Optimization of Entanglement Purification Quantum Circuits With Evolutionary Computing Techniques

Muhammad Usaid

Habib University, Pakistan

Abstract: Entanglement Purification is an important accept in Quantum communication. In this paper, genetic algorithms is used to find the optimized circuit for entanglement purification.

Index Terms: Quantum Information, Genetic Algortihms, Quantum Computing, Entanglement Purification.

1. Introduction

1.1. Entanglement

Quantum Entanglement is a phenomenon in which states of quantum particles are dependent on each other. Entanglement is a really important concept in the study of Quantum Communication. It is a key element in effects such as quantum teleportaion, fast quantum algorithms and quantum error correction [1].

1.2. Bell States

Depending on the concept of quantum entanglement, we have another ingredient in quantum teleportation and super dense coding called Bell States [1]. These are based on pairs of entangled particles. The four basic bell states on 2-Qubits are following:

$$|A\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}} \tag{1}$$

$$|B\rangle = \frac{|01\rangle - |10\rangle}{\sqrt{2}} \tag{2}$$

$$|C\rangle = \frac{|01\rangle + |10\rangle}{\sqrt{2}} \tag{3}$$

$$|D\rangle = \frac{|00\rangle - |11\rangle}{\sqrt{2}} \tag{4}$$

These correlations are the reason of debate in physics community since the famous paper of Einstein, Podolsky and Rosen came out in which they pointed out the strange properties of states like Bell States. [2]

1.3. Fidelity

Fidelity is defined as following:

> "Fidelity measures the average distortion introduced by the compression scheme. The idea of quantum data compression is that the compressed data should be recovered with very good fidelity. Think of the fidelity as being analogous to the probability of doing the decompression correctly." [1]

Loosely Speaking, Fidelity is the degree of exactness or measure of closeness of two quantum states.

1.4. Entanglement Purification

Quantum communication highly depends on the purity of entanglement. The problem with entangled states is that due to unavoidable noise in quantum channel, the entangled states are disturbed. This will create problems where we won't be sure about the correctness of the information we are receiving on a quantum channel. This problem can be solved by entanglement purification protocols. Suppose there are two parties Alice and Bob where Alice wants to send classical information through noisy quantum channel to Bob. She wants to make sure that the entangled states are purified and not disturbed by noise while sending the information over a larger distance. It can be described as starting with a larger number of pure states ψ and repeating the process again and again to get the maximum copies of pure Bell state $|A\rangle$ from 1 using local circuits and classical communication [1]. It is also known as entanglement distillation in literature.

1.5. Local Circuits and Classical Communication

Local circuits and classical communication is described in quantum information theory as a method in which local operations are performed on a circuit and the result is communicated to another part of the circuit classically where another local operation is conditioned on the information received.

1.6. Quantum Circuits for Entanglement Purification

Quantum Circuits are used for Entanglement Purification. The idea is to break the circuit in two parts where half of the circuit is with Bob while the other half is with Alice [3]. They can only communicate classically. The Bell States are provided with given probabilities. The circuit is run different times to get the pure $|A\rangle$ state as many times as possible in it's purified form.

1.7. Genetic Algorithms

Within Evolutionary Computing, Genetic algorithm is a meta-heuristic inspired by Darwin's theory of natural selection that is used for solving optimization problems. According to natural selection, traits that enable survival are most likely to be carried into the successive generations through reproduction. As a result, with successive generations organisms with the best traits tend to reproduce more and eventually increase overtime.
Similarly, Genetic algorithms generate good solutions by evolving solutions through genetic processes including mutation, crossover and selection. The method was first proposed in 1960 according to which a number of candidate solutions are generated according to the population size. Each iteration of the program is called a generation and at every generation some candidates are selected for crossover using selection. After crossover, the off springs are mutated and according to a selection criteria, the candidates are sent to the next generation. A fitness function determines how fit a candidate solution is and helps in the selection process [4].

2. Method

2.1. Genetic Algorithm For Entanglement Purification

In this paper, I have applied genetic algorithm to make a protocol for entanglement purification. The individuals consists of the circuit as their attribute. The algorithm ran for 50 generations.

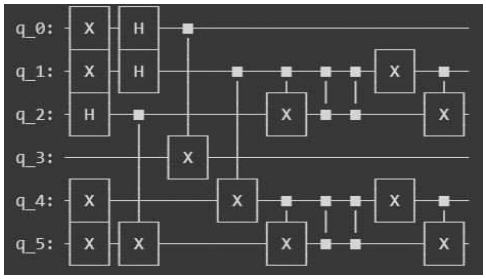

Fig. 1. Circuit from 50th generation

The parents were selected with rank based scheme. The survivors were selected with binary tournament scheme. The algorithm seemed to converge before 50 generations. The fitness values were all close to 0.6. The code used is provided in the appendix.

2.2. Representation

The circuit is represented as a list of tuples. The tuples contain the circuit gate and the respected wire to act on. For example, ('x',(1)) means X gate on wire 1, ('cx',(1,2)) means CNOT gate on wire 2 controlled by wire 1.

2.3. Crossover

The crossover was performed between 2 individuals or parents and four kids were produced. The first kid was produced by taking first half of parent 1 and second half of parent 2 excluding the measurement gates. The measurement gates were added at the end, one of each parent. The second kid was produced in the same way but exchanging the halves of both parents. The third and fourth kid were produced by taking random gates from both parents having a random length of the circuit. This was done to take into account the exploration even in crossover operation.

2.4. Mutation

The mutation was performed by exchanging any two gates in the middle of the circuit. This was dependant on mutation rate. Mutation rate started from 100 in the first generation and then it decreased. At the end of every generation, mutation rate was multiplied by 0.98. Mutation helped to explore some more possibilities.

2.5. Fitness Function

The fitness function was dependent of some different parameters. The 'operations' was defined as the total number of gates used in the circuit. The 'steps' is defined as the total number of steps taken to execute the circuit. The difference between operations and steps is that steps take into account the parallel gates that can be performed in a single time unit. The 'probability' p is defined as the probability of getting a coincidence measurement. The fitness function is then defined by the equation

$$f = 0.2 \times \frac{1}{operations} + 0.2 \times \frac{1}{steps} + 0.6 \times p \tag{5}$$

3. Results

Few circuits from the 50th generation are shown in Fig 1, 2 and 3. The graph in Fig 4 shows the relation between generations and average fitness. We can see the average fitness increasing with generations and converging to 0.67.

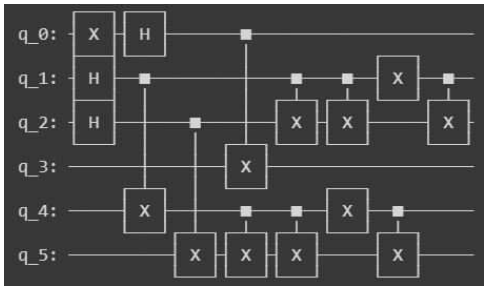

Fig. 2. Circuit from 50th generation

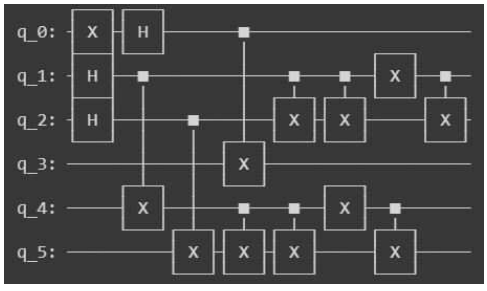

Fig. 3. Circuit from 50th generation

4. Conclusions

The problem of generating circuits with Genetic Algorithms is a really promising approach and can be used to find more new circuits.

4.1. Future Improvements

This algorithm can be checked with different schemes and fine tuning the hyper parameters. Then we can see improve the circuit to see it's performance on more than 3 qubits. Other than that, more techniques like ant colony optimization might be used to see if they provide better results.

Acknowledgements

Special Thanks to Dr. Abdullah Khalid for his support and help in understanding the problem.

5. Appendix

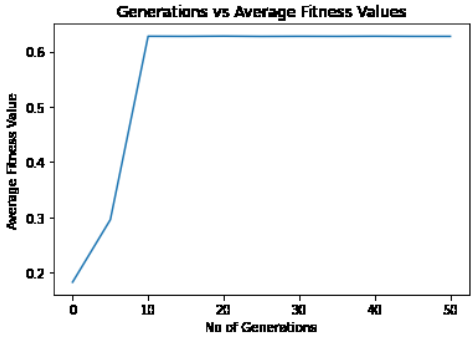

Fig. 4. Graph Showing the result

```
class Individual:
  def __init__(self, n):
    self.qreg = QuantumRegister(n * 2, 'q')
    self.creg = ClassicalRegister(n * 2, 'c')
    self.circ =  QuantumCircuit(self.qreg, self.creg)
    self.n = n
    self.gates = []
    self.operations = 0
    self.steps = 0
  def generateRandomCircuit(self):
    n = self.n
    gates = random.choices(['cx','cz','x','h'], k = random.randint(2,8)
       )
    gates += random.choices(['X','Z'], k = 2)
    j = n-1
    for i in gates:
      if i == 'cx' or i == 'cz':
        a,b = random.sample(set(range(1,n)),2)
        self.gates.append((i, (a,b)))
      elif i == 'h' or i == 'x':
        self.gates.append((i,random.randint(0,n-1)))
      elif i == 'X' or i == 'Z':
        self.gates.append((i, j))
        j -= 1
  def updateParameters(self):
    self.steps = self.circ.width()
    self.operations = len(self.circ)
  def measurementX(self, qbit, cbit):
    '''It will measure in X Basis'''
    self.circ.h(qbit)
    self.circ.measure(qbit, cbit)
    self.circ.h(qbit)
  def measurementZ(self, qbit, cbit):
    '''Measurement in usual z basis which is default'''
    self.circ.measure(qbit, cbit)
  def bellStates(self, qbit1, qbit2):
    '''Will take circuit and qubits to prepare Bell States'''
    self.circ.h(qbit1)
    self.circ.cx(qbit1,qbit2)
  def initializeCircuit(self):
    n = self.n
    self.qreg = QuantumRegister(n * 2, 'q')
    self.creg = ClassicalRegister(n * 2, 'c')
    self.circ =  QuantumCircuit(self.qreg, self.creg)
  def initializeInput(self, inputs):
    # To put x gates according to the input
    for i in range(len(inputs)):
      if inputs[i] == 1:
        self.circ.x(self.qreg[i])
    n = len(inputs)
    for i in range(0,n//2):
```

```
      self.bellStates((self.qreg)[i], (self.qreg)[i+n//2])
  def addGates(self):
    j = 0
    while j < self.n+1:
      for i in self.gates:
        key = i[0]
        val = i[1]
        if type(val) == type(1):
          val += j
          if key == 'x':
            self.circ.x(val)
          elif key == 'h':
            self.circ.h(val)
          elif key == 'X':
            self.measurementX(self.qreg[val], self.creg[val])
          elif key == 'Z':
            self.measurementZ(self.qreg[val], self.creg[val])
        elif len(val) == 2:
          a,b = val
          a += j
          b += j
          if key == "cx":
            self.circ.cx(a,b)
          elif key == "cz":
            self.circ.cz(a,b)
      j += self.n
        #print(self.circ)
    self.updateParameters()
  def showResults(self):
    simulator = Aer.get_backend('qasm_simulator')
    job = execute(self.circ,simulator,shots=1000)
    result = job.result()
    counts = result.get_counts()
    return counts
  def checkCoincidence(self,counts):
    '''This Function checks if measurements on respective sacrificial
       pairs are coincidence or not'''
    for i in counts:
      if not (i[0] == i[3] and i[1] == i[4]):
        return False
    return True
  def coincidenceProbability(self):
    dic = {'A': (0,0), 'B': (1,1), 'C': (0,1), 'D': (1,0)}
    n = self.n
    inputs = [0] * (n * 2)
    F = 25 #The value of F is multiplied by 100 to use in choices
       method properly
    q = (100-F)/3
    weights = [F, q, q, q]
    countCoincidence = 0
    for _ in range(100):
```

```
      for i in range(n):
        arg = random.choices(["A", "B", "C", "D"], weights = weights, k
            = 1)
        inputs[i],inputs[i+n] = dic[arg[0]]
      self.initializeCircuit()
      self.initializeInput(inputs)
      self.addGates()
      counts = self.showResults()
      if self.checkCoincidence(counts):
        countCoincidence += 1
    return countCoincidence/100
  def fitness(self):
    prob = self.coincidenceProbability()
    return 0.2 * (1/self.steps) + 0.2 * (1/self.operations) + 0.6 * (
        prob)
class geneticAlgorithm:
  def generateRandomPopulation(self, size, numberOfQubits):
    population = []
    for i in range(size):
      pop = Individual(numberOfQubits)
      pop.generateRandomCircuit()
      population.append(pop)
    return population
  def calculateFitnesses(self, population):
    fitnesses = []
    for i in population:
      fitnesses.append(i.fitness())
    return fitnesses
  def crossover(self, sol1, sol2):
    n1 = (len(sol1) - 2) // 2
    n2 = (len(sol2) - 2) // 2
    n = n1 + n2
    sol = sol1[:len(sol1)-2] + sol2[:len(sol2)-2]
    kid1 = sol1[:n1] + sol2[n2:1+(2*n2)] + [sol1[-1]] + [sol2[-2]]
    kid2 = sol2[:n2] + sol1[n1:1+(2*n1)] + [sol1[-2]] + [sol2[-1]]
    kid3 = random.choices(sol, k = random.randint(2,(2*n)-1))
    kid4 = random.choices(sol, k = random.randint(2,(2*n)-1))
    kid3 += [sol1[-1]] + [sol2[-2]]
    kid4 += [sol1[-2]] + [sol2[-1]]
    return [kid1, kid2, kid3, kid4]
  def mutation(self, sol, rate):
    if random.randint(0,100) < rate:
      a, b = random.randint(0,len(sol) - 2), random.randint(0,len(sol)
          - 2)
      sol[a],sol[b] = sol[b], sol[a]
    return sol
  def fitnessProportional(self, fitnesses, numberSelected):
    '''Takes a List of fitness Values of Solutions and returns a list
        of the numberSelected indexes of selected solutions'''
    F = sum(fitnesses)
    newFit = list(map(lambda x: x/F,fitnesses))
```

```
    for i in range(1,len(fitnesses)):
      newFit[i] = newFit[i] + newFit[i-1]
    result = []
    for k in range(numberSelected):
      r = random.random()
      for i in range(len(fitnesses)):
        if r < newFit[i]:
          result.append(i)
          break
    return result
  def rankBased(self, fitnesses, numberSelected):
    '''Takes a List of fitness Values of Solutions and returns a list
       of the numberSelected indexes of selected solutions'''
    rank = [(fitnesses[i],i) for i in range(len(fitnesses))]
    rank.sort()
    newFit = fitnesses.copy()
    for i in range(len(rank)):
      newFit[rank[i][1]] = i+1
    return self.fitnessProportional(fitnesses, numberSelected)
  def binaryTournament(self, fitnesses, numberSelected):
    '''Takes a List of fitness Values of Solutions and returns a list
       of the numberSelected indexes of selected solutions'''
    result = []
    for i in range(numberSelected):
      r1 = random.randint(0,len(fitnesses)-1)
      r2 = random.randint(0,len(fitnesses)-1)
      result.append(max(r1,r2))
    return result
  def randomSelection(self, fitnesses, numberSelected):
    '''Takes a List of fitness Values of Solutions and returns a list
       of the numberSelected indexes of selected solutions'''
    result = []
    for i in range(numberSelected):
      result.append(random.randint(0,len(fitnesses)-1))
    return result
  def truncation(self, fitnesses, numberSelected):
    '''Takes a List of fitness Values of Solutions and returns a list
       of the numberSelected indexes of selected solutions'''
    rank = [(fitnesses[i],i) for i in range(len(fitnesses))]
    rank.sort(reverse=True)
    return [rank[i][1] for i in range(len(rank))][:numberSelected]
  def select(self, solutions, fitnesses, number, scheme):
    '''Takes a list of solutions, fitnesses and returns the number
       selections according to the Scheme'''
    indexes = []
    if scheme == "fps":
      indexes = self.fitnessProportional(fitnesses, number)
    elif scheme == "rbs":
      indexes = self.rankBased(fitnesses, number)
    elif scheme == "bin":
      indexes = self.binaryTournament(fitnesses, number)
```

```
    elif scheme == "random":
      indexes = self.randomSelection(fitnesses, number)
    elif scheme == "trun":
      indexes = self.truncation(fitnesses, number)
    else:
      print("Error In Parent Selection: Scheme is wrong \n Enter fps,
          rbs, bin, random or trun")
      return
    result = []
    resultFitness = []
    for i in indexes:
      result.append(solutions[i])
      resultFitness.append(fitnesses[i])
    return result, resultFitness
  def evolve(self, populationSize, generations, numberOfQubits,
      parentScheme, survivorScheme):
    population = self.generateRandomPopulation(populationSize,
        numberOfQubits)
    mutationRate = 100
    for g in range(generations):
      print("Generation:",g)
      fitnesses = self.calculateFitnesses(population)
      print("Fitnesses:", fitnesses)
      parents = self.select(population, fitnesses, populationSize//2,
          parentScheme)[0]
      kids = []
      for i in range(0,len(parents),2):
        kids += self.crossover(parents[i].gates,parents[i+1].gates)
      newPop = []
      #print(kids)
      for i in kids:
        c = self.mutation(i,mutationRate)
        a = Individual(numberOfQubits)
        a.gates = c
        #print(i)
        newPop.append(a)
      kidFitnesses = self.calculateFitnesses(newPop)
      population += newPop
      fitnesses += kidFitnesses
      population = self.select(population, fitnesses, populationSize,
          survivorScheme)[0]
      mutationRate *= 0.98
    return population
```

References

[1] M. A. Nielsen and I. Chuang, *Quantum computation and quantum information*. American Association of Physics Teachers, 2002.

[2] A. Einstein, B. Podolsky, and N. Rosen, "Can quantum-mechanical description of physical reality be considered complete?" *Physical review*, vol. 47, no. 10, p. 777, 1935.

[3] K. Fujii and K. Yamamoto, "Entanglement purification with double selection," *Physical Review A*, vol. 80, no. 4, p. 042308, 2009.

[4] D. Whitley, "A genetic algorithm tutorial," *Statistics and computing*, vol. 4, no. 2, pp. 65–85, 1994.