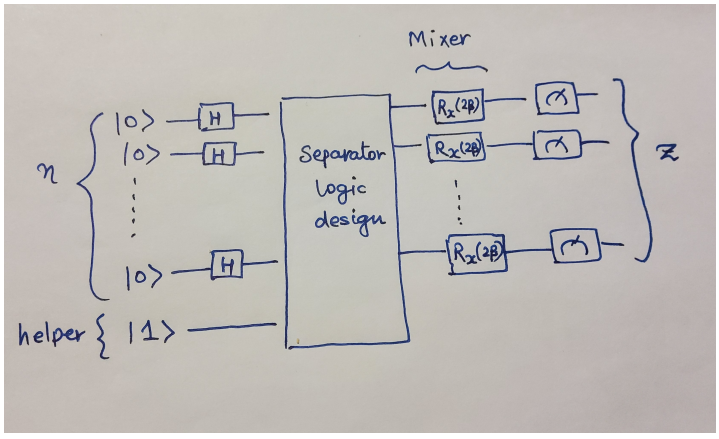

Figure 7: Overall Circuit logic

Each such objective is achieved using the following process:
i. Estabilishing Ground Truth

- **Black box function dependent on n and m:** We obtain a random list of m entries, each defining one clause. Further, each such clause is composed of two literals. The literals are denoted as 1,2,3,...n. In case of negation, we simply denote it as -1, -2,..etc. This gives us a very compact representation.

```
Algorithm: Black Box function for ground truth
Input: n, m.
Output: List of clauses
Method:
def getFx(m,n,verbose = True):
  lis = []
  coun = 0
  all_tups = []
  while(coun<m):
    s = np.random.choice(n, 2, replace=True)
    s = s + 1

    sig = np.random.randint(2, size = 2)*2-1
    s = sorted(s*sig)
    if(tuple(s) not in all_tups):
      all_tups.append(tuple(s))
      lis.append(list(s))
      coun += 1
  return lis
#Sample List:
#[[3, 3], [-4, 4], [-1, 4], [-3, 2], [-4, -4]]
```

For above sample list the underlying 2CNF is:

$$(x_2 \vee x_2) \wedge (\sim x_3 \vee x_3) \wedge (\sim x_0 \vee x_3) \wedge (\sim x_2 \vee x_1) \wedge (\sim x_3 \vee \sim x_3)$$

We have used 0-indexing, hence count starts from 0, and hence for every clause we see the subscript is one less than integer listed in sample list. The negative is used for integer in sample list for negating the literal in clause.

- **Creating classical solver for ground truth:** For checking accuracy and time running later, we need to equip black box function using a solver.

```
Algorithm: Classical Solver for ground truth
Input: List of clauses, n, m, t
Output: 1 if it satisfies at least t clauses, 0 otherwise
Method:

def solve(lis,n):
  dic = {}
  maxx = -1e10
  max_lis = []

  for i in range(2**n):
    val = bin(i)[2:].zfill(n)
    list_val = np.asarray(list(map(int,val)))*2-1
    summ = 0
    for j in range(len(lis)):
      summ += (np.sign(list_val[abs(lis[j][0])-1]) == np.sign(lis[j][0]))
              or
              (np.sign(list_val[abs(lis[j][1])-1]) == np.sign(lis[j][1]))

    dic[val] = summ
```

```
  for i in range(2**n):
    val = bin(i)[2:].zfill(n)
    # list_val = np.asarray(list(map(int,val)))*2-1
    if(dic[val]==max(dic.values())):
      max_lis.append(val)
  return max_lis,max(dic.values())
# Give the classical output: 1 or 0 : depends if at least t clauses are satisfied or not
def classical(lis,n,t):
  outs,outs2 = solve(lis,n)
  if(outs2>=t):
    return 1
  else:
    return 0
```

ii. Devising Mechanism for Separator Circuit

We need to repeat the below process for all of the m clauses:

- We first need three gates dependent on γ : R(γ) (1-qubit), CR($-\gamma$) (2-qubit), CCR($-\gamma$) (3-qubit)

  ```
  # Get the Three gates
  def gamma_matrix(size, gamma):
    matrix_R = np.identity(size, dtype=complex)
    matrix_R[-1][-1] = np.exp(-1j*gamma)
    return matrix_R

  def obtain_gamma_gates(gamma):
    gate1_rgamma = Oracle(1,gamma_matrix(2, gamma),"R")
    gate2_crgamma = Oracle(2, gamma_matrix(4, -1*gamma), "CR")
    gate3_ccrgamma = Oracle(3,gamma_matrix(8, -1*gamma),"CCR")
    return gate1_rgamma, gate2_crgamma, gate3_ccrgamma
  ```

  Here, Oracle is a custom gate implementation.

- We observe that each clause is a union (or disjunction) of two literals. And if for a particular assignment, the clause is satisfied, the separator will multiply the tensor product of qubits by $exp(-j * \gamma)$, where $j = \sqrt{-1}$

- Handling negation: Further, any literal can be negated. So we use a NOT gate for that particular qubit, use it for a clause (as indicated in previous step), then undo this process by using another NOT gate. This is shown in Figure 8

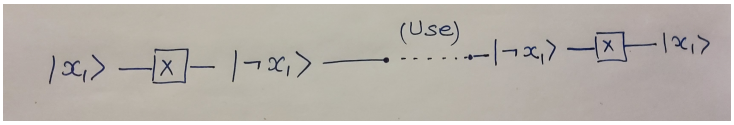

Figure 8: Handling negation

We handle these negation case as shown in below implementation:

```
#Handling negation and literal cases:
def separator_utility(circuit, first, second, qubits, gamma):
  # parsing
  line1 = np.abs(first) - 1
  line2 = np.abs(second) - 1
  gate1_rgamma, gate2_crgamma, gate3_ccrgamma = obtain_gamma_gates(gamma)
  # Negating the qubits
  if(first < 0):
```

```
    #meaning the literal is negated
    circuit.append([cirq.X.on(qubits[line1])])
  if(line2!=line1 and second<0):
    circuit.append([cirq.X.on(qubits[line2])])

  circuit = utility_helper(circuit, line1, line2, qubits, gate1_rgamma,
                           gate2_crgamma, gate3_ccrgamma)

  # Undoing negation of qubits
  if(first < 0):
    #meaning the literal is negated
    circuit.append([cirq.X.on(qubits[line1])])
  if(line2!=line1 and second<0):
    circuit.append([cirq.X.on(qubits[line2])])
  return circuit
```

- Depending on the two literals, our circuit can be one of the two types:
  - Both literals in one clause are same:

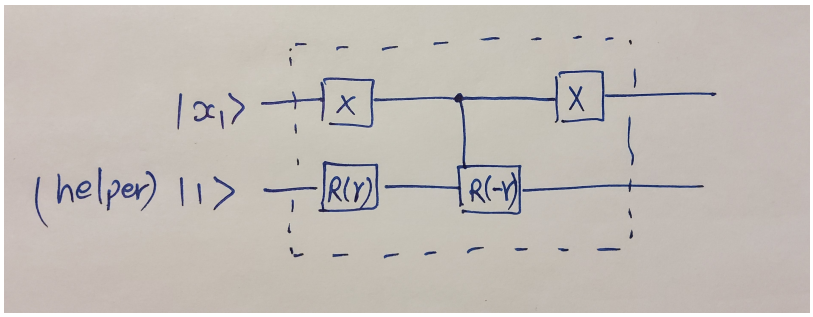

Figure 9: Both literals in one clause are same

  - Both literals in one clause are different:

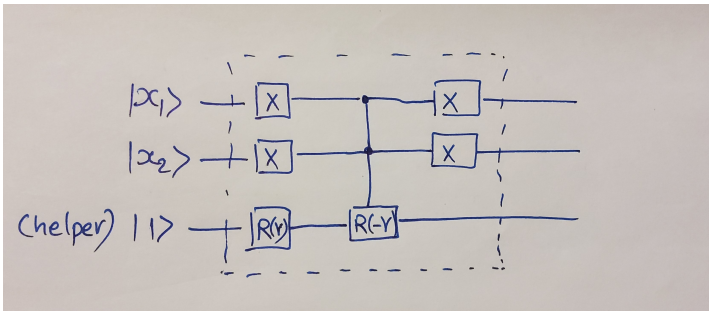

Figure 10: Both literals in one clause are different

We handle these two cases as shown in below implementation:

```
#Separator implementation
def utility_helper(circuit, line1, line2, qubits, gate1_rgamma,
                   gate2_crgamma, gate3_ccrgamma):
  if(line1 == line2):
    circuit.append([cirq.X.on(qubits[line1])])
    circuit.append([gate1_rgamma.on(qubits[-1])])
    circuit.append([gate2_crgamma.on(qubits[-1], qubits[line1])])
```

```
            circuit.append([cirq.X.on(qubits[line1])])
        else:
            circuit.append([cirq.X.on(qubits[line1])])
            circuit.append([cirq.X.on(qubits[line2])])
            circuit.append([gate1_rgamma.on(qubits[-1])])
            circuit.append([gate3_ccrgamma.on(qubits[-1], qubits[line1],qubits[line2])])
            circuit.append([cirq.X.on(qubits[line1])])
            circuit.append([cirq.X.on(qubits[line2])])
        return circuit
```

iii. Design *Mix(β)*:
This is a relatively straight forward implementation, as it is dependent on number of qubits, or n, and involves applying Rotation gate to all the qubits (except helper qubit).

```
#Mixer implementation
def createMixer(n,beta):
    R = np.array([[np.cos(beta),-1j*np.sin(beta)],[-1j*np.sin(beta),np.cos(beta)]])
    return R

for b1 in range(0,n):
    temp_c.append([u_beta_o[b1].on(qubits[b1])])
```

iv. Measure z and determining count(z): The bitstring generated at the end will be used to evaluate Count(z). If count(z) $\geq$ t, we output 1, else we output 0. Further, we aim to obtain a reliably better solution by varying the values in (γ, β) pair.

We expect that Separator circuit creation time depends on the no. of clauses it needs to process, as for each such clause, it needs to use either 1- , 2- or 3-qubit gates, which is otherwise constant, independent of n.

This is further illustrated in the experiments and analysis that we have added in the following sections.

1.2.2 Code readability:

We have modularized our code and used the name of variables and functions, following the conventions of the problem statement or the names are self indicated of the utility they perform. "Verbose", a parameter when passed as true by the user would generate detailed description of the main circuit An exhaustive description of the functions used for implementation is enlisted below for reference:

- `getFx(m,n,verbose = True)`: Here m, n denote m clauses and n variables. This would output a list (or the boolean formula representation), containing m entries each representing one clause. Each such clause would be having 2 values, out of 1,2,....n, -1,-2,...-n, which denote that particular qubit/variable is involved in the clause. The minus sign denotes negation of a particular variable in the literal.
- `solve(lis,n)`: The classical solver that takes input the boolean formula and exhaustively search for the solution across all possible variables, denoted as n. It returns the list of possible solutions and count.
- `classical(lis,n,t)`: This takes the lis and gives the clear output (analogous to QAOA quantum solver) in classical domain dependent on t. Here t can be from 1,2,....m.
- `createMixer(n,beta)`: This function takes β and n and returns the Rotation - β matrix which is then used as a gate, applied on all the qubits.
- `class Oracle(cirq.Gate)`: Custom gate implementation for defining any oracle with unitary matrix as input and number of qubits specified.
- `gamma_matrix(size, gamma)`: Returns the rotation gamma matrix for a given gamma and of a given size

- `obtain_gamma_gates(gamma)`: It generates 3 different gates: R(γ), CR($-\gamma$), CCR(-γ) and returns these gates as output depending on the value of γ.
- `utility_helper(circuit, line1, line2, qubits, gate1_rgamma, gate2_crgamma, gate3_ccrgamma)`: line1 and line2 denotes the indices of particular variable/qubit involved in a particular clause; circuit is the equivalent circuit; qubits denote linequbits passed from maincircuit;
  `gate1_rgamma, gate2_crgamma, gate3_ccrgamma` are all gates passed as input. It applies the separator logic and returns the circuit.
- `separator_utility(circuit, first, second, qubits, gamma)`: first and second denote the list entries (not indices) ranging from 1,2,...n,-1,-2,...-n. qubits are line qubits. It handles the separator logic of negation of a literal and calls `utility_helper(circuit, line1, line2, qubits, gate1_rgamma, gate2_crgamma, gate3_ccrgamma)` to perform the case by case job.
- `runMainCircuit(n,m,t,lis,beta_divs = 5,gamma_divs = 5,repeats=1, verbose = True)`:
  Provide the number of variables:n
  Provide the number of clauses:m
  Provide the number of clauses to be satisfied:t
  Provide the boolean formula in the format:lis (or use `getFx(n,m)` to obtain the randomized lis
  Provide the number of beta divisions the user wants to investigate.
  Provide the number of gamma divisions the user wants to investigate.

All the subsequent case-wise analyses are appended in the later sections.

1.2.3 Parametrizing the solution in n

We have successfully parameterized the solution in terms of n: as the number of qubits that we have are dependent on n (alongwith additional helper qubit). Moreover, the gates that we have added include X, H, R, CR, CCR. The number of gates in the circuit are directly a function of m. In comparison to classical approach, which uses all 2^n possible solutions for satisfying the Max2Sat problem, the implementation of QAOA is linear in the sense that we ran it for O(n*m) for given number of γ-divisions and β-divisions.

Moreover, we didn't use (nxn) custom gate - matrix implementation for separator logic circuit. Instead we devised optimized approach for implementing separator logic through use of 1-qubit, 2-qubit and 3-qubit gates, as entailed in previous sections.

Next section illustrates the example of how we parametrized in greater detail.

1.2.4 Sample Circuit Generated for a particular boolean formula:

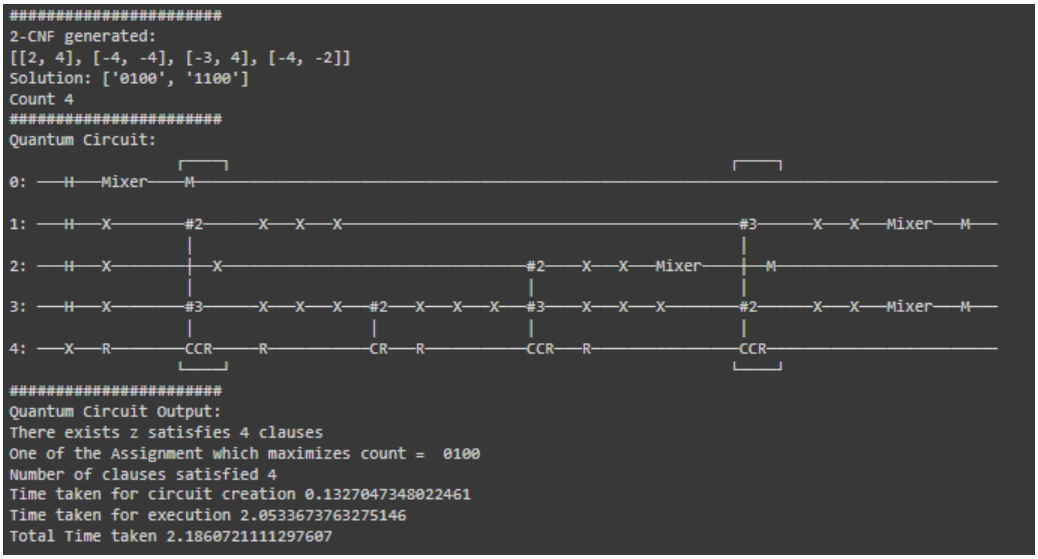

Figure 11: Sample Circuit (A)

As shown in the Figure 11,

2-CNF denotes boolean formula

$[[2, 4], [-4, -4], [-3, 4], [-4, -2]]$

is same as $(x_1 \vee x_3) \wedge (\neg x_3 \vee \neg x_3) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_3 \vee \neg x_1)$

solution denotes the classical solution

count denotes the possible no. of maximum satisfied clauses

t=4 is equal to this count

Hence, we do have solution both from quantum and from classical, which matches the ground truth

Output: 1

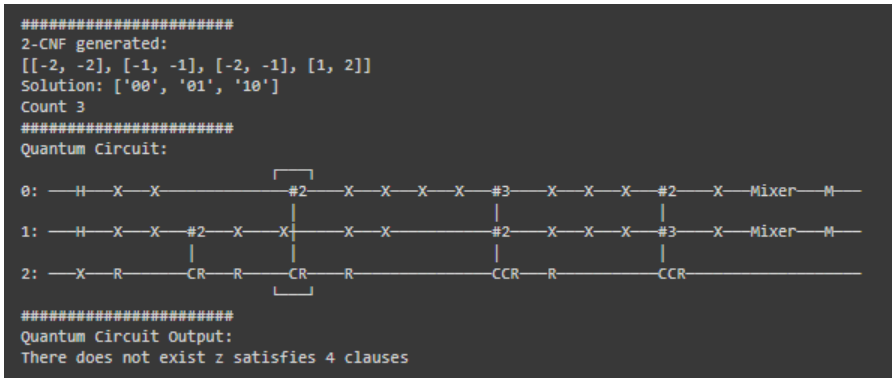

Figure 12: Sample Circuit (B)

As shown in the Figure 12,

2-CNF denotes boolean formula

$[[-2,-2],[-1,-1],[-2,-1],[1,2]]$

is same as $(\neg x_1 \vee \neg x_1) \wedge (\neg x_0 \vee \neg x_0) \wedge (\neg x_1 \vee \neg x_0) \wedge (x_0 \vee x_1)$

solution denotes the classical solution

count denotes the possible no. of satisfied clauses

t=4 is greater than this count

Hence, we don't have solution neither from quantum nor from classical, which matches the ground truth

Output: 0

1.2.5 Histogram of assignment values for variables

Firstly, we conducted an experiment on single circuit where we tried to run the circuit on particular β and γ for 1000 iterations. We did this for whole search space and analaysed the histogram of variables assignments outputs from circuit for the best β and γ which has the highest count of outputs which satisfy more than equal to *t* clauses (as described in problem statement). Following was the histogram obtained:

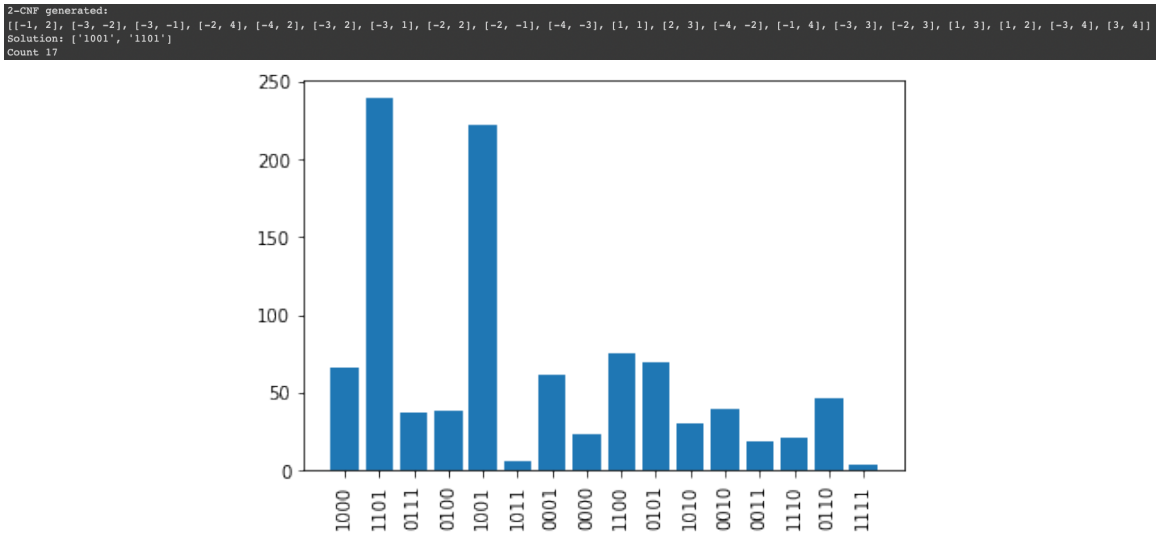

Figure 13: First figure shows the 2CNF (with m = 20 and n = 4 and t = 17) used and corresponding set of solutions which give the maximum count and bottom figure is the histogram of assignment values for 1000 iterations for best β and γ

- We observe as expected that histogram has 2 big peaks for two solutions `[1001]` and `[1101]` (variable assignments in order for 4 variables), as expected as we can see that both of these assignments of values for variables has highest count = 17
- The rest of assignment values get a lesser peak in histogram as the separator and mixer circuits reduced their coefficients and hence the probability of observing
- We further observed that over various runs, sometimes a sub-otimal assignment gets a higher peak over optimal assignment. This sub-optimal assignment was observed to satisfy 1 or 2 less clauses than the optimal clause. The primary reason for this is that QAOA is an approximation algorithm and can give sub-optimal output (even for various iterations). In this process, since the sub-optimal assignment has almost equal clauses satisfied as optimal one, it leads to getting picked up many times during runs.

- As mentioned in paper, as we increase the number of mixer and seperator circuits to infinity (currently its just 1 circuit for each), we measure optimal solution always.

1.2.6 Variation of various Times with n (number of variables)

We conducted various experiments where we varied number of variables and observed the various parts of circuit times

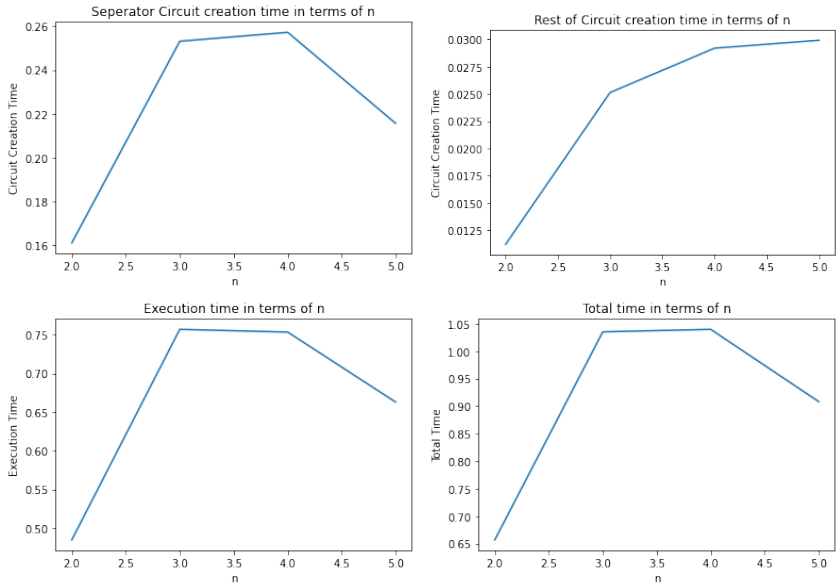

Figure 14: Variation of separator circuit creation time, rest of circuit creation time, Execution and total running time vs number of variables

Following were the observations in general

- The separator circuit was the deciding factor in the overall circuit times. In general separator circuit time heavily dependent on the actual clauses, hence we dont see a clear increase in seperator circuit times as we increase n.
- The seperator circuit time also decided the execution time, since majority of gates were contributed by seperartor circuit. Hence we see a similar, not exact, curve in execution time vs n graph
- The rest of circuit creation time was pretty small and random and highly dependent on how the compiler and interpreter works during run time, since such small times were comparable to other delays associated with run times of compilers. It may have points where it will decrease for higher n too.
- The total time hence was primarily decided by execution and separator circuit creation time, which indeed actually dependent effectively on seperator circuit creation time, hence we see a similar curve for total time vs n as well.

1.2.7 Variation of various Times with m (number of clauses)

We conducted various experiments where we analysed the times taken by various componets of circuit vs number of clauses. For this experiment, we increased the number of clauses in sequential manner, where we increase the number of clauses for our CNF one by one.

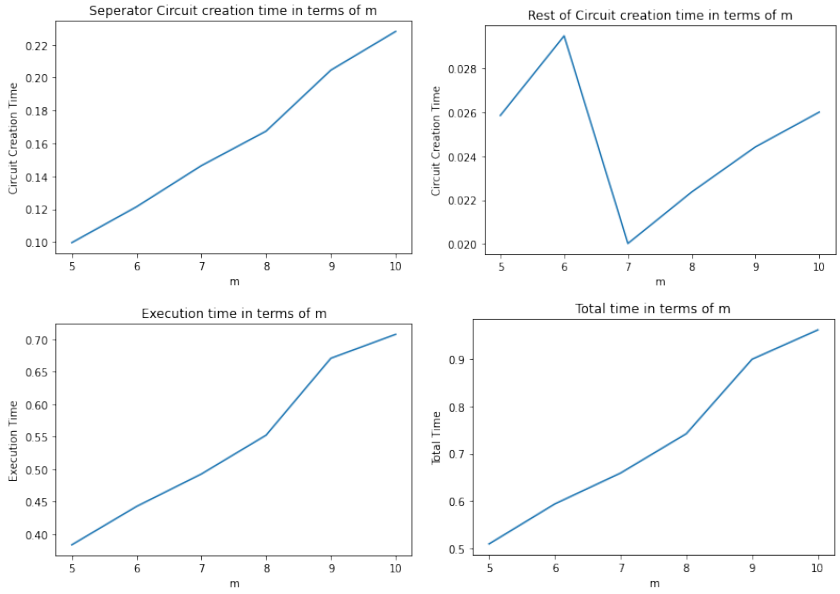

Figure 15: Variation of separator circuit creation time, rest of circuit creation time, Execution and total running time vs number of clauses

Following were the observations in general

- As discussed before, execution time and total time, more or less depend on seperator circuit creation time, and this is observed here too with similar curves in graph.
- The seperator circuit creation time this time was observed to increase always, this circuit iterates over the clauses and adds the gates correspondingly. Since, we added the clauses in sequential manner to our CNF, hence as m increased, the circuit creation time also increased.
- The rest of circuit creation time was again very small and had randomness associated with it due to reasons mentioned in previous section. The total time was not affected due to it much since the scale of values for this part were very small.

1.2.8 Variation of Accuracy versus (γ, β)

In this section ,we analyse how our approximation algorithm works in case we increase our search space. Specifically, we increased the number of beta and gamma divisions, keeping other paramter fixed, and observed the accuracy which is achieved by comparing output everytime with classical solver. For both of the experiments, we generated random 20 2CNF and used them to verify the accuracy for each gamma division.

- Accuracy vs Gamma divisions

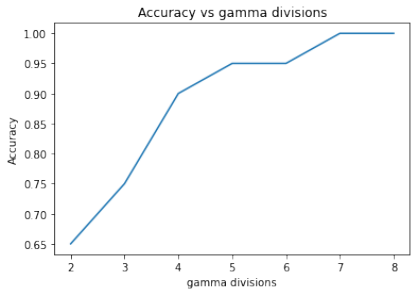

Figure 16: Variation of accuracy of circuit vs number of divisions in γ

Following were observations in general:

- We see that as we increase the number of divisions in γ, the accuracy in general increased.
- This we quite expected since our search space increases and hence we find a better optimal γ which will better separate to get the optimal assignment.

- Accuracy vs Beta divisions

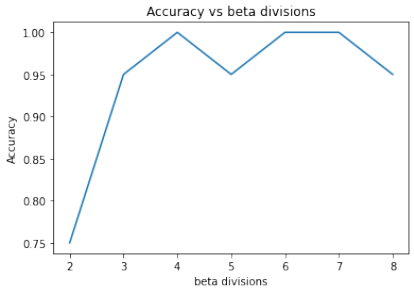

Figure 17: Variation of accuracy of circuit vs number of divisions in β

Following were observations in general:

- We see that as we increase the number of divisions in β, the accuracy in general increased.
- Again this was expected since our search space increases and hence we find a better optimal β which will better separate to get the optimal assignment.

In both of above analysis, its observed that sometimes there is drop in accuracy with increasing divisions. Following are the possible reasons:

- In both of graphs, over many random clauses, we observed that the decrease is not that much.
- One of the reasons for such dip in accuracy by little amount is that, the set of divisions of both β and γ are not exactly the superset of previous iteration set. For example, for β divisions = 3, we will have set $\beta = [0, \frac{\pi}{2}, \pi]$ and for β divisions = 4, we will have set $\beta = [0, \frac{\pi}{3}, \frac{2\pi}{3}, \pi]$. Here we observe that although we increased our search space, the optimal β might be near $\frac{\pi}{2}$ than either of $\frac{\pi}{3}$ or $\frac{2\pi}{3}$ and in that case accuracy will be higher for first case. Similar story for γ too. Hence we observe a dip in accuracy sometimes.
- Second reason for such fluctuations is that eventually this is an approximation algorithm and its very much possible that while measuring we would have measured a lower probable assignment of variables, which is sub-optimal and hence we observe sometimes lower accuracy. This is completely dependent on that particular run time.
- The dip in accuracy is always not much, which is obvious due to above reasons. In general, once we have increased divisions to great extent, the first cause effects starts to mitigate.

2 README

1. Download the python notebook. All the required libraries are installed in case you are running on colab. If you are running locally install cirq, numpy and matplotlib.
2. Follow these links: Install Cirq, Install numpy and Install matplotlib.
3. Run the notebooks one by one.
4. **Shor**: Call runMainCircuit with parameters n, verbose (optional, default = True) and max_attempts (optional, default = 25). eg: runMainCircuit(21, True, 20).

5. **QAOA**: Call runMainCircuit with parameters n, m , t,
    - lis (The 2CNF in format mentioned before). If kept empty, the function will create a randomized 2CNF during run time using `getFx(m,n)`. The CNF can also be explicitly provided to the function.
    - `gamma_divs` (optional, default = 5)
    - `beta_divs` (optional, default = 5)
    - repeats (optional, can be modified to draw histograms, when we wish to determine probabilistic outputs, default value = 1)
    - verbose (optional, default = True)

References

[1] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. *A Quantum Approximate Optimization Algorithm*. 2014. arXiv: `1411.4028 [quant-ph]`.