Cirq'a Giriş ve Kuantum Algoritmalar

Prof. Dr. İhsan Yılmaz ve Osman Semi Ceylan

Contents

1 Giriş . . . 5
1.1 Cirq Nedir? . . . 5
1.2 Cirq Ne Yapar? . . . 5
1.3 Kurulum . . . 5
1.3.1 Sorun Giderme . . . 5
2 İş akışı . . . 5
2.1 İş akışının unsurları . . . 6
2.2 Adım Adım İlk Kullanım . . . 6
2.2.1 Paketlerin İçe Aktarımı . . . 6
2.2.2 Devrenin Başlatılması . . . 6
2.2.3 Devreyi Tasarlamak . . . 7
2.2.4 Devrenin Çizimi . . . 8
2.2.5 Devrenin Simülasyonu . . . 8
2.2.6 Sonuçların Görselleştirilmesi . . . 8
3 Algoritmalar . . . 13
3.1 Işınlama Algoritması . . . 13
3.1.1 Kurulum . . . 13
3.1.2 Adım 1: Bell durumu paylaşma . . . 14
3.1.3 Adım 2: Alice qubiti kodlar . . . 14
3.1.4 Adım 3: Alice ölçer . . . 15
3.1.5 Adım 4: Bob işlemleri . . . 15
3.1.6 Simülasyon . . . 15
3.2 Quantum Fourier Dönüşümü . . . 17
3.2.1 Kurulum . . . 17
3.2.2 Adım 1: Devre hazırlama . . . 17
3.2.3 Simülasyon . . . 18
3.3 Ters Quantum Fourier Dönüşümü . . . 19
3.3.1 Kurulum . . . 19
3.3.2 Adım 1: Devre Hazırlama . . . 19
3.3.3 Kuantum Fourier ile Ters Kuantum Fourier . . . 20

3.3.4 Simülasyon . . . 21
3.4 Faz Tahmini Algoritması . . . 22
3.4.1 Kurulum . . . 23
3.4.2 Adım 1: Gereken Değerler Seçilir . . . 23
3.4.3 Adım 2: Hadamard . . . 24
3.4.4 Adım 3: Kontrollü-Faz Kapıları . . . 24
3.4.5 Adım 4: Ters QFT . . . 24
3.4.6 Adım 5: Ancilla kubiti . . . 25
3.4.7 Adım 6: Ölçümler . . . 25
3.4.8 Simülasyon . . . 25
3.5 Deutsch-Jozsa Algoritması . . . 26
3.5.1 Kurulum . . . 26
3.5.2 Adım 1: Hadamard . . . 26
3.5.3 Adım 2: Oracle . . . 27
3.5.4 Adım 3: Tekrar Hadamard . . . 29
3.5.5 Adım 4: Ölçüm . . . 29
3.5.6 Simülasyon . . . 29
3.6 Shor Algoritması . . . 30
3.6.1 Kurulum . . . 30
3.6.2 Adım 1: Hadamard . . . 31
3.6.3 Adım 2: AmodN Dönüşümü . . . 31
3.6.4 Adım 3: Ters Kuantum Fourier Dönüşümü . . . 33
3.6.5 Adım 4: Ölçüm . . . 34
3.6.6 Simülasyon . . . 34
3.6.7 Adım 5: Periyot Bulma . . . 35
3.7 Grover Algoritması . . . 36
3.7.1 Kurulum . . . 36
3.7.2 Adım 1: Hadamard . . . 37
3.7.3 Adım 2: Oracle . . . 37
3.7.4 Adım 3: Diffision . . . 38
3.7.5 Adım 4: Ölçüm . . . 38
3.7.6 Simülasyon . . . 39
3.8 Süperyoğun Kodlama Algoritması . . . 40
3.8.1 Kurulum . . . 41
3.8.2 Adım 1: Bell çiftinin hazırlanması . . . 41
3.8.3 Adım 2: Mesajın kodlanması . . . 41
3.8.4 Adım 3: Bob'un mesajı çözmesi . . . 42
3.8.5 Simülasyon . . . 42
3.9 Simon Algoritması . . . 43
3.9.1 Kurulum . . . 44

3.9.2 Adım 1: Hadamard . . . 44
3.9.3 Adım 2: Simon Oracle . . . 45
3.9.4 Adım 3: Tekrar Hadamard . . . 45
3.9.5 Adım 4: Ölçüm . . . 45
3.9.6 Simülasyon . . . 46
4 Yüksek Boyuttan Algoritmalar . . . 48
4.1 Temel Dönüşüm Matrisleri . . . 48
4.1.1 Dönüşümleri Uygulama . . . 49
4.1.2 Temel Dönüşüm Kapısı . . . 51
4.1.3 Temel ID Kapısı . . . 52
4.1.4 Hadamard Kapısı . . . 52
4.1.5 Hermitik Hadamard Kapısı . . . 53
4.1.6 Kontrollü Not Kapısı . . . 54
4.2 Yüksek Boyutta Işınlama Algoritması . . . 55
4.2.1 Kurulum . . . 55
4.2.2 Devre Tasarımı . . . 55
4.2.3 Simülasyon . . . 57
4.3 Yüksek Boyutta Kuantum Fourier Dönüşümü . . . 57
4.3.1 Kurulum . . . 57
4.3.2 Devre Tasarımı . . . 59
4.3.3 Simülasyon . . . 61
4.4 Yüksek Boyutta Bernstain-Vazirani Algoritması . . . 61
4.4.1 Kurulum . . . 62
4.4.2 Adım 1: Hadamard . . . 63
4.4.3 Adım 2: Aktarım Devresi Tasarla . . . 63
4.4.4 Adım 3: Tekrardan Hadamard . . . 63
4.4.5 Adım 4: Ölçüm . . . 64
4.4.6 Simülasyon . . . 64
4.5 Gelişmiş Grover Algoritması . . . 65
4.5.1 Kurulum . . . 65
4.5.2 Adım 1: Hadamard . . . 66
4.5.3 Adım 2: Oracle . . . 66
4.5.4 Adım 3: Diffission . . . 67
4.5.5 Adım 4: Tekrarlama . . . 68
4.5.6 Adım 5: Ölçüm . . . 68
4.5.7 Simülasyon . . . 68
4.6 Yüksek Boyutta Shor Algoritması . . . 69
4.6.1 Kurulum . . . 69
4.6.2 Adım 1: Hadamard . . . 71
4.6.3 Adım 2: Kontrollü Amod15 . . . 71

4.6.4 Adım 3: Ters Kuantum Fourier Dönüşümü 72
4.6.5 Adım 4: Ölçüm 73
4.6.6 Simülasyon 73
4.6.7 Adım 5: Periyot Bulma 74

1 Giriş

1.1 Cirq Nedir?

Cirq, IBM tarafından geliştirilen, açık kaynak kodlu, bulut üzerinden gerçek kuantum bilgisayarlarına erişim izni tanıyan bir kuantum devre simülatörüdür. Cirq bir devre simülatörünün dışında bünyesinde amacına yönelik çok çeşitli farklı modülleri de barındıran çevresel bir yazılımdır.

1.2 Cirq Ne Yapar?

Cirq, kuantum sistemleri ve simülatörlerle etkileşim için gereken eksiksiz araç setini sağlayarak kuantum uygulamalarının geliştirilmesini hızlandırır.

1.3 Kurulum

Cirq, Python bulunduran tüm masaüstü işletim sistemlerinde çalışacak şekilde geliştirilmiştir. Linux, MacOS veya Windows üzerinde Python 3.6 veya üzeri bir sürüm ile kullanılabilir.

```
1 # Bu paket tüm Cirq modüllerini yükler.
2 pip install -U cirq
```

Uzakta veya yerelde bulunan Jupyter Notebook gibi Python interaktif ortamlarda kurulum kaçış karakteri ile şu şekilde yapılabilir.

```
1 # Bazen iki adet (%) gerekebilir.
2 %pip install -U cirq
```

1.3.1 Sorun Giderme

Cirq kurulumu gereksinimlerin karşılanamadığı durumlarda tamamlanamaz. Böyle bir durumda sanal bir Python ortamı üzerinden devam edilmedir.

```
1 # Komut satırı üzerinden
2 python3 -m venv /path/to/new/virtual/environment
```

Daha fazla bilgi almak için tıklayınız!

2 İş akışı

Bir kullanıcı herhangi bir yazılımı kullanmadan önce yazılımın nasıl çalıştığını bilmesi gerekmektedir. İlk öğrenmenin en başarılı yolu ise genel kullanım durumunun bir örnek üzerinden anlatılmasıdır.

2.1 İş akışının unsurları

Cirq'i kullanırken, bir kullanıcı için genel kullanım durumu olarak aşağıdaki dört üst düzey adımdan oluşur:

- **Tasarım**: Bu aşamada kullanıcı kendi problemini temsil ettiği devreyi tasarlamaları beklenmektedir.
- **Derleme**: Bu aşamada kullanıcı belirli bir kuantum hizmeti için devrelerini derlemelidir. Bu aşama çoğu zaman kullanıcı etkileşimi olmadan otomatik yapılmaktadır.
- **Çalıştırma**: Bu aşamada kullanıcı aktif değildir, yerelde veya bulut üzerinden tasarlanmış devre simüle edilir.
- **Analiz**: Bu aşamada simülasyondan gelen sonuçlar kullanıcı tarafından analiz edilerek bir sonuç elde eder. Bu aşamada görsel bir plot kullanılması simülasyondan dönen sonuçların analiz edilmesini kolaylaştırır.

2.2 Adım Adım İlk Kullanım

Bu bölümde bir devre oluşturarak simülasyon yaptığımız bir örneği adım adım anlatacağız. Bu örnekte anlatılacak olanlar **jupyter_notebooks** dosyasının içinde **cirqe_giris.ipynb** dosyasında incelenebilir.

2.2.1 Paketlerin İçe Aktarımı

Bir uygulama geliştirme sürecinde her zaman ilk adım gereksinimlerin içe aktarımı olmuştur. Cirq kendi içinde bulundurduğu tüm modülleri tek bir başlıkla içe aktarır.

```
1 import numpy
2 import cirq
```

Numpy: Numpy Python üzerinde popüler çok amaçlı bir matematik ve lineer cebir kütüphanesidir. Örnekler üzerinde kullanacağız.

2.2.2 Devrenin Başlatılması

Bu aşamada kullanıcılar devrelerini ilklemelidir. Kullanıcı programlamak istediği kuantum devrenin qudit boyutunu ve sayısını girmelidir. Cirq devre oluşturma üzerinde geniş çaplı yöntemler kümesi sunmaktadır. Aşağıda en basit ve en etkin olduğunu düşündüğümüz yöntem ile kuantum devreyi başlatacağız.

```
# Devreyi ilkler.
my_circuit = cirq.Circuit()

# 2 boyutta 2 adet qubit oluşturur.
qubits = cirq.LineQubit.range(2)

# 2'den farklı boyutta şöyle kullanılabilir.
# Örnek 4.boyuttan 2 adet kudit.
qudits = cirq.LineQid.range(2, dimension=4)

print("List of qubits in circuit: ", qubits)
```

```
List of qubits in circuit:
[cirq.LineQubit(0), cirq.LineQubit(1)]
```

2.2.3 Devreyi Tasarlamak

Bu aşamada kullanıcılar devre üzerinde istedikleri devre elemanlarını istedikleri yere konumlandırmadır. Cirq üzerinde kullanılabilen devre elemanları şunlardır:

- **Kapı**: Operatör olarak da adlandırılan bu devre elemanı devrenin en önemli yapı taşıdır. Uygulanmak istenen kapının girdi sayısı ile kapının uygulandığı kubit sayısı ve kubit boyutu ile eşleşmelidir.
- **Ölçüm**: Ölçüm operatörü ile kullanıcılar devrelerinde kubitler üzerinde ölçüm yapabilirler.
- **Bariyer**: Bu devre elemanı devre akışı içerisinde kapılar arasında bariyer sağlayarak oluşabilecek bazı karmaşıklıklardan kaçınabilirler.
- **İkili Kontrollü Kapılı**: Bu kapılar adından da anlaşılacağı gibi klasik bit kontrol olacak şekilde kubitlere kapı uygular.

```
# Hadamard ilk kubite uygulanır.
my_circuit.append(cirq.H(qubits[0]))

# CNOT ilk ve ikinci kubite uygulanır. (Ilk kubit
  kaynak, ikinci kubit hedef)
my_circuit.append(cirq.CNOT(qubits[0], qubits[1]))
my_circuit.append(cirq.measure(qubits[0], qubits[1]))
```

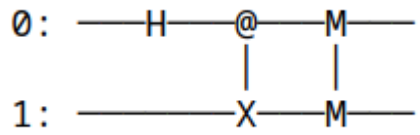
Devreye operatörler **append** metodu ile eklenirler. Bu metot parametre olarak eklenmesi istenen kapıyı almaktadır.
Üstteki devre ile bir Bell çifti oluşturulur ve her iki kubiti de ölçülür:

- *Hadamard: cirq.H()* ile belirlen kapı bir kubiti süperpozisyon durumuna getirir. Burada ilk kubite (0) uygulanmaktadır.
- *Kontrollü-NOT: cirq.CNOT()* ile belirtilen kapı iki kubitlik gerektirir. Burada ilk kubit (0) kaynak, ikinci kubit (1) ise hedeftir.
- *Ölçüm: cirq.measure()* ile ölçüm işlemi yapılabilir. Kubitler parametre olarak bu metoda gönderilmelidir.

2.2.4 Devrenin Çizimi

Tasarlanan devreler *print()* yöntemi ile çizilebilir.

```
1 print(my_circuit)
```



2.2.5 Devrenin Simülasyonu

Cirq kuantum devreler için yüksek boyutlu kudit bulunduran devreleri de okuyabilen bir simülatöre sahiptir. Bu simülatörü kullanarak tasarladığımız devreyi 1024 kere simüle edecek kod aşağıdaki gibi yazılabilir.

```
1 # Repetitions kullanarak bir devre için örnekleme elde
    edebiliyoruz.
2 sim = cirq.Simulator()
3 results = sim.run(my_circuit, repetitions=1024)
```

2.2.6 Sonuçların Görselleştirilmesi

Bir devrenin başarılı bir simülasyonunun sonunda Cirq bize bir sonuç objesi göndermektedir. Bu sonuç objesinin **measurements()** ve **historgram()** metotları büyük önem taşımaktadır.

Measurements() metotu

Devre simülasyonun sonucunda geri dönen objenin bir metodu olan bu çağrı bize devre içinde yapılmış olan ölçümlerin sonuçlarını bir Python sözlük (dictionary) olarak sunar.

```
1 print(type(results.measurements))
2
3 ... <class 'dict'>
```

Bilindiği üzere bir Python sözlüğü erişim anahtar kelimeler üzerinden yapılmaktadır. Örnekte olduğu üzere eğer devreyi tasarlar iken ölçüm operatörüne bir anahtar kelime kullanmaz isek Cirq otomatik olarak bu ölçüm sonuçları için birer anahtar kelime atamaktadır.

```
1 print(results.measurements.keys())
2
3 ... dict_keys(['0,1'])
```

Ölçüm eklendiği sırada eklenen anahtar kelimelere göre veya eklenmemiş ise çıkan sonuçlara göre bu anahtar kelimelerin ismi değişiklik göstermektedir. Mesela aşağıda bu örnek için ölçülen değerlere bakabiliriz.

```
1 print("Toplam ölçüm sayısı: ", len(results.measurements
    ['0,1']))
2 print("Ölçümler: ")
3 print(results.measurements['0,1'])
```

```
1 Toplam ölçüm sayısı:  1024
2 Ölçümler:
3 [[0 0]
4  [0 0]
5  [1 1]
6  ...
7  [1 1]
8  [0 0]
9  [1 1]]
```

Bu örnekte yapılmak istenen Bell çifti devresinin sonuçlarına bakılacak olursak her iki kubitin ölçüm sonucunun aynı olduğu gözlemlenebilir. Fakat ölçüm sonuçlarını böyle incelemek anlaşılabilirlik konusunda çok zayıf kalmaktadır. Bu nedenle ölçüm operatörü kullanırken kullanıcı tarafından eklenen anahtar kelimeleri önemlidir. Aşağıda bir ölçüm operatörüne farklı anahtar nasıl bağlandığı örneğimiz üzerinden gösterilmektedir.

```
# Tekrardan devreyi kuralım.
my_circuit = cirq.Circuit()
my_circuit.append(cirq.H(qubits[0]))
my_circuit.append(cirq.CNOT(qubits[0], qubits[1]))

# Bu adımda anahtar kelime verelim.
my_circuit.append(cirq.measure(qubits[0], qubits[1],
  key='R'))

sim = cirq.Simulator()
results = sim.run(my_circuit, repetitions=1024)
print(results.measurements.keys())

... dict_keys(['R'])
```

Yukarıda verilen kodda görüleceği üzere *measurement()* metodundan gelen sözlüğün anahtar kelimeleri değiştirilebilmektedir. Böylelikle istediğimiz ölçüm sonuçlarını gruplandırarak daha anlaşılır veriler elde edilebilir. Yine örnek üzerinden devam eden aşağıdaki kod ölçüm sonuçlarını ayrı inceleme fırsatı vermektedir.

```
# Tekrardan devreyi kuralım.
my_circuit = cirq.Circuit()
my_circuit.append(cirq.H(qubits[0]))
my_circuit.append(cirq.CNOT(qubits[0], qubits[1]))

# Bu adımda ölçümleri ayıralım.
my_circuit.append(cirq.measure(qubits[0], key='Kubit_0'
  ))
my_circuit.append(cirq.measure(qubits[1], key='Kubit_1'
  ))

sim = cirq.Simulator()
results = sim.run(my_circuit, repetitions=1024)
print("Keywords: ", results.measurements.keys())

... Keywords:  dict_keys(['Kubit_0', 'Kubit_1'])
```

Bu anahtar kelimeleri kullanarak ölçüm sonuçları ayrı gösterilebilir.

```
# Yukarıdan gelen anahtar kelimeler.
print("İlk kubit:", results.measurements["Kubit_0"])
print("İkinci kubit:", results.measurements["Kubit_1"])
```

```
İlk kubit:   İkinci kubit:
[[0]       [[0]
 [0]        [0]
 [0]        [0]
 ...        ...
 [0]        [0]
 [1]        [1]
 [1]]       [1]]
```

Yukarıda verilen kodda görüleceği üzere ölçüm için anahtar kelimeler ayrı tutulabilir. Aşağıdaki verilen kodda bu ölçüm sonuçlarını nasıl olasılıklar ile ifade edilebileceğini göstermektedir.

```
hist = dict()

# Sonuçlar için bir sözlük. 2 kubit = 4 farklı olası
  durum.
hist["00"] = 0
hist["01"] = 0
hist["10"] = 0
hist["11"] = 0

# Repetition kadar.
for i in range(len(results.measurements["Kubit_0"])):
    k1 = results.measurements["Kubit_0"][i]
    k2 = results.measurements["Kubit_1"][i]

    if k1 == k2 == 0:
        hist["00"] += 1
    elif k1 == k2 == 1:
        hist["11"] += 1
    elif k1 != k2 and k1 == 0:
        hist["01"] +=1
    else:
        hist["10"] += 1

print(hist)
```

```
{'00': 511, '01': 0, '10': 0, '11': 513}
```

Yukarıdaki sonuçtan görüleceği üzere Bell çifti oluşturma örneği ile "00" ile "11" durumlarının sonuçlarda görüldüğü ve birbirine çok yakın oranda oldukları anlaşılmaktadır. Bu değerler tekrarlanma sayısının toplamına eşittirler

ve her biri tekrarlanma sayısına bölündükleri takdirde yüz üzerinden (%) olarak olasılıkları ortaya çıkacaktır.

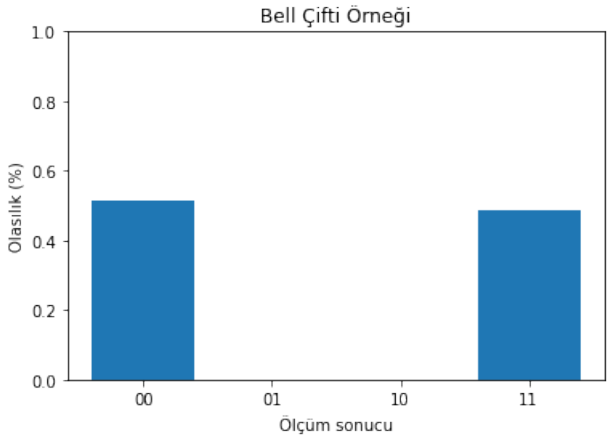
Histogram metotu

Cirq kuantum devre simulatörünün başarılı bir simülasyon geri dönen sonuçlar objesinin bir diğer metodu olan *historgram()* çağrısı ile sonuçlara daha kolayca görüntülenebilir. Fakat bu metot tek bir anahtar aldığı için ölçüm operatörü eklendiği zaman anahtar kelimenin ilgili şekilde gruplanması gerekmektedir. Bu Bell çifti örneğinde her iki kubitin sonuçlarını aynı gözlemlemek istediğimiz için aynı ölçümleri aynı anahtar kelime kullanarak gruplama yapılmalıdır. Aşağıda bu metodu bu örnek üzerinde nasıl kullanılacağını göstermektedir.

```
# Tekrardan devreyi kuralım.
my_circuit = cirq.Circuit()
my_circuit.append(cirq.H(qubits[0]))
my_circuit.append(cirq.CNOT(qubits[0], qubits[1]))

my_circuit.append(cirq.measure(qubits[0], qubits[1],
  key="R"))

sim = cirq.Simulator()
results_2 = sim.run(my_circuit, repetitions=1024)
print(results_2.histogram(key="R"))

# 00 = 0, 01 = 1, 10 = 2, 11 = 3
... Counter({3: 529, 0: 495})
```

Anlatılan **measurements()** ve **histogram()** metotları kullanılarak **matplot** üzerinde plotlama yapılabilir. Aşağıdaki kod ile daha önce *measurement()* bölümünde elde ettiğimiz sözlüğü kullanarak kolayca anlaşılabilir bir plot oluşturulmaktadır.

```
# Matplotlib kütüphanesini kurmayı unutmayın!
from matplotlib import pyplot

for key in hist:
  hist[key] /= 1024

print(hist)
pyplot.bar(hist.keys(), hist.values())
pyplot.ylim(0, 1.0)
```



3 Algoritmalar

Günümüze kadar gelen her ayrık matematik problemi klasik bilgisayarlar ile çözümü harcanan zaman ve enerji bakımından olanaklı görünmüyor. Kuantum bilgisayarlar ise klasiğe karşın olan üstünlüklerini kullanarak bu problemlere olanaklı çözümler getirebiliyor. Bu bölümün tamamında sadece kuantumda yapılabilen veya kuantum bilgisayarlar ile yapılmasının daha olanaklı olduğu algoritmaların Cirq üzerinde nasıl kodlandığını içermektedir.

3.1 Işınlama Algoritması

Bu algoritma bir kubitin durumunu uzak bir kubite aktarımını sağlamaktadır. Bir kuantum durum bir tarafta yıkılıp diğer tarafta oluşmasını sağladığı için adına ışınlama denilmektedir. Bu bölümde kodlanacak olan ışınlanma algoritması juptyter-notebooks klasörünün içerisinde **kuantum_teleportasyon.ipynb** dosyasında bulunmaktadır.

3.1.1 Kurulum

İlk etapta kullanacağımız kütüphaneler içe aktarılmalıdır. Algoritmanın adımlarına başlamadan önce devreyi de tanımlamak gerekmektedir. İki kubit (taraf) arasında gerçekleşecek bu algoritma için toplam 3 adet kubite ihtiyacımız vardır.

```
1 import cirq
2 import numpy as np
3 from matplotlib import pyplot
```

```
my_circuit = cirq.Circuit()

# 2 boyutta 3 adet qubit oluşturur.
qubits = cirq.LineQubit.range(3)
tp_qubit = qubits[0]
alice_qubit = qubits[1]
bob_qubit = qubits[2]
```

3.1.2 Adım 1: Bell durumu paylaşma

Bu adımda aşağıdaki kod ile devremiz üzerinde Bell durumunu oluşturacağız. Bu işlemi bir üçüncü tarafın yaptığını ve Alice ile Bob'a eşleri gönderdiğini düşünerek devam edeceğiz. Bu kubit eşlerinden q_1 Alice, q_2 ise Bob'a gönderildiğini varsayalım.

```
my_circuit.append(cirq.H(alice_qubit))
my_circuit.append(cirq.CNOT(alice_qubit, bob_qubit))
print(my_circuit)
```

```
1: ──H──@──
        │
2: ─────X──
```

3.1.3 Adım 2: Alice qubiti kodlar

Bu aşamada Alice, kendisine gelen q_1 eşi hedef ve ışınlamak istediği kubit olan q_0 kontrol kubiti olmak üzere *Kontrollü-not* kapısı ve ardından q_0 kubitine *Hadamard* uygular.

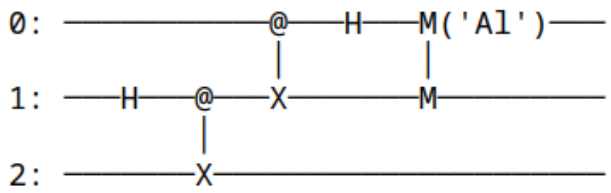
```
my_circuit.append(cirq.CNOT(tp_qubit, alice_qubit))
my_circuit.append(cirq.H(tp_qubit))
print(my_circuit)
```

```
0: ─────────@──H──
            │
1: ──H──@───X─────
        │
2: ─────X─────────
```

3.1.4 Adım 3: Alice ölçer

Bu aşamada Alice kendi elinde olan q_0 ve q_1 kubitleri ölçer ve ölçüm sonuçlarını Bob'a gönderir.

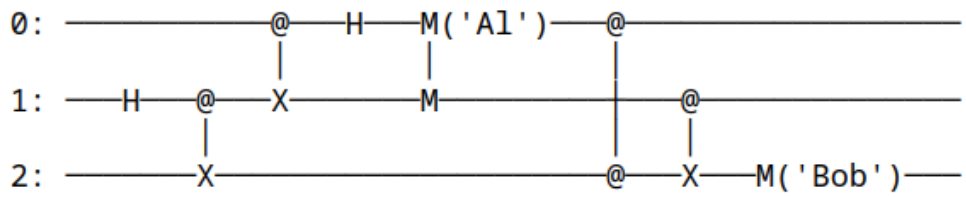
```
1 my_circuit.append(cirq.measure(tp_qubit, alice_qubit,
   key="Al"))
2 print(my_circuit)
```



3.1.5 Adım 4: Bob işlemleri

Ölçüm sonuçlarını alan Bob, bu verilere göre kendi elindeki q_2 kubitine kapı veya kapılar uygular. Fakat Cirq klasik kontrollü kapıların kullanılmasına izin vermemektedir. Bu nedenle eş değerliliğini CZ ve CX kapıları kullanılarak yapılabilir.

```
1 my_circuit.append(cirq.CZ(tp_qubit, bob_qubit))
2 my_circuit.append(cirq.CX(alice_qubit, bob_qubit))
3 print(my_circuit)
```



3.1.6 Simülasyon

Bu aşamada tasarladığımız bu devreyi simüle etmemiz gerekir.

```
1 sim = cirq.Simulator()
2 results = sim.run(my_circuit, repetitions=1024)
```

```
1 pyplot.hist(results.measurements["Bob"])
2 pyplot.xticks([0, 1])
3 pyplot.xlim(-0.2, 1.2)
4 pyplot.title("Işınlama algoritması")
5 pyplot.xlabel("Ölçüm sonucu")
6 pyplot.ylabel("Adet")
```

Işınlamak istediğimiz kubiti temel durumunda ele almıştık. Bu nedenle Figür-2(a)'da Bob'un ölçümüne ait olan klasik bitin (ilk olan) ölçüm soncunu hep "0" olur. Eğer Alice'in ışınlanmak istenen durum q_0 kubitine X uygulanırsa Figür-2(b)'de Bob'un ölçümüne ait olan klasik bitin (ilk olan) ölçüm soncunu hep "1" olur. Eğer Alice'in ışınlanmak istenen durum q_0 kubitine *Hadamard* uygulanırsa Figür-1'deki gibi tüm durumları görme imkanımız olur.

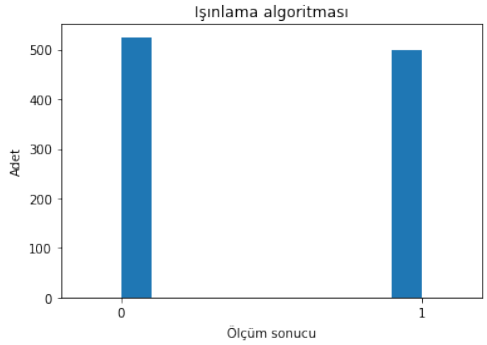
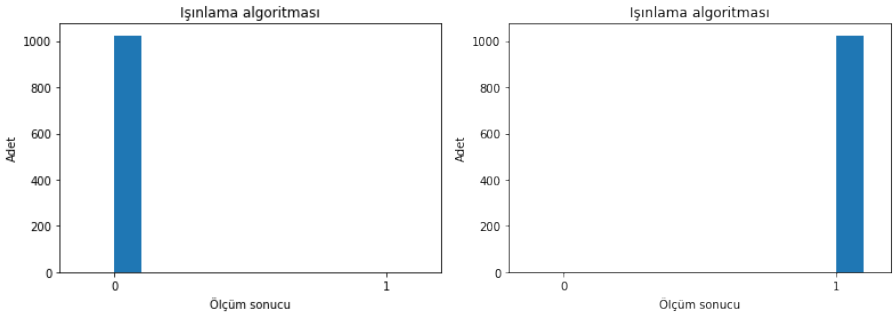


Fig. 1: Işınlaması istenen tüm durumlar



(a) Işınlaması istenen kubit "0" ise (b) Işınlaması istenen kubit "1" ise

Fig. 2: Işınlanma algoritması, Bob sonuçları

3.2 Quantum Fourier Dönüşümü

Bu algoritma klasikte sinyal işlemede ve veri sıkça kullanılmaktadır. Kuantumda ise dalga fonksiyonunun genlikleri üzerinde bu dönüşümü uygular. Bu bölümde kodlanacak olan kuantum fourier algoritması juptyter-notebooks klasörünün içerisinde **qft_alg.ipynb** dosyasında bulunmaktadır.

3.2.1 Kurulum

```
1 import cirq
2 import numpy as np
3 from matplotlib import pyplot
```

3.2.2 Adım 1: Devre hazırlama

Bu adımda genel bir kuantum fourier dönüşümü yöntemi yazacağız. Bunu yaparken *Kontrollü-faz(CPhase)* kapılarını farklı açılarda kullanacağız. Bu örnekte aşağıda hazırlanan fonksiyonu 4 kubit için kullanılacaktır. Aşağıda farklı açılarda kullandığımız bu *Kontrollü-faz* kapılarının aynı zamanda Z, T, S kapısı gibi özel isimleri bulunmaktadır.

```
# Bir QFT Devresi oluşturur.
def make_qft(qubits):
    qreg = list(qubits)
    while len(qreg) > 0:
        q_head = qreg.pop(0)
        yield cirq.H(q_head)
        for i, qubit in enumerate(qreg):
            yield (cirq.CZ ** (1 / 2 ** (i + 1)))(qubit
   , q_head)
```

```
1 qubits = cirq.LineQubit.range(4)
2 qft = cirq.Circuit(make_qft(qubits))
3 print(qft)
```

```
0: ──H──@──────@──────@──────────────────────────────
        │      │      │
1: ─────@^0.5───┼──H───@──────@──────────────────────
                │             │
2: ─────────────@^0.25────────@^0.5────H────@─────────
                │                         │
3: ─────────────@^(1/8)─────────@^0.25─────@^0.5──H───
```

Cirq ayrıca QFT için bir iç metot bulundurmaktadır. Bu metot aşağıdaki şekilde kullanılabilir.

```
qft_operation = cirq.qft(*qubits, without_reverse=True)
qft_cirq = cirq.Circuit(qft_operation)
print(qft_cirq)
```

```
0: ──qft[norev]──
      │
1: ──#2──────────
      │
2: ──#3──────────
      │
3: ──#4──────────
```

3.2.3 Simülasyon

Kuantum fourier dönüşümünün etkisini anlamak için "5" rakamını aşağıdaki gibi kodlanmıştır. Fakat bu etki bir sonraki başlık olan ters kuantum fourier dönüşümünde gösterilecektir.

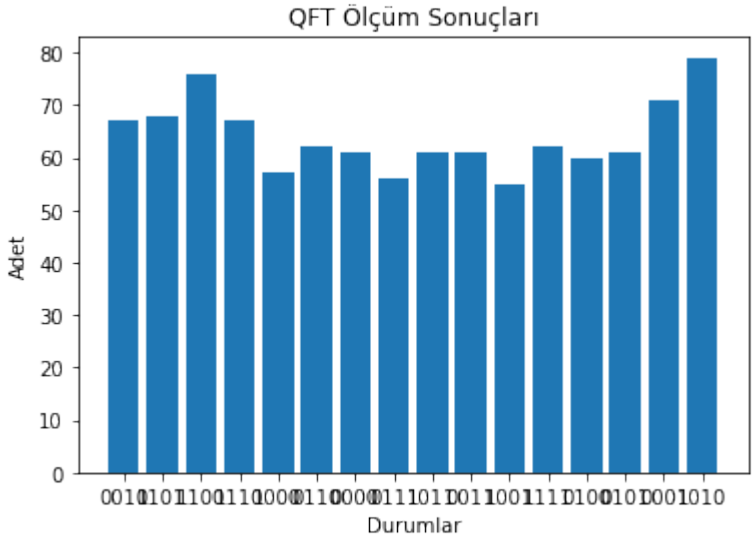
```
# 5 = bin(101). Bu nedenle 0. ve 2. kubitlere X uygulanır.
qft.append(cirq.X(qubits[0]))
qft.append(cirq.X(qubits[2]))
qft.append(cirq.measure(*qubits, key="M"))
print(qft)
```

```
sim = cirq.Simulator()
results = sim.run(qft, repetitions=1024)
```

QFT kullanarak sayı kodladığımız bu devre için Z bazında ölçüm sonuçları anlam ifade etmemektedir. Bir sonraki algoritma olan ters kuantum fourier dönüşümü ile etki incelenecektir. Aşağıdaki kod parçası ile 1024 örnekleme içerisinde ölçülen değerler gözlemlenebilir.

```
hist = dict()
for val in results.measurements["M"]:
    new_str = ""
    for s in val:
        new_str += str(s)
    try:
        hist[new_str] += 1
    except KeyError:
        hist[new_str] = 1
```

```
1 pyplot.bar(hist.keys(), hist.values())
2 pyplot.title("QFT Ölçüm Sonuçları")
3 pyplot.xlabel("Durumlar")
4 pyplot.ylabel("Adet")
```



3.3 Ters Quantum Fourier Dönüşümü

Bu algoritma klasikte sinyal işlemede ve veri sıkça kullanılmaktadır. Bir önceki tanıtılan kuantum fourier dönüşümü ile kodlanmış kubitleri çözmek için kullanılır. Bu bölümde kodlanacak olan ters kuantum fourier algoritması juptyter-notebooks klasörünün içerisinde **ters_qft.ipynb** dosyasında bulunmaktadır.

3.3.1 Kurulum

```
1 import cirq
2 import numpy as np
3 from matplotlib import pyplot
```

3.3.2 Adım 1: Devre Hazırlama

Bu adımda devre kuantum fourier dönüşümünde kullanılan kapıları tersten dizilerek tasarlanmaktadır.

```
# Bir Ters QFT Devresi oluşturur.
def make_qft_inverse(qubits):
    qreg = list(qubits)[::-1]
    while len(qreg) > 0:
        q_head = qreg.pop(0)
        yield cirq.H(q_head)
        for i, qubit in enumerate(qreg):
            yield (cirq.CZ ** (-1 / 2 ** (i + 1)))(
   qubit, q_head)
```

```
qubits = cirq.LineQubit.range(4)
tqft = cirq.Circuit(make_qft_inverse(qubits))
tqft.append(cirq.measure(*qubits, key="M"))
print(tqft)
```

```
0: ──────────────────────────────────@──────────────────@──────────@──────H──
                                     │                  │          │
1: ──────────────@───────────────────┼──────────@───────┼──────H───@^-0.5─────
                 │                   │          │       │
2: ──────@───────┼──────H────────────┼──────@^-0.5──────@^-0.25────────────────
         │       │                   │
3: ──H───@^-0.5──@^-0.25─────────────@^(-1/8)──────────────────────────────────
```

Cirq ayrıca Ters QFT için bir iç metot bulundurmaktadır. Bu metot aşağıdaki şekilde kullanılabilir.

```
iqft_operation = cirq.qft(*qubits, inverse=True,
    without_reverse=True)
iqft_cirq = cirq.Circuit(iqft_operation)
print(iqft_cirq)
```

```
0: ──qft[norev]^-1──
     │
1: ──#2─────────────
     │
2: ──#3─────────────
     │
3: ──#4─────────────
```

3.3.3 Kuantum Fourier ile Ters Kuantum Fourier

Bu adım bu algoritmanın bir adımı olmamakla beraber bir önceki bölümde gösterilen kuantum fourier dönüşümü algoritması ile beraber kullanıldığı bir bölümdür. Bu alt bölüm kuantum fourier algoritma kısmında devr-

eye kodlanan 5 rakamını ters kuantum fourier algoritması ile çözülmesini içermektedir.

```
qubits = cirq.LineQubit.range(4)

# 5 = bin(101). Bu nedenle 0. ve 2. kubitlere X uygulanır.
qft_cirq = cirq.Circuit()
qft_cirq.append(cirq.X(qubits[0]))
qft_cirq.append(cirq.X(qubits[2]))
qft_operation = cirq.qft(*qubits, without_reverse=True)
qft_cirq.append(qft_operation)
print(qft_cirq)
```

```
0: ──X──qft[norev]──
       │
1: ────────#2───────
       │
2: ──X──#3─────────
       │
3: ────────#4───────
```

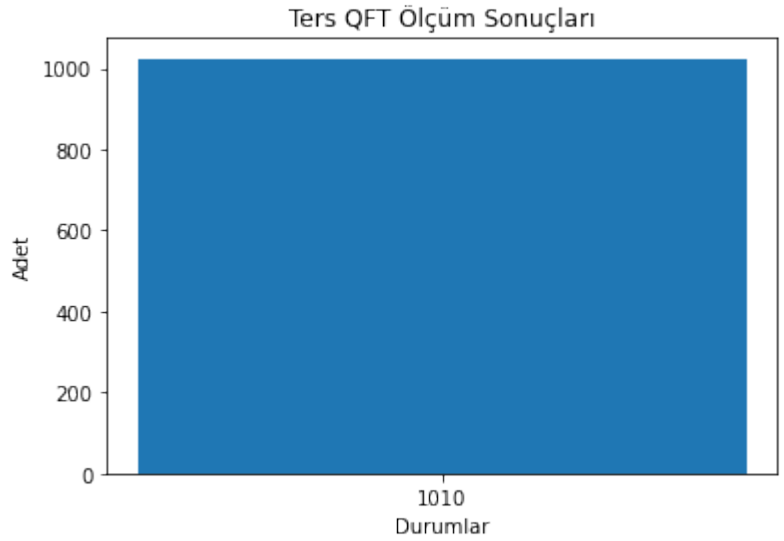
```
iqft_operation = cirq.qft(*qubits, inverse=True, without_reverse=True)
iqft_cirq = cirq.Circuit(iqft_operation)
qft_cirq.append(iqft_cirq)
qft_cirq.append(cirq.measure(*qubits, key="M"))
print(qft_cirq)
```

```
0: ──X──qft[norev]──qft[norev]^-1──M('M')──
       │            │              │
1: ────────#2───────#2─────────────M────────
       │            │              │
2: ──X──#3─────────#3─────────────M────────
       │            │              │
3: ────────#4───────#4─────────────M────────
```

3.3.4 Simülasyon

Bu devre kuantum fourier dönüşümü yardımıyla kodlanan bir değeri ters kuantum fourier dönüşümü kullanarak çözmektedir. Bu devrenin simülasyonu sonucunda ilk üç kubitin beklenen ölçüm değeri "$5 = (101)_2$".

```
sim = cirq.Simulator()
results = sim.run(qft_cirq, repetitions=1024)
hist = dict()
for val in results.measurements["M"]:
    new_str = ""
    for s in val:
        new_str += str(s)
    try:
        hist[new_str] += 1
    except KeyError:
        hist[new_str] = 1

pyplot.bar(hist.keys(), hist.values())
pyplot.title("Ters QFT Ölçüm Sonuçları")
pyplot.xlabel("Durumlar")
pyplot.ylabel("Adet")
```



Yukarıdaki görselden görüleceği üzere beklenen değer elde edilmiştir.

3.4 Faz Tahmini Algoritması

Faz tahmin algoritması, birimsel bir operatörün eşdeğerlerini kuantum fourier dönüşümü kullanarak tahmin eder. Bu bölümde kodlanacak olan faz tahmini algoritması juptyter-notebooks klasörünün içerisinde **faz_tahmin.ipynb** dosyasında bulunmaktadır.

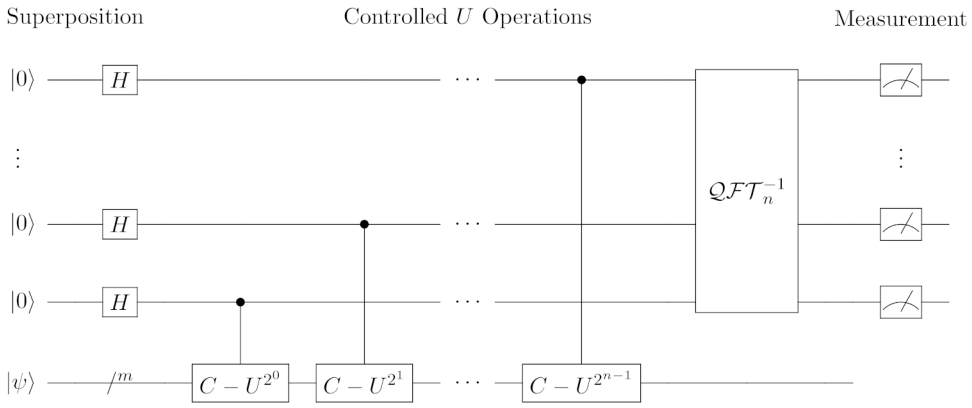


Fig. 3: Faz tahmini algoritmasının krokisi

3.4.1 Kurulum

```
import cirq
import numpy as np
from matplotlib import pyplot
```

```
# Bir Ters QFT Devresi oluşturur.
def make_qft_inverse(qubits):
    qreg = list(qubits)[::-1]
    while len(qreg) > 0:
        q_head = qreg.pop(0)
        yield cirq.H(q_head)
        for i, qubit in enumerate(qreg):
            yield (cirq.CZ ** (-1 / 2 ** (i + 1)))(
  qubit, q_head)
```

3.4.2 Adım 1: Gereken Değerler Seçilir

İlk adımda algoritma için gereken $theta(\theta)$ ve tahmin edilmesi istenen değerler belirlenmelidir. Ayrıca isteğe bağlı olarak ancilla kubitinin ilk durumu bir kapı yardımıyla değiştirilebilir.

```
theta = 0.15
max_nvals = 16
nvals = np.arange(1, max_nvals)
```

```
# Theta açısının faz kapısı
U = cirq.Z ** (2 * theta)

# Ancilla kubiti için başlangıç durumu (Kapı seçiniz)
prepare_eigenstate_gate = cirq.X
```

3.4.3 Adım 2: Hadamard

Her bir tahmin için devre oluşması gerekiyor fakat her devre için tüm adımlar aynı gerçekleşmektedir. Bu adımda devrenin tüm kubitleri süperpozisyon durumuna getirilir.

```
estimates = []
for n in nvals:
  # Her bir döngüde yeni devre gerekiyor.
  qubits = cirq.LineQubit.range(n)
  u_bit = cirq.NamedQubit('u')

  # Her kubite Hadamard uygulanır.
  phase_estimator = cirq.Circuit(cirq.H.on_each(*qubits
  ))
```

3.4.4 Adım 3: Kontrollü-Faz Kapıları

Bu adımda devreye kontrollü faz kapılarını algoritmanın görselinde görüleceği gibi farklı açılarda yerleştirilir.

```
...
# Kontrollü-U kapılarını belirli açılarda uygular.
for i, bit in enumerate(qubits):
  phase_estimator.append(cirq.ControlledGate(U).on(bit,
  u_bit) ** (2 ** (n - 1 - i)))
```

3.4.5 Adım 4: Ters QFT

Bu adımda ters kuantum fourier dönüşümü tersten yerleştirilir.

```
# Ters kuantum fourier dönüşümü tersten yerleştirilir.
phase_estimator.append(make_qft_inverse(qubits[::-1]))
```

3.4.6 Adım 5: Ancilla kubiti

Bu adımda ancilla kubiti için başlangıç durumuna getirilir.

```
# Ancilla kubiti için başlangıç durumu seçilir.
phase_estimator.insert(0, prepare_eigenstate_gate.on(
    u_bit))
```

3.4.7 Adım 6: Ölçümler

Bu adımda tüm kubitlere (ancilla hariç) ölçüm eklenir.

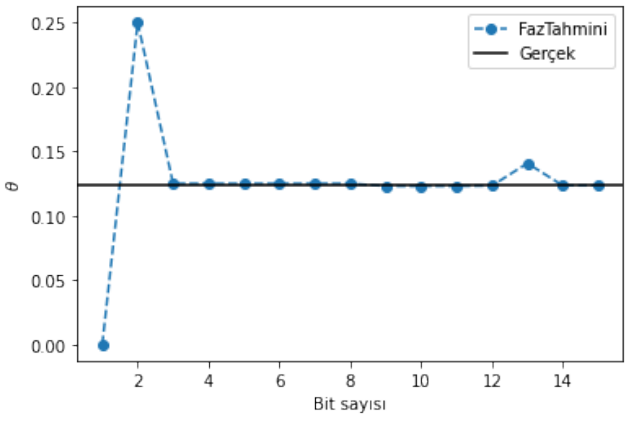
```
# Kubitlere ölçüm eklenir.
phase_estimator.append(cirq.measure(*qubits, key='m'))
```

3.4.8 Simülasyon

Bu adımda simülasyondan çıkan ölçüm sonuçlarına göre bir ortalama bir *theta*(θ) değeri hesaplanır.

```
# Ölçüm sonuçlarına göre bir ortalama bir theta değeri
    hesaplanır.
theta_estimates = np.sum(2**np.arange(n)*result.
    measurements['m'], axis=1)/2**n
estimates.append(theta_estimates[0])
```

```
pyplot.plot(nvals,estimates,"--o",label="FazTahmini")
pyplot.axhline(theta, label="Gerçek", color="black")
pyplot.xlabel("Bit sayısı")
pyplot.ylabel(r"$\theta$");
```



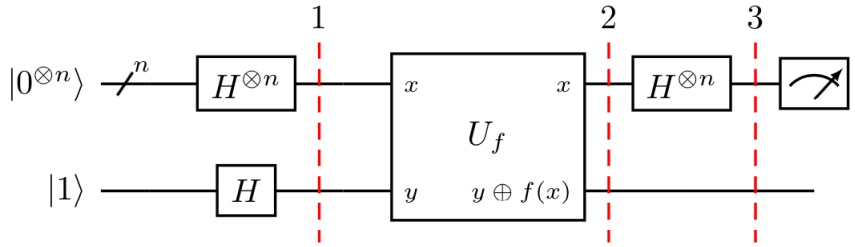
Simülasyon sonuçlarından görüleceği üzere faz tahmini algoritması bit sayısı bir değer ulaştıktan sonra gerçek veya gerçeğe yakın sonuçlar elde etmeyi başarmıştır.

3.5 Deutsch-Jozsa Algoritması

Bu algoritma tek bit ile çalışan Deutsch algoritmasının n bite uzatılmış versiyonudur. Bu algoritma girdi olarak gelen bitleri 0 veya 1' e dönüştüren gizli bir fonksiyona sahiptir.

Kuantum bilgisayarlarda kodlamak için bu algoritma $|x>|y>$ durumunu $|x>|y+f(x)>$ durumuna dönüşümünü yapan bir $U_f(oracle)$ fonksiyonundan faydalanır.

Bu bölümde kodlanacak olan Deutsch-Jozsa juptyter-notebooks klasörünün içerisinde **dj_alg.ipynb** dosyasında bulunmaktadır. Algoritma 4 adımda tamamlanmaktadır. Aşağıdaki şekilde bu adımları devre üzerinde gösterilmektedir.



3.5.1 Kurulum

Her algoritmada olduğu gibi ilk olarak gerekli kütüphaneleri içe aktarmakla başlanması gerekir.

```
1 import cirq
2 import numpy as np
3 from matplotlib import pyplot
```

3.5.2 Adım 1: Hadamard

Bu adımda ancilla kubitini $|1\rangle$ durumuna getirdikten sonra tüm kubitlere Hadamard kapısı uygulanarak süperpozisyon durumuna getirilir. Bu örnek için algoritma için kubit sayısı 4 olarak seçilmiştir.

```
1 n = 4
2 qubits = cirq.LineQubit.range(n + 1)
```

```
ancila = qubits[-1]
dj_circuit = cirq.Circuit()

# Ancilla kubitini |1> durumuna getir.
dj_circuit.append(cirq.X(ancila))

# Tüm kubitler süperpozisyon durumuna getirilir.
dj_circuit.append(cirq.H.on_each(*qubits))

print(dj_circuit)
```

```
0: ──H─────
1: ──H─────
2: ──H─────
3: ──H─────
4: ──X──H──
```

3.5.3 Adım 2: Oracle

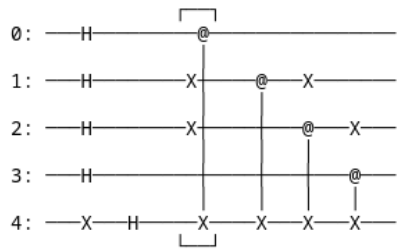
Kuantum bilgisayarlarda kodlamak için bu algoritma $|x>|y>$ durumunu $|x>|y + f(x)>$ durumuna dönüşümünü yapan bir $U_f(oracle)$ fonksiyonundan faydalanır. Oracle fonksiyonu sabit veya dengeli olarak iki farklı şekilde gerçekleşebilir. Aşağıdaki fonksiyon kullanılarak istenilen kubit sayısına sahip Oracle devresi tasarlanabilir.

```
def dj_oracle(case, n):
    # Bu devre n+1 kubite sahip: girdi kubit boyutu,
    # ve bir de çıktı kubit
    oracle_qc = cirq.Circuit()
    qubits_2 = cirq.LineQubit.range(n+1)

    # Dengeli oracle isteniyor ise:
    if case == "balanced":
        b = np.random.randint(1,2**n)
        # İkili stringi formatla.
        b_str = format(b, '0'+str(n)+'b')

        # Sonra, ilk X kapıları eklenir.
        # İkili stringteki her rakam bir
        # kubite denk geldiği için,
        # string 1 ise X kapısı gerekir.
        for qubit in range(len(b_str)):
```

```
18              if b_str[qubit] == '1':
19                  oracle_qc.append(cirq.X(qubits_2[qubit
    ]))
20
21          # Çıktı kubiti hedef olacak şekilde,
22          # tum kubitler ile CNOT uygulanir.
23          for qubit in range(n):
24              oracle_qc.append(cirq.CX(qubits_2[qubit],
    qubits_2[n]))
25
26          # Sonra, son X kapılarını eklenir.
27          for qubit in range(len(b_str)):
28              if b_str[qubit] == '1':
29                  oracle_qc.append(cirq.X(qubits_2[qubit
    ]))
30
31      # Sabit oracle isteniyor ise:
32      if case == "constant":
33      # Ilk olarak hangi sabit çıktı
34      # isteniyor rastgele secelim.
35          output = np.random.randint(2)
36
37          # Rasgele 1 gelirse, son kubite X uygulanır.
38          if output == 1:
39              oracle_qc.append(cirq.X(qubits_2[qubit]))
40      return oracle_qc
```

Bu metot aşağıdaki gibi çağrılarak istenilen bir devreye kolayca eklenebilir.

```
1 dj_circuit.append(dj_oracle("balanced", n),  range(n +
   1))
2 print(dj_circuit)
```



3.5.4 Adım 3: Tekrar Hadamard

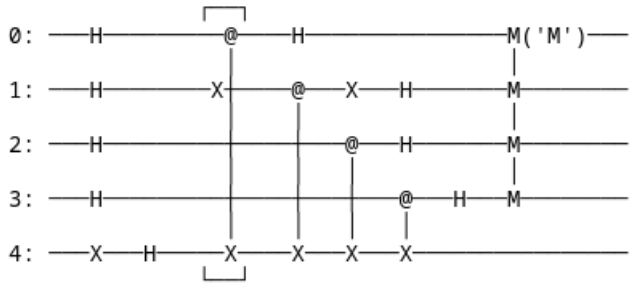
Bu adımda tüm kubitlere (ancilla dışında) Hadamard dönüşümü uygulanır.

```
qubits_to_hm = qubits[:-1]
dj_circuit.append(cirq.H.on_each(qubits_to_hm))
```

3.5.5 Adım 4: Ölçüm

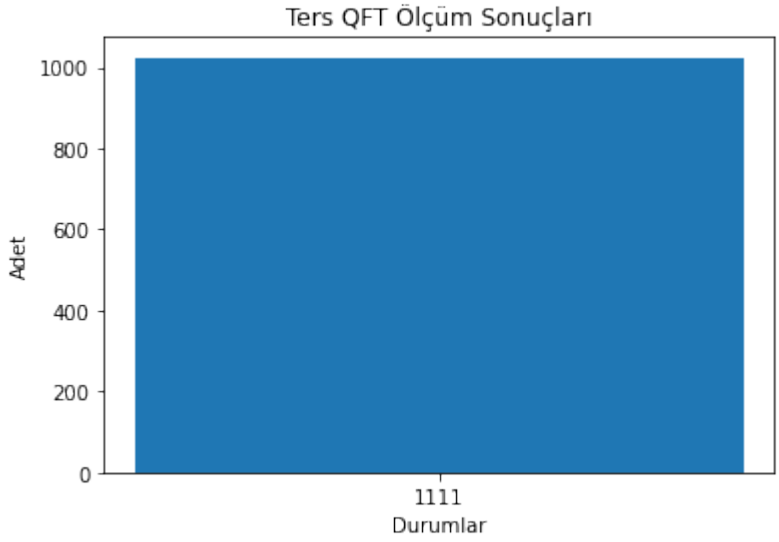
Bu algoritmanın son adımında tüm kubitler (ancilla dışında) ölçülür.

```
dj_circuit.append(cirq.measure(*qubits_to_hm, key="M"))
print(dj_circuit)
```



3.5.6 Simülasyon

```
sim = cirq.Simulator()
results = sim.run(dj_circuit, repetitions=1024)

hist = dict()
for val in results.measurements["M"]:
    new_str = ""
    for s in val:
        new_str += str(s)
    try:
        hist[new_str] += 1
    except KeyError:
        hist[new_str] = 1

pyplot.bar(hist.keys(), hist.values())
pyplot.title("Deutsch-Jozsa Algoritması")
pyplot.xlabel("Durumlar")
pyplot.ylabel("Adet")
```



3.6 Shor Algoritması

Shor algoritmasının amacı tekrarlayan serilerin periyotunu bulmaktadır. Klasikte bu işlem zaman bakımından üstel artan olduğu için büyük asal rakamlarda maliyetlidir. Fakat bu işlem kuantum bilgisayarlarda daha imkanlı olması günümüzde asal sayılar ile korunan internet güvenliği gibi alanları tehdit etmektedir. Bu bölümde kodlanacak olan shor algoritması juptyter-notebooks klasörünün içerisinde **shor_alg.ipynb** dosyasında bulunmaktadır.

3.6.1 Kurulum

Bu algoritma için fazladan Python içinde gelen bir kaç adet kütüphaneye ihtiyacımız bulunmaktadır.

```
1 import cirq
2 import numpy as np
3 from matplotlib import pyplot
4 from math import gcd
5 from numpy.random import randint
6 import pandas as pd
7 from fractions import Fraction
```

Her bir a ve N değeri için $U|y> = |aymodN>$ denkliğini sağlayan bir Oracle fonksiyonuna ihtiyacımız vardır. Bizler bu örnek için $a = 7$ ve $N = 15$

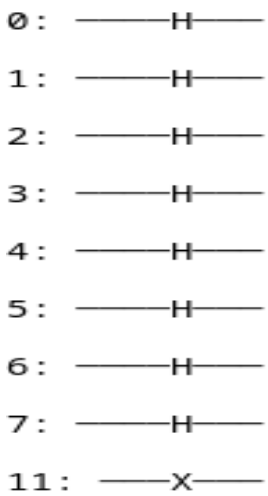
olarak alacağız. 8 adet sayma kubiti ve 15'i ikili tabanda temsil edebilecek 4 adet kayıt kubiti gerekmektedir.

```
n_count = 8  # sayma kubitler
a = 7
N = 15
# 8 sayma + 4 kayıt

all_qubits = cirq.LineQubit.range(n_count + 4)
count_qubits = all_qubits[:8]
register_qubits = all_qubits[8:]

shor_circuit = cirq.Circuit()
```

3.6.2 Adım 1: Hadamard

Bu adımda tüm sayma kubitlerine *Hadamard* uygulanır ve N=15 olacak şekilde kayıtçı kubitler başlatılır.

```
# Saymalara Hadamard uygula.
shor_circuit.append(cirq.H.on_each(*count_qubits))

# Kayıtçı 4 kubiti bin(15) olarak ayarla.
shor_circuit.append(cirq.X(register_qubits[3]))
print(shor_circuit)
```



3.6.3 Adım 2: AmodN Dönüşümü

Bu adımda 7mod15 için Oracle fonksiyonunu ekleyeceğiz. İlk olarak a değerlerini kontrol edilmesi gerekmektedir.

```
# Kayıtçılar için Amod15 devresi
def c_amod15(qubs, a, power):
    if a not in [2,7,8,11,13]:
        raise ValueError("'a' must
  be 2,7,8,11 or 13")

    U = cirq.Circuit()
```

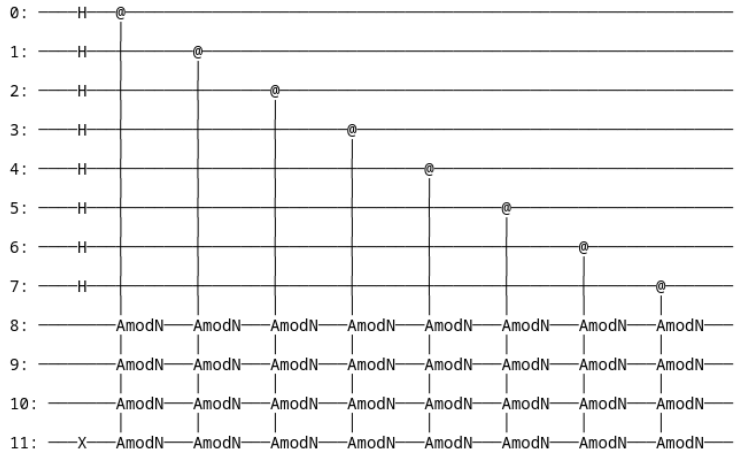
```
for iteration in range(power):
    if a in [2,13]:
    U.append(cirq.SWAP(qubs[0], qubs[1]))
    U.append(cirq.SWAP(qubs[1], qubs[2]))
      U.append(cirq.SWAP(qubs[2], qubs[3]))
  if a in [7,8]:
      U.append(cirq.SWAP(qubs[2], qubs[3]))
    U.append(cirq.SWAP(qubs[1], qubs[2]))
    U.append(cirq.SWAP(qubs[0], qubs[1]))
  if a == 11:
    U.append(cirq.SWAP(qubs[1], qubs[3]))
    U.append(cirq.SWAP(qubs[0], qubs[2]))
if a in [7,11,13]:
    for q in range(4):
    U.append(cirq.X(qubs[q]))
  return U
```

Üstteki bu metot kullanılarak *Amod15* için bir Cirq devresi oluşturulabilir. Fakat bu devrenin kullanılabilmesi için kapıya dönüştürmemiz gerekmektedir.

```
class amod15_Gate(cirq.Gate):
    def __init__(self, a, power):
        super(amod15_Gate, self)
        self.a = a
        self.power = power

    def _num_qubits_(self):
        return 4

    def _decompose_(self, qubits):
        yield c_amod15(qubits, self.a, self.power)

    def _circuit_diagram_info_(self, args):
        return ["AmodN"] * self.num_qubits()
```

Üstte *cirq.Gate* sınıfı kullanılarak devre veya birimsel matris kullanılarak bir kapı oluşturulabilir. Bunun en büyük avantajı bize kontrollü kapıların kullanılmasına olanak tanımasıdır. Her kapı bir *controlled*() özniteliğine sahiptir. Bu metot kullanılarak kapı veya dönüşüm için kontrol kubit sayısını ve kontrol kubit değerlerini değiştirebilir. Bu kapıyı aşağıdaki gibi kontrollü hale getirip örnekteki devreye eklenmelidir. Daha fazla bilgi almak için tıklayınız!

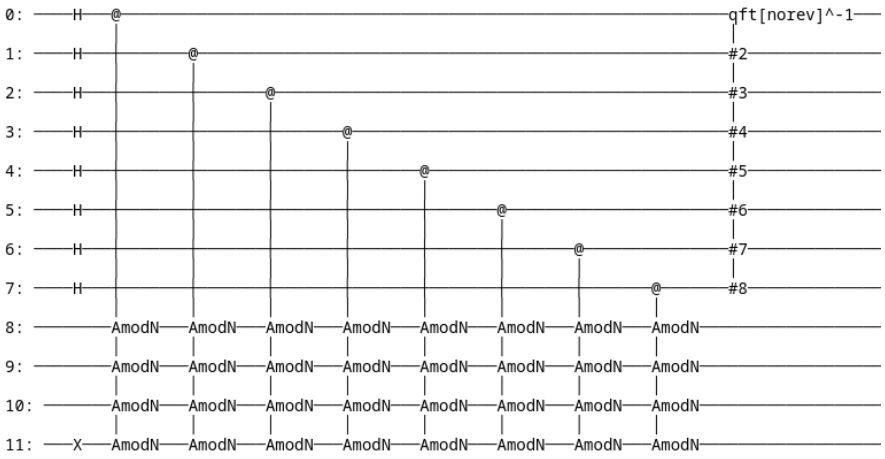
```
# Kontrollü-U kapılarını uygula.
for q in range(len(count_qubits)):
    shor_circuit.append(amod15_Gate(a, 2**q).controlled
  ().on(all_qubits[q], *register_qubits))
print(shor_circuit)
```



3.6.4 Adım 3: Ters Kuantum Fourier Dönüşümü

Bu adımda sayma kubitlerine ters kunatum fourier dönüşşümü uygulanır.

```
iqft_operation = cirq.qft(*count_qubits, inverse=True,
   without_reverse=True)
shor_circuit.append(iqft_operation)
print(shor_circuit)
```



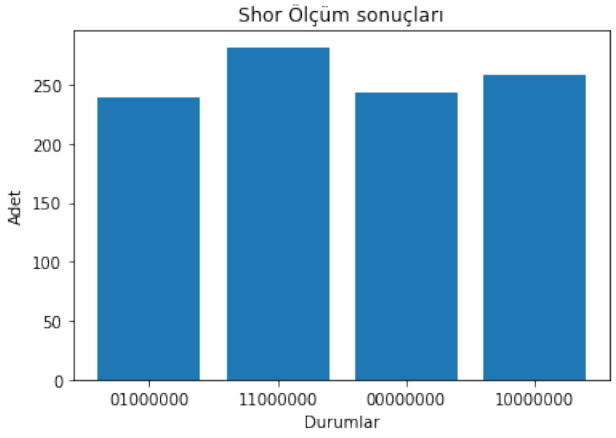
3.6.5 Adım 4: Ölçüm

Bu son adımda tüm sayma kubitleri ölçülmesi gerekmektedir.

```
1 shor_circuit.append(cirq.measure(*count_qubits, key="C"
   ))
2 print(shor_circuit)
```

3.6.6 Simülasyon

```
1  sim = cirq.Simulator()
2  results = sim.run(shor_circuit, repetitions=1024)
3  hist = dict()
4  for val in results.measurements["C"]:
5      new_str = ""
6      for s in val:
7          new_str += str(s)
8      try:
9          hist[new_str] += 1
10     except KeyError:
11         hist[new_str] = 1
12
13 pyplot.bar(hist.keys(), hist.values())
14 pyplot.title("Shor Ölçüm sonuçları")
15 pyplot.xlabel("Durumlar")
16 pyplot.ylabel("Adet")
```



Bu sonuçlar henüz bize bir bilgi vermiş değildir. Bu sonuçlardan anlam çıkarılması için klasik yöntemler kullanacağız.

3.6.7 Adım 5: Periyot Bulma

Bu adımda ilk olarak *pandas* kullanılarak çıkan sonuçların belirttiği fazları incelemekle başlayacağız.

```
rows, measured_phases = [], []
for output in hist:
    decimal = int(output, 2)  # İkilik string -> ondalık sayı
    phase = decimal/(2**n_count)  # Alakalı eşdeğeri bul
    measured_phases.append(phase)
    rows.append([f"{output}(bin) = {decimal:>3}(dec)",
                 f"{decimal}/{2**n_count} = {phase:.2f}"])
headers=["Kayıçı Değerleri", "Faz"]
df = pd.DataFrame(rows, columns=headers)
print(df)
```

	Kayıtçı Değerleri	Faz
0	01000000(bin) = 64(dec)	64/256 = 0.25
1	11000000(bin) = 192(dec)	192/256 = 0.75
2	00000000(bin) = 0(dec)	0/256 = 0.00
3	10000000(bin) = 128(dec)	128/256 = 0.50

Şimdi bulunan bu fazları *Fraction* kullanarak denk geldiği **r** periyotunu bulalım.

```
rows = []
for phase in measured_phases:
    frac = Fraction(phase).limit_denominator(15)
    rows.append([phase, f"{frac.numerator}/{frac.denominator}", frac.denominator])

headers=["Faz", "Kalan", "Tahmin: r"]
df = pd.DataFrame(rows, columns=headers)
print(df)
```

	Faz	Kalan	Tahmin: r
0	0.25	1/4	4
1	0.75	3/4	4
2	0.00	0/1	1
3	0.50	1/2	2

Şimdi bulduğumuz **r=4** periyotunun **N=15**'den küçük ortak bölenlerini bulalım.

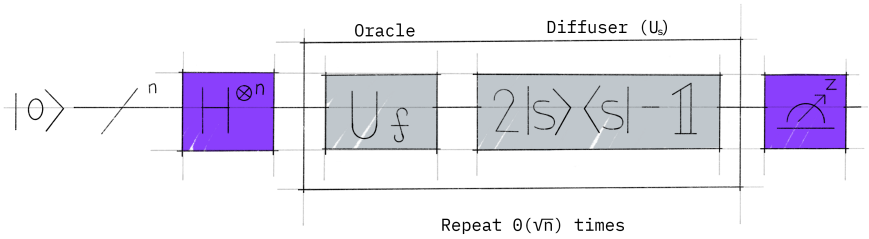
```
# En çok rastlanan priyot 4.
r = 4
guesses = [gcd(a**(r//2)-1, N), gcd(a**(r//2)+1, N)]
print(guesses)

... [3, 5]
```

Böylelikle 15 sayısının asal çarpanlarını kuantum bilgisayarlar üzerinde periyot hesaplayarak bulmuş olduk.

3.7 Grover Algoritması

Kuantum bilgisayarların bir diğer avantajı ise veritabanlarında hızlı aramalar gerçekleştiriyor olmasıdır. Grover algoritması ile çok hızlı şekilde veriler arasında aranan bir veriyi bulabiliriz. Bu bölümde kodlanacak olan Grover algoritması juptyter-notebooks klasörünün içerisinde **grover_alg.ipynb** dosyasında bulunmaktadır.

$U_w|x> = -(1)f(x)|x>$ Oracle fonksiyonu ile genlik arttıran Diffision bu algoritmanın omurgasını oluşturur.



3.7.1 Kurulum

```
import cirq
import numpy as np
from matplotlib import pyplot
import random
```

```
# Kubit sayısı.
nqubits = 2

# Devreyi ve ancilla kubitlerini ayarla.
qubits = cirq.LineQubit.range(nqubits)
ancilla = cirq.NamedQubit("Ancilla")
```

Grover algoritmasına başlamadan önce ile arayacağımız değerin ayarlanması gerekir. İsteğe bağlı rastgele veya belirli bir değer seçilebilir.

```
# Burada rastgele bir bit string işaretlenir.
# Grover ile bu değeri ayıyoruz.
xprime = [random.randint(0, 1) for _ in range(nqubits)]
print(f"Işaretlenen: {xprime}")

... Işaretlenen: [1, 1]
```

3.7.2 Adım 1: Hadamard

Bu adımda devreyi oluşturup her kubite *Hadamard* uygulayacağız.

```
circuit = cirq.Circuit()

# Her kubite Hadamard uygulanır.
circuit.append(cirq.H.on_each(*qubits))

# Ancilla kubiti |-> durumuna getirlir.
circuit.append([cirq.X(ancilla), cirq.H(ancilla)])
print(circuit)
```

```
0: ─────────H───────

1: ─────────H───────

Ancilla: ──X──H──
```

3.7.3 Adım 2: Oracle

Bu adımda devreye Oracle kara kutu metodu eklenmelidir.

```
def oracle():
    """ Bu metot ile Oracle devresi oluşturulur. """

    yield (cirq.X(q) for (q, bit) in zip(qubits, xprime
   ) if not bit)
    yield (cirq.TOFFOLI(qubits[0], qubits[1], ancilla))
    yield (cirq.X(q) for (q, bit) in zip(qubits, xprime
   ) if not bit)
```

3.7.4 Adım 3: Diffision

Bu adımda devreye bir diffision eklenmelidir.

```
def diffuser():
    # Diffisson oluşturulur.

    yield (cirq.H.on_each(*qubits))
    yield (cirq.X.on_each(*qubits))
    yield (cirq.H.on(qubits[1]))
    yield (cirq.CNOT(qubits[0], qubits[1]))
    yield (cirq.H.on(qubits[1]))
    yield (cirq.X.on_each(*qubits))
    yield (cirq.H.on_each(*qubits))
    return circuit
```

Adım 3 ve 4, $(\sqrt{\pi})$ kadar tekrarlanmalıdır.

```
# Gorver devresi oluşturulur.
for i in range(int(np.sqrt(nqubits))):
    circuit.append(oracle())
    circuit.append(diffuser())
```

3.7.5 Adım 4: Ölçüm

```
# Tüm girdi kubitleri ölçülür.
circuit.append(cirq.measure(*qubits, key="R"))
print(circuit)
```

```
0: ─────────H─────────@───H───X─────────@───X───H─────────M('R')──
                     │                 │                 │
1: ─────────H─────────@───H───X───H───X───H───X───H───M─────────
                     │
Ancilla: ──X───H───X──────────────────────────────────────────────
```

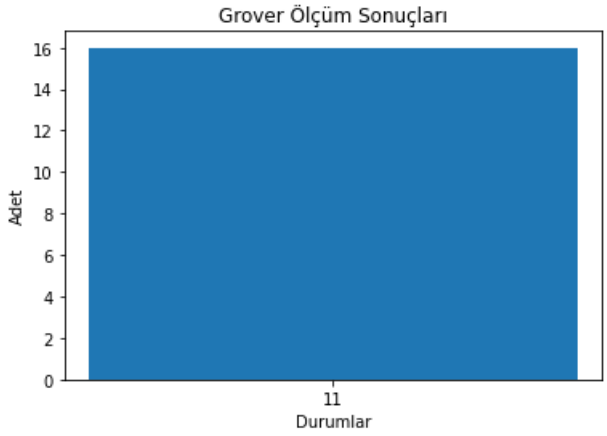
3.7.6 Simülasyon

```
def bitstring(bits):
    return "".join(str(int(b)) for b in bits)

# Devre simüle edilir.
simulator = cirq.Simulator()
result = simulator.run(circuit, repetitions=16)
```

```
hist = dict()
for val in result.measurements["R"]:
    new_str = ""
    for s in val:
        new_str += str(s)
    try:
        hist[new_str] += 1
    except KeyError:
        hist[new_str] = 1

pyplot.bar(hist.keys(), hist.values())
pyplot.title("Grover Ölçüm Sonuçları")
pyplot.xlabel("Durumlar")
pyplot.ylabel("Adet")

frequencies = result.histogram(key="R", fold_func=
 bitstring)
print('Sonuçlar:\n{}'.format(frequencies))

# Doğru mu kontrol et.
most_common_bitstring = frequencies.most_common(1)
 [0][0]
print("\nEn çok bulunan sonuç: {}".format(
 most_common_bitstring))
print("Doğru mu? {}".format(most_common_bitstring ==
 bitstring(xprime)))
```

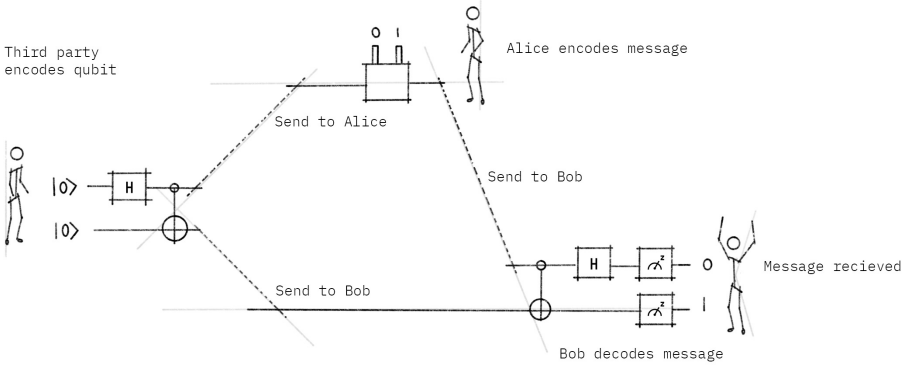
Grover algoritmasının simülasyonu sonucunda ölçülen değerlere bakılacağı zaman en yüksek tekrarlayan durumun en başta aramak için belirlediğimiz ikili sayı ile eşleşmesini beklemekteyiz. Bu örnek üzerinde aramak için rastgele seçtiğimiz ikili sayısı "11" olduğundan dolayı "$|11>$" ölçüm sonucunun domine olması beklentisi aşağıdaki şekil ile sağlanmıştır.

```
Sonuçlar:
Counter({'11': 16})

En çok bulunan sonuç: 11
Doğru mu? True
```



3.8 Süperyoğun Kodlama Algoritması

Bu algoritma ışınlanma algoritmasının bir gelişmiş versiyonudur. Bu algoritma bir Bell çifti ile 2 klasik biti göndermektedir. Aşağıdaki görselde algoritmanın nasıl çalıştığı vurgulanmaktadır. Bu bölümde kodlanacak olan süperyoğun kodlama algoritması juptyter-notebooks klasörünün içerisinde **sdc_alg.ipynb** dosyasında bulunmaktadır.



3.8.1 Kurulum

```
1 import cirq
2 import numpy as np
3 from matplotlib import pyplot
```

```
1 sdc_circuit = cirq.Circuit()
2 alice_pair = cirq.NamedQubit("alice_qubit")
3 bob_pair = cirq.NamedQubit("bob_qubit")
4
5 # Kodlanacak bitleri belirleyelim.
6 message = np.random.randint(0, 2, 2)
7 print("Kodlanacak Bitler: ", message)
8
9 ... Kodlanacak Bitler:  [1 1]
```

3.8.2 Adım 1: Bell çiftinin hazırlanması

Bu adımda üstteki görselde görüleceği üzere 3.taraf kişi tarafından bir Bell çifti oluşturulup Alice ile Bob'a gönderilmektedir.

```
1 # Bell çifti oluştur.
2 sdc_circuit.append(cirq.H(alice_pair))
3 sdc_circuit.append(cirq.CX(alice_pair, bob_pair))
4 print(sdc_circuit)
```

```
alice_qubit: ──H──@──
                  │
bob_qubit: ───────X──
```

3.8.3 Adım 2: Mesajın kodlanması

Bu adımda Alice, göndermek istediği 2 bite göre kendisine gelen kubite kapılar uygular ve Bob'a gönderdiği varsayılır. Eğer ilk bit "1" ise Z kapısı, ikinci kubit "1" ise X kapısı Alice'in Bell eşine eklenir. Alice kodladığı bu kubiti Bob'a gönderir.

```
1 # Bu adımda mesaj Alice'in Bell eşine kodlanır.
2 if message[1] == 1:
3     sdc_circuit.append(cirq.X(alice_pair))
4 if message[0] == 1:
5     sdc_circuit.append(cirq.Z(alice_pair))
6 print(sdc_circuit)
```

```
alice_qubit: ──H──@──X──Z──
                  │
bob_qubit: ───────X────────
```

3.8.4 Adım 3: Bob'un mesajı çözmesi

Bob bu adımda 2 kubite sahiptir. İlk kubit Bell çiftinin kendisine ait olan kubit ve diğeri ise Alice'in bir önceki adımda gönderdiği kubittir. Bob aşağıdaki kuantum kapıları uyguladıktan sonra ölçüm yapması gerekir.

```
sdc_circuit.append(cirq.CX(bob_pair, alice_pair))
sdc_circuit.append(cirq.H(bob_pair))
sdc_circuit.append(cirq.measure(alice_pair, bob_pair,
 key="M"))
print(sdc_circuit)
```

```
alice_qubit: ──H──@──X──Z──X──────M('M')──
                  │     │      │
bob_qubit: ───────X─────@──H──M────────
```

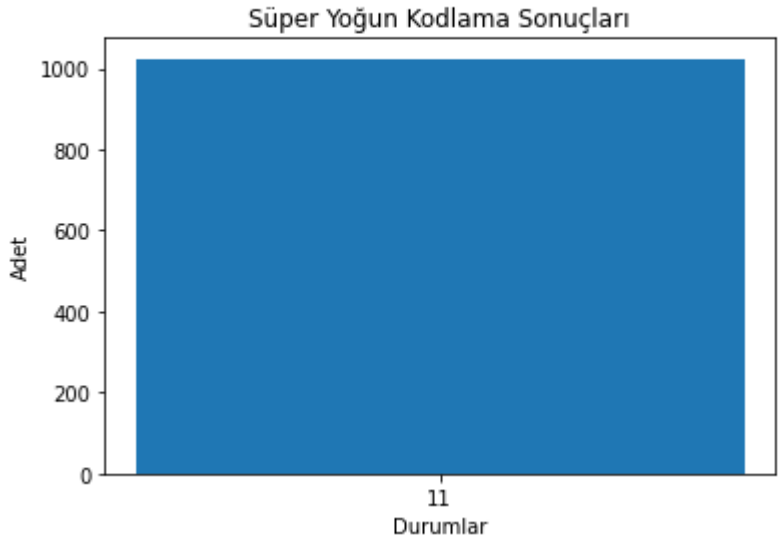
3.8.5 Simülasyon

```
# Devre simüle edilir.
simulator = cirq.Simulator()
result = simulator.run(sdc_circuit, repetitions=1024)
```

```
hist = dict()
for val in result.measurements["M"]:
    new_str = ""
    for s in val:
        new_str += str(s)
    try:
        hist[new_str] += 1
    except KeyError:
        hist[new_str] = 1

pyplot.bar(hist.keys(), hist.values())
pyplot.title("Süper Yoğun Kodlama Sonuçları")
pyplot.xlabel("Durumlar")
pyplot.ylabel("Adet")
```

```
# Mesajın doğruluğunu kontrol et.
st = ""
for i in message:
    st += str(i)
print("Doğru mu?{}".format(hist[st] > 1024 * 75 / 100))
```

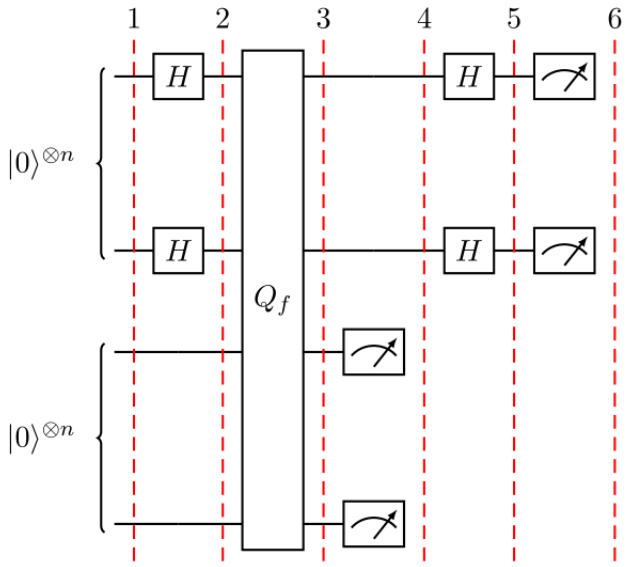
```
Doğru mu? True
```



Görüleceği üzere Alice'in kendi Bell eşine kodladığı **"11"** mesaj Bob tarafından başarılı şekilde çözülmüştür.

3.9 Simon Algoritması

Simon algoritması bir karakutu fonksiyonu kullanarak gizli bir mesajı kubitlere kodlamakta kullanılmaktadır. Bu nedenle bir *simon_oracle()* fonksiyonu kodlamakta ihtiyaç vardır. Bu bölümde kodlanacak olan Simon algoritması juptyter-notebooks klasörünün içerisinde **simon_alg.ipynb** dosyasında bulunmaktadır. Bu algoritmada kodlanması istenilen ikili değerin basamak sayısı kadar kayıtçı kubit ve ancilla kubit gerekmektedir. Bu algoritmanın adımları aşağıdaki görselde incelenebilir.



3.9.1 Kurulum

```
1 import cirq
2 import numpy as np
3 from matplotlib import pyplot
```

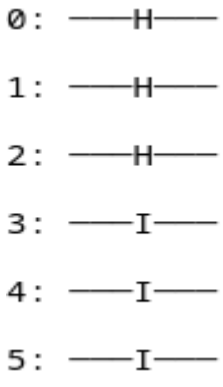
```
1 b = '110' # Mesaj
2 n = len(b) # 3*2 adet kubit gerekli
3
4 simon_circuit = cirq.Circuit()
5 all_qubits = cirq.LineQubit.range(n*2)
6 register_qubits = all_qubits[0:n]
7 ancilla_qubits = all_qubits[n:n*2]
```

3.9.2 Adım 1: Hadamard

Bu adımda kubitlerin ilk yarısına *Hadamard* uygulayarak süperpozisyon durumuna getirilmesi gerekir..

```
1 # Her kayıtçı kubitine Hadamard uygulanır.
2 simon_circuit.append(cirq.H.on_each(*register_qubits))
3 simon_circuit.append(cirq.I.on_each(*ancilla_qubits))
4 print(simon_circuit)
```

3.9.3 Adım 2: Simon Oracle



Bu adımda Simon algoritması için gereken Oracle karakutu fonksiyonu devreye eklenmesi gerekir.

```
# Bu adımda Simon algoritması için gereken Oracle karakutu fonksiyonunu eklenir.
b_t = b[::-1] # B'yi tersler
n = len(b_t)

# Dönüşüm: |x>|0> -> |x>|x>
for q in range(n):
    simon_circuit.append(cirq.CX(all_qubits[q], all_qubits[q+n]))
```

```
# 1:1 eşleşmeleri atla.
if '1' not in b:
    pass
else:
    i = b_t.find('1')  # b'de ilk sıfır olmayan bit.
    for q in range(n): # |x> -> |s.x> uygula eğer q_i = 1
        if b_t[q] == '1':
            simon_circuit.append(cirq.CX(all_qubits[i], all_qubits[q+n]))
print(simon_circuit)
```

3.9.4 Adım 3: Tekrar Hadamard

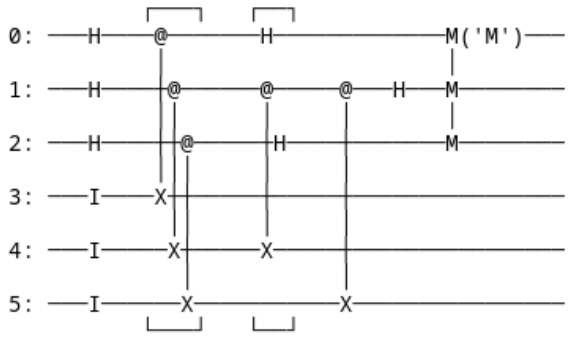
Bu adımda kubilerin ilk yarısı yani girdi kubitlere tekrardan *Hadamard* kapısı uygulanır.

```
# Bu adımda girdi kubitlere tekrar Hadamard uygulanır.
simon_circuit.append(cirq.H.on_each(*register_qubits))
```

3.9.5 Adım 4: Ölçüm

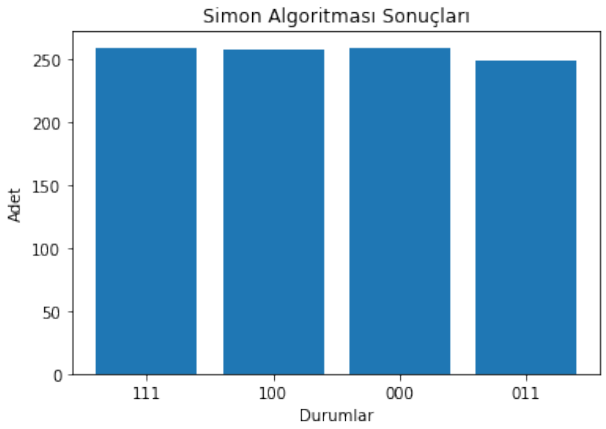
Simon algoritmasının bu son adımda girdi kubitlerine ölçüm eklenir.

```
simon_circuit.append(cirq.measure(*register_qubits, key="M"))
print(simon_circuit)
```



3.9.6 Simülasyon

```
simulator = cirq.Simulator()
result = simulator.run(simon_circuit, repetitions=1024)
hist = dict()
for val in result.measurements["M"]:
    new_str = ""
    for s in val:
        new_str += str(s)
    try:
        hist[new_str] += 1
    except KeyError:
        hist[new_str] = 1
pyplot.bar(hist.keys(), hist.values())
pyplot.title("Simon Algoritması Sonuçları")
pyplot.xlabel("Durumlar")
pyplot.ylabel("Adet")
```



Simon algoritması gizli mesaj **b**'yi bularak başarıyla tamamlanması için bir klasik adım ihtiyaç duymaktadır. Bu klasik adım ile her bir ölçüm sonucu için aşağıdaki denklemi çözmek gerekmektedir.

Using these results, we can recover the value of $b = 110$ by solving this set of simultaneous equations. For example, say we first measured `001`, this tells us:

$$\begin{aligned} b \cdot 001 &= 0 \\ (b_2 \cdot 0) + (b_1 \cdot 0) + (b_0 \cdot 1) &= 0 \\ (\cancel{b_2 \cdot 0}) + (\cancel{b_1 \cdot 0}) + (b_0 \cdot 1) &= 0 \\ b_0 &= 0 \end{aligned}$$

If we next measured `111`, we have:

$$\begin{aligned} b \cdot 111 &= 0 \\ (b_2 \cdot 1) + (b_1 \cdot 1) + (\cancel{0 \cdot 1}) &= 0 \\ (b_2 \cdot 1) + (b_1 \cdot 1) &= 0 \end{aligned}$$

Which tells us either:

$$b_2 = b_1 = 0, \quad b = 000$$

or

$$b_2 = b_1 = 1, \quad b = 110$$

```
1 def bdotz(b, z):
2     accum = 0
3     for i in range(len(b)):
4         accum += int(b[i]) * int(z[i])
5     return (accum % 2)
6
7 for z in counts:
8     print( '{}.{} = {} (mod 2)'.format(b, z, bdotz(b,z)
   ))
```

```
110.111 = 1 (mod 2)
110.100 = 1 (mod 2)
110.000 = 0 (mod 2)
110.011 = 0 (mod 2)
```

Yukarıdaki metot bizlere tüm $b.z$ iç çarpım sonuçlarını göstermektedir. Buradan **b**'nin **110** olduğunu doğrulayabiliriz.

4 Yüksek Boyuttan Algoritmalar

Cirq diğer bir çok simülatörün aksine yüksek boyuttan kubitler ile devre tasarlanmasına ve simüle etmesine olanak tanımaktadır. Yüksek boyuttan kubitler ku(d)it olarak adlandırılırlar. Boyutun derecesine göre orta harfler değişmektedir Örn: $3 boyut - kutrit$. Yüksek düzeyde *Hilbert uzayı* iki boyuta göre daha karmaşıktır. Her ne kadar bu bölümde bulunan algoritmalar bir önceki bölümde gösterilen iki boyutta yapılan algoritmalar ile aynı işleyişi paylaşsalar da kullanılan kapıların yüksek boyutta karşılıkları ifade edilmelidir.

4.1 Temel Dönüşüm Matrisleri

Yüksek boyuttan kodlama yapılmadan önce boyuta ait olan temel dönüşüm matrisleri $\mathbf{U}$ bulunmalıdır. Her boyut için n^2 adet dönüşüm bulunmaktadır. Öncelikle kolaylık olması açısından bazı değerleri değişkenlere atamak faydalı olacaktır. Aşağıdaki bu değerler $e^{\theta_1 \pi}$ olarak dönüşümlerde bulunacaktır.

```
1 # complex 0 + 1i
2 Imag = complex(0, 1)
3
4 A = (2 * pi * Imag / 4)
5 B = (4 * pi * Imag / 4)
6 C = (6 * pi * Imag / 4)
7 D = (8 * pi * Imag / 4)
8 E = (12 * pi * Imag / 4)
9 F = (18 * pi * Imag / 4)
```

Aşağıdaki kodlarda sırasıyla 3. ve 4. boyutta temel matris bulunmaktadır. Bu dönüşümler yüksek boyutta olan tüm algoritmaların kodlanmasında kullanılacaktır.

```
Unitaries = [
    [
        [np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]], dtype=complex)],
        [np.array([[0, 0, 1], [1, 0, 0], [0, 1, 0]], dtype=complex)],
        [np.array([[0, 1, 0], [0, 0, 1], [1, 0, 0]], dtype=complex)]
    ],
    [
        [np.array([[1, 0, 0], [0, A, 0], [0, 0, B]], dtype=complex)],
        [np.array([[0, 0, B], [1, 0, 0], [0, A, 0]], dtype=complex)],
        [np.array([[0, A, 0], [0, 0, B], [1, 0, 0]], dtype=complex)]
    ],
    [
        [np.array([[1, 0, 0], [0, B, 0], [0, 0, A]], dtype=complex)],
        [np.array([[0, 0, A], [1, 0, 0], [0, B, 0]], dtype=complex)],
        [np.array([[0, B, 0], [0, 0, A], [1, 0, 0]], dtype=complex)]
    ]
]
```

Fig. 4: 3. Boyutun birimsel dönüşüm matrisleri

```
Unitaries = [
    [
        [np.array([[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1]], dtype=complex)],
        [np.array([[0, 0, 0, 1], [1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0]], dtype=complex)],
        [np.array([[0, 0, 1, 0], [0, 0, 0, 1], [1, 0, 0, 0], [0, 1, 0, 0]], dtype=complex)],
        [np.array([[0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1], [1, 0, 0, 0]], dtype=complex)]
    ],
    [
        [np.array([[1, 0, 0, 0], [0, A, 0, 0], [0, 0, B, 0], [0, 0, 0, C]], dtype=complex)],
        [np.array([[0, 0, 0, C], [1, 0, 0, 0], [0, A, 0, 0], [0, 0, B, 0]], dtype=complex)],
        [np.array([[0, 0, B, 0], [0, 0, 0, C], [1, 0, 0, 0], [0, A, 0, 0]], dtype=complex)],
        [np.array([[0, A, 0, 0], [0, 0, B, 0], [0, 0, 0, C], [1, 0, 0, 0]], dtype=complex)]
    ],
    [
        [np.array([[1, 0, 0, 0], [0, B, 0, 0], [0, 0, D, 0], [0, 0, 0, E]], dtype=complex)],
        [np.array([[0, 0, 0, E], [1, 0, 0, 0], [0, B, 0, 0], [0, 0, D, 0]], dtype=complex)],
        [np.array([[0, 0, D, 0], [0, 0, 0, E], [1, 0, 0, 0], [0, B, 0, 0]], dtype=complex)],
        [np.array([[0, B, 0, 0], [0, 0, D, 0], [0, 0, 0, E], [1, 0, 0, 0]], dtype=complex)]

    ],
    [
        [np.array([[1, 0, 0, 0], [0, C, 0, 0], [0, 0, E, 0], [0, 0, 0, F]], dtype=complex)],
        [np.array([[0, 0, 0, F], [1, 0, 0, 0], [0, C, 0, 0], [0, 0, E, 0]], dtype=complex)],
        [np.array([[0, 0, E, 0], [0, 0, 0, F], [1, 0, 0, 0], [0, C, 0, 0]], dtype=complex)],
        [np.array([[0, C, 0, 0], [0, 0, E, 0], [0, 0, 0, F], [1, 0, 0, 0]], dtype=complex)]
    ]
]
```

Fig. 5: 4. Boyutun birimsel dönüşüm matrisleri

4.1.1 Dönüşümleri Uygulama

Cirq içerisinde yüksek boyuttan Bu temel dönüşüm matrislerini kod içinde kullanılmadan önce bir operatör olarak tanımlamak gerekmektedir. Daha önce iki boyutlu algoritmalar içinde de kullandığımız **cirq.Gate()** sınıfını kullanarak özel kapılar üretmek gerekmektedir. Aşağıdaki kod parçası özel bir matris sınıfını başlatır.

```
class CustomGate(cirq.Gate):
    def __init__(self):
      super().__init__()
```

Bu sınıfın bir çok alt metodu ve özniteliği bulunmaktadır. Bunlardan en önemli ve doldurulması gerekenler aşağıdaki gibi sıralanabilir.

```
# Kapı biçimi
def _qid_shape_(self):
      return 3,
```

Kapı biçimi metodu olan $\_qid\_shape\_(self)$, kapının uygulanabileceği kudit(ler)in biçimini ifade eder. *Tupple* döndüren bu metot için tuple içeriği uygulanmak istenen kudit ile aynı boyutta ve uzunluğu ise uygulanmak istenen kudit(ler) kadar olmalıdır.

```
# Matris dönüşümü
  def _unitary_(self):
      return np.array([
      ... ])
```

Matris açılımı olan $\_unitary\_(self)$, oluşturulan kapının matrisini geri döndürmelidir. Bu matris **complex** tipinde bir *numpy* dizisi olmalıdır. Matrisin boyutları uygulanmak istenen kudit(ler)in boyutuna ve adetine uygun olmalıdır.

```
# Diyagram bilgisi
  @staticmethod
  def _circuit_diagram_info_(args):
      return 'U'
```

Diyagram bilgisini tutan $\_circuit\_diagram\_info(args)$, *print* ile çizilen devre üzerinde kullanılan bu devrenin nasıl çizileceğini göstermektedir.

```
# Çözüm kapıları
  @staticmethod
  def _decompose_(self, qubits):
      yield ...
      yield ...
```

Özel kapıyı başka kapılar biçiminden ifade eden $\_decompose\_()$, *yield* anahtar kelimesi ile kullanılmaktadır. Bu kapıyı ifade eden diğer kapılar *yield* anahtar kelimesinin sağ el tarafında bulunmalıdır. $\_decompose\_()$ ile $\_unitary\_()$ aynı anda **kullanılmamalıdır**.

```
def _num_qubits_():
      return 2
```

Kapının uygulandığı kuditlerin sayısını belirleyen metot olan $\_num\_qubits\_(self)$ geriye *integer* değerinde bir sayı döndürmektedir. Bu metot olmadığı zaman $\_qid\_shape\_()$ veya $\_unitary\_()$ bulunduğunda otomatik hesaplanır.

4.1.2 Temel Dönüşüm Kapısı

Temel dönüşüm matrislerini Cirq içerisinde kullanmak için kapıya dönüştürülmesi gerekmektedir. Aşağıdaki sınıf ile yukarıda verilen Unitary(U) kapılar kullanılarak bu sağlanmış olur.

```
# 3 Boyut için.
class CustomUGate(cirq.Gate):
    def __init__(self, unitary_matrix):
        self.unitary_matrix = unitary_matrix
        super().__init__()

    def _qid_shape_(self):
        return 3,

    def _num_qubits_():
        return 1

    def _unitary_(self):
        return self.unitary_matrix

    @staticmethod
    def _circuit_diagram_info_(args):
        return 'U'

    @property
    def transform_matrix(self) -> np.ndarray:
        return self._unitary_()

    def __str__(self):
        return str(self._unitary_())
```

Bu sınıf $\_\_init\_\_()$ üretici metodundan anlaşılacağı üzere bir matris alır ve alakalı bir kapı oluşturur. Daha yüksek boyutlarda kullanılmak istendiğinde $\_qid\_shape\_()$ değiştirilip alakalı temel dönüşüm matrisi verilmelidir.

4.1.3 Temel ID Kapısı

Daha öncesinde verilen Unitary(U) matrisleri kullanılarak yüksek boyutlarda ID kapısı oluşturulabilir.

```
class QutritIdle(cirq.Gate):
    def _qid_shape_(self):
        return 3,

    @staticmethod
    def _unitary_():
        return Unitaries[0][0][0]

    @staticmethod
    def _circuit_diagram_info_(args):
        return 'I'

    @property
    def transform_matrix(self) -> np.ndarray:
      return self._unitary_()

    def __str__(self):
        return str(self._unitary_())
```

4.1.4 Hadamard Kapısı

Aşağıdaki kod parçasında 3. boyutta Hadarmad kapısının oluşturulması gösterilmektedir.

```
class QutritHadamard(cirq.Gate):
    def _qid_shape_(self):
        return 3,

    @staticmethod
    def _unitary_():
        arr = np.array([
            [1, 1, 1],
            [1, exp(A), exp(B)],
            [1, exp(B), exp(A)]
        ], dtype=complex)
        arr *= 1 / pow(3, 0.5)
        return arr
```

```
    @staticmethod
    def _circuit_diagram_info_(args):
        return 'H'

    @property
    def transform_matrix(self) -> np.ndarray:
        return self._unitary_()

    def __str__(self):
        return str(self._unitary_())
```

4.1.5 Hermitik Hadamard Kapısı

Aşağıdaki kod parçasında 3. boyutta hermitik Hadarmad kapısının oluşturulması gösterilmektedir.

```
class QutritHadamardHermitik(cirq.Gate):
    def _qid_shape_(self):
        return 3,

    @staticmethod
    def _unitary_():
        return QutritHadamard().transform_matrix.
   conjugate().T

    @staticmethod
    def _circuit_diagram_info_(args):
        return 'Ht'

    @property
    def transform_matrix(self) -> np.ndarray:
        return self._unitary_()

    def __str__(self):
        return str(self._unitary_())
```

⚠ Hermitik işlemi aslında bir matrisin eşleniğinin tersi olarak ifade edilmektedir. Bu nedenle *_unitary_()* metoduna bir üstteki Hadamard kapısının matrisi tekrardan üstte görüldüğü gibi kullanılabilir.

4.1.6 Kontrollü Not Kapısı

Aşağıdaki kod parçasında 3. boyutta hermitik kontrollü-not kapısının oluşturulması gösterilmektedir.

```
class QutritCNOT(cirq.Gate):
    def _qid_shape_(self):
        return 3, 3,

    @staticmethod
    def _unitary_():
        arr = np.array([
            [1, 0, 0, 0, 0, 0, 0, 0, 0],
            [0, 1, 0, 0, 0, 0, 0, 0, 0],
            [0, 0, 1, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 1, 0, 0, 0],
            [0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 0, 0, 1, 0],
            [0, 0, 0, 0, 0, 0, 0, 0, 1],
            [0, 0, 0, 0, 0, 0, 1, 0, 0]
        ], dtype=complex)
        return arr

    @property
    def transform_matrix(self) -> np.ndarray:
        return self._unitary_()

    @staticmethod
    def _circuit_diagram_info_(args):
        return protocols.CircuitDiagramInfo(
            wire_symbols=('@', 'X'))

    def __str__(self):
        return str(self._unitary_())
```

⚠ Kontrollü-Not kapısı ilgili Unitary(U) dönüşüme kontrol eklenerek de yapılabilir. Hermitik kontrollü-Not kapısı ise *_unitary_*() aşağıdaki şekilde değiştirilerek yapılabilir.

```
  @staticmethod
  def _unitary_():
      return QutritCNOT().transform_matrix.conjugate().
T
```

4.2 Yüksek Boyutta Işınlama Algoritması

Bu algoritma iki boyutta olduğu gibi aynı aşamalar ile devam etmektedir. Bu bölümde kodlanacak olan yüksek boyutta ışınlama algoritması 3. ve 4. boyutta yapılacak olup juptyter-notebooks klasörünün içerisinde **hdim_tp_alg.ipynb** dosyasında bulunmaktadır.

4.2.1 Kurulum

Yüksek boyutta kodlama için gereken e sayısını, üst alma $pow()$ ve π değerlerini de içe aktarmak gerekmektedir.

```
from cmath import exp
from math import pi, pow

import cirq
import numpy as np
from cirq import protocols
from matplotlib import pyplot
```

4.2.2 Devre Tasarımı

Yüksek boyutta ışınlama devresi iki boyutta yapılan ile aynı adımlar bulundurmakta fakat yüksek boyuttan kapılar ile kodlanması gerekmektedir.

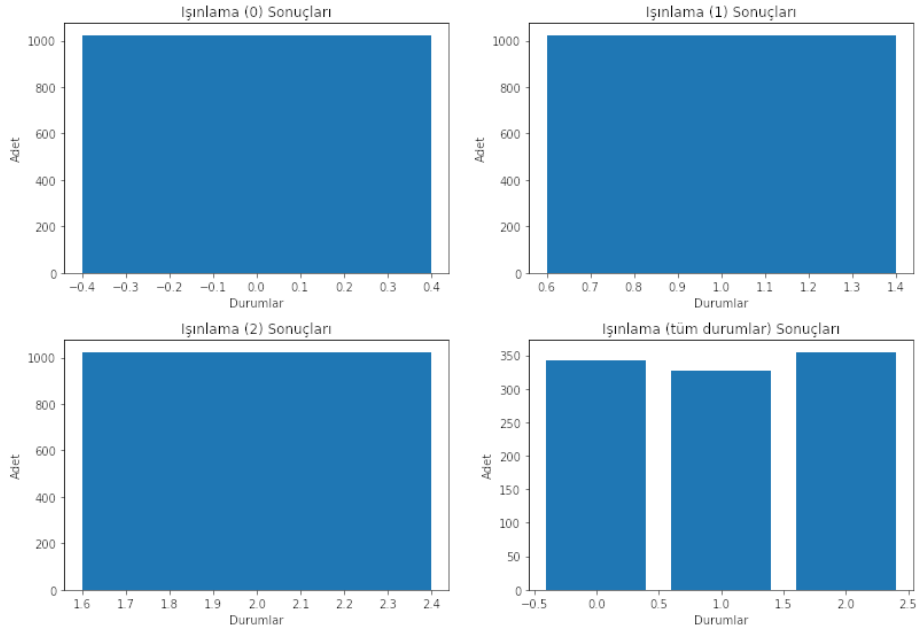
```
def teleportation_test(durum, shots):
    qI, qA, qB = cirq.LineQid.range(3, dimension=3)
    simulator = cirq.Simulator()

    # Başlangıç durumu seç
    if durum == "0":
        first = QutritIdle()
    elif durum == "1":
        first = CustomUGate(Unitaries[0][1][0])
    elif durum == "2":
        first = CustomUGate(Unitaries[0][2][0])
    elif durum == "hepsi":
        first = QutritHadamard()
    else:
        return
```

```
    # Devreyi tasarla.
    circuit = cirq.Circuit()
    circuit.append(first.on(qI))
    circuit.append(QutritHadamard().on(qA))
    circuit.append(QutritCNOT().on(qA, qB))
    circuit.append(QutritCNOTHermitik().on(qI, qA))
    circuit.append(QutritHadamard().on(qI))

     # Bu özel kapı iki tenary kontrollü kapıların
birleşimidir.
    circuit.append(SpecialGate().on(qI, qA, qB))
    circuit.append(cirq.measure(qB, key='qB'))
    print(circuit)

    # Simüle et ve sonuçları döndür.
    result = simulator.run(circuit, repetitions=shots)
    unique, counts = np.unique(result.measurements['qB'
], return_counts=True)
    new = dict(zip(unique, counts))
    return new
```

```
0 (d=3): ──I──────@'──H──S──────────
                  │     │
1 (d=3): ──H──@───X─────S──────────
              │         │
2 (d=3): ─────X─────────S──M('qB')──
```

⚠ Devrenin sonunda bulunan özel kapı Alice'in ölçüm sonuçlarını Bob'a ilettiğini zaman Bob'un yapması gereken dönüşümü içermektedir. Cirq klasik değişken kontrollü kapıların kodlanmasına izin vermediği için bu özel kapı aynı işlemi yapan 3 adet tenary kontrollü kapının matris çarpımını içermektedir.

4.2.3 Simülasyon



Yüksek boyutta ışınlama algoritması her boyutta benzer işlemler ile çalışır. Üstteki örneğin içerdiği *.ipynb* dosyasında 4. boyutta yapılan ışınlama algoritması da bulunmaktadır.

4.3 Yüksek Boyutta Kuantum Fourier Dönüşümü

Bu algoritma iki boyutta olduğu gibi aynı aşamalar ile devam etmektedir. Bu bölümde kodlanacak olan yüksek boyutta fourier dönüşümü 3. ve 4. boyutta yapılacak olup juptyter-notebooks klasörünün içerisinde **hdim_qft.ipynb** dosyasında bulunmaktadır.

4.3.1 Kurulum

Yüksek boyutta kodlama için gereken e sayısını, üst alma $pow()$ ve π değerlerini de içe aktarmak gerekmektedir. Bunun yanı sıra yüksek boyuttan kontrollü-faz kapısı işlemi için özel bir kapı tasarlamak da gerekmektedir.

```
from cmath import exp
from math import pi, pow

import cirq
import numpy as np
from cirq import protocols
from matplotlib import pyplot
```

```
class ControlledQutritPhaseGate(cirq.Gate):
    def __init__(self, root):
        self._root = root

    def _qid_shape_(self):
        return 3, 3,

    def _unitary_(self):
        arr = np.array([
            [1,0,0,0,0,0,0,0,0],
            [0,1,0,0,0,0,0,0,0],
            [0,0,1,0,0,0,0,0,0],
            [0,0,0,1,0,0,0,0,0],
            [0,0,0,0,exp(A/self._root),0,0,0,0],
            [0,0,0,0,0,exp(B/self._root),0,0,0],
            [0,0,0,0,0,0,1,0,0],
            [0,0,0,0,0,0,0,exp(A/self._root),0],
            [0,0,0,0,0,0,0,0,exp(B/self._root)]
        ], dtype=complex)
        return arr

    @staticmethod
    def _circuit_diagram_info_(args):
        return protocols.CircuitDiagramInfo(
            wire_symbols=('@', 'Z'))

    @property
    def transform_matrix(self) -> np.ndarray:
        return self._unitary_()

    def __str__(self):
        return str(self._unitary_())
```

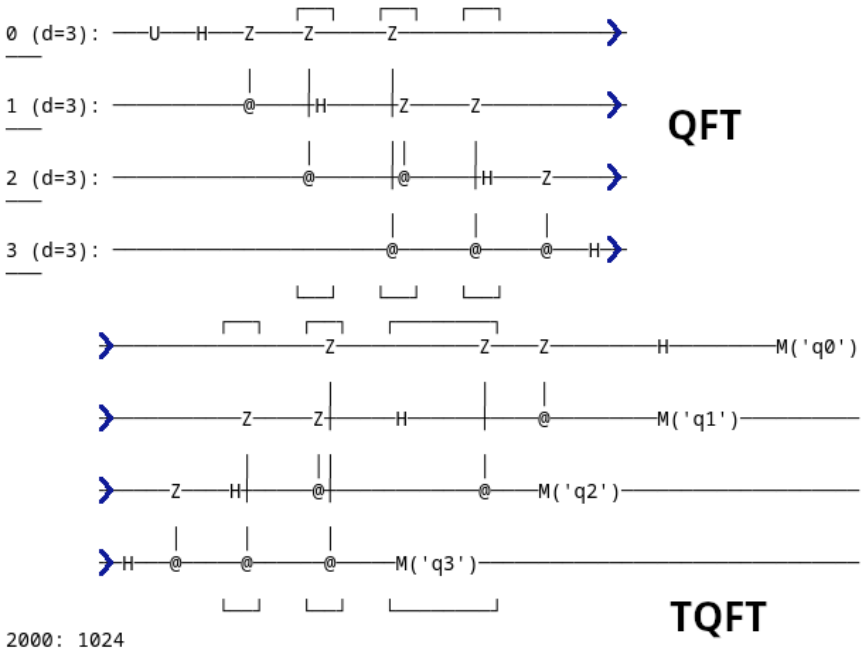
⚠ Bu kapı yüksek boyutta faz kapısını kontrollü hale getirerek de yapılabilir.

4.3.2 Devre Tasarımı

```
def test(shots, tersi=False):
    simulator = cirq.Simulator()
    q_0, q_1, q_2, q_3 = cirq.LineQid.range(4, 3)

    circuit = cirq.Circuit()
    circuit.append(CustomUGate(Unitaries[0][1][0]).on(
  q_0))
    circuit.append(QutritHadamard().on(q_0))
    circuit.append(ControlledQutritPhaseGate(1).on(q_1,
   q_0))
    circuit.append(ControlledQutritPhaseGate(2).on(q_2,
   q_0))
    circuit.append(ControlledQutritPhaseGate(4).on(q_3,
   q_0))
    circuit.append(QutritHadamard().on(q_1))
    circuit.append(ControlledQutritPhaseGate(1).on(q_2,
   q_1))
    circuit.append(ControlledQutritPhaseGate(2).on(q_3,
   q_1))
    circuit.append(QutritHadamard().on(q_2))
    circuit.append(ControlledQutritPhaseGate(1).on(q_3,
   q_2))
    circuit.append(QutritHadamard().on(q_3))
```

```
 # Tersi bayrağı var ise Ters QFT' de ekle.
    if tersi:
        circuit.append(QutritHadamard().on(q_3))
        circuit.append(ControlledQutritPhaseGate(1).on(
  q_3, q_2))
        circuit.append(QutritHadamard().on(q_2))
        circuit.append(ControlledQutritPhaseGate(2).on(
  q_3, q_1))
        circuit.append(ControlledQutritPhaseGate(1).on(
  q_2, q_1))
        circuit.append(QutritHadamard().on(q_1))
        circuit.append(ControlledQutritPhaseGate(4).on(
  q_3, q_0))
        circuit.append(ControlledQutritPhaseGate(2).on(
  q_2, q_0))
        circuit.append(ControlledQutritPhaseGate(1).on(
  q_1, q_0))
        circuit.append(QutritHadamard().on(q_0))
```

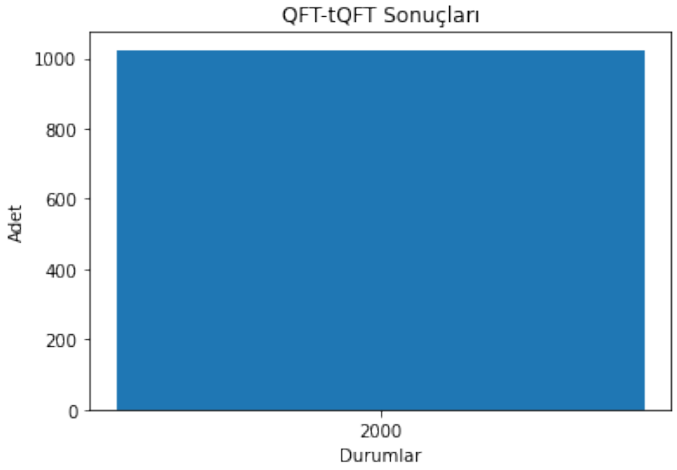
```
1    # Ölçümleri de ekleyelim.
2      circuit.append(cirq.measure(q_0, key="q0"))
3      circuit.append(cirq.measure(q_1, key="q1"))
4      circuit.append(cirq.measure(q_2, key="q2"))
5      circuit.append(cirq.measure(q_3, key="q3"))
6      results = simulator.run(circuit, repetitions=shots)
7      print(circuit)
8      return results
```




```
2000: 1024
```

Yukarıdaki *test*() fonksiyonuna bakacak olursak, 3 boyutta 4 kuditlik bir devre oluşturuluyor. Daha sonra ilk kutrit üzerinde Unitary[0][1] matris dönüşümü uygulanarak $54(2000)_3$ sayısı kutritlere kodlanıyor. Daha sonra ardından yüksek boyutta ters kuatum fourier dönüşümü devreye ekleniyor ve ölçüm yapılıyor.

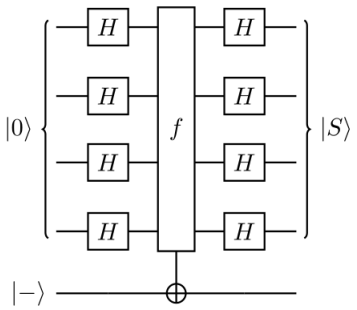
4.3.3 Simülasyon



Yuklarıdaki şekilde görüleceği üzere yüksek boyutta kuantum fourier ile kodlanan $54(2000)_3$ değerine yüksek boyutta ters kuantum fourier dönüşümü ile elde etmiş olduk. 4. boyutta örneği *.ipynb* üzerindeki örnekte devam etmektedir.

4.4 Yüksek Boyutta Bernstain-Vazirani Algoritması

Bu algoritma iki boyutta alogoritmalar bölümde ele aldığımız Deutsch-Jozsa algoritmasının bir uzantısı olarak görülebilir. Bu eklenti Deutsch-Jozsa algoritmasının yetersiz kaldığı karmaşık problemler için tanıtılmıştır. Bu bölümde kodlanacak olan yüksek boyutta Bernstain-Vazirani algoritması 3. ve 4. boyutta yapılacak olup juptyter-notebooks klasörünün içerisinde **bern_vazi.ipynb** dosyasında bulunmaktadır.



4.4.1 Kurulum

Bu algoritma için bir kutrit faz kapısına ihtiyaç olmaktadır. Bu kapı aşağıdaki gibi kodlanabilir.

```
class QutritPhaseGate(cirq.Gate):
    def __init__(self, root):
        self._root = root

    def _qid_shape_(self):
        return 3,

    def _unitary_(self):
        arr = np.array([
            [1, 0, 0],
            [0, exp(A / self._root), 0],
            [0, 0, exp(B / self._root)]
        ], dtype=complex)
        return arr

    @staticmethod
    def _circuit_diagram_info_(args):
        return "Z"

    @property
    def transform_matrix(self) -> np.ndarray:
        return self._unitary_()

    def __str__(self):
        return str(self._unitary_())
```

Devredeki kutritlerin sayısı gizli değerin uzunluğu kadar olmalıdır.

```
dimension = 3

# Gizli numarayı incele.
for s in secret:
  if int(s) >= dimension:
      print("Gizli numarada içerisinde kuditlerin
   boyuttan büyük rakam olamaz!")
      return

# Quditleri oluştur.
qudits = cirq.LineQid.range(secret.__len__() + 1,
   dimension=dimension)
last_qudit = qudits[-1]
```

4.4.2 Adım 1: Hadamard

Bu algoritmaya ilk olarak tüm kutritle Hadamard eklenerek başlanmaktadır.

```
# Hepsine Hadamart uygula.
circuit = cirq.Circuit()
for i in range(secret.__len__()):
    circuit.append(QutritHadamard().on(qudits[i]))
circuit.append(QutritHadamard().on(last_qudit))
```

4.4.3 Adım 2: Aktarım Devresi Tasarla

Bu adımda bir Oracle karakutu fonksiyonu eklenmektedir. Kudite denk gelen gizli mesajın rakam değerine göre yüksek boyutlu kontrollü-not veya hermitik kontrollü-not eklenmektedir.

```
# Son qudite boyutun faz kapısını uygula.
circuit.append(QutritPhaseGate(root=1).on(last_qudit))

# Aktarım devresi.
secret = secret[::-1]
for i in range(secret.__len__()):
    if secret[i] == '0':
        pass
    elif secret[i] == '1':
        circuit.append(QutritCNOT().on(qudits[i],
    last_qudit))
    else:
        circuit.append(QutritCNOTHermitik().on(qudits[i
    ], last_qudit))
```

4.4.4 Adım 3: Tekrardan Hadamard

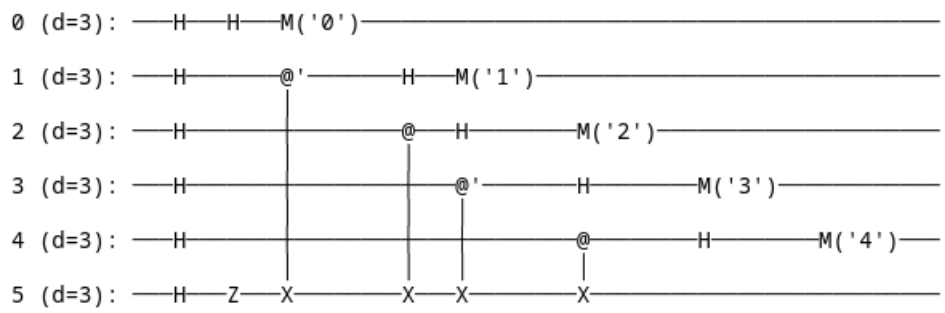
Bu adımda tekrardan tüm kubitlere Hadamard uygulanır.

```
# Son Hadamardları ve ölçümleri ekle
for i in range(secret.__len__()):
    circuit.append(QutritHadamard().on(qudits[i]))

for i in range(secret.__len__()):
    circuit.append(cirq.measure(qudits[i], key=str(i)))
```

4.4.5 Adım 4: Ölçüm

Bu adımda tüm kuditlere ölçüm eklenir.

```
# Tüm qubitlere ölçüm ekle.
for i in range(secret.__len__()):
    circuit.append(cirq.measure(qudits[i], key=str(i)))
print(circuit)
```



4.4.6 Simülasyon

```
secret = "12120"
res = test(1, secret)

result_string = ""
for i in range(secret.__len__()):
    new_list = list()
    res_list = res.measurements[str(i)].tolist()
    for r in res_list:
        new_list.append(r[0])
    res_list = np.array(new_list)
    counts = np.bincount(res_list)
    result_string += str(np.argmax(counts))

print("Gizli numara: ",  secret)
print("Bulunan ölçüm sonuçları: ", result_string[::-1])
print("Doğru mu? {}".format(secret == result_string
  [::-1]))

...
Gizli numara:  12120
Bulunan ölçüm sonuçları:  12120
Doğru mu? True
```

Yukarıdaki sonuçlardan görüleceği üzere yüksek boyutta Bernstain-Vazirani algoritması ile başarılı bir kodlama yapılmıştır. 4. boyutta örneği *.ipynb* üzerindeki örnekte devam etmektedir.

4.5 Gelişmiş Grover Algoritması

Daha önceki bölümde anlatılan iki boyutta Grover algoritması ile seçilen bir hedefi arama yapılıyordu. Bu bölüm ise yüksek boyutta ve aynı anda birden fazla hedef arayabilen bir Grover algoritması örnek üzerinden açıklanmasını içermektedir. Bu bölümde kodlanacak olan yüksek boyutta gelişmiş Grover algoritması 3. boyutta yapılacak olup juptyter-notebooks klasörünün içerisinde **hdim_grover.ipynb** dosyasında bulunmaktadır.

4.5.1 Kurulum

Her ne kadar Grover algoritması iki boyutta olduğu gibi aynı prensibe sahip olsa da yüksek boyutta kodlama için yüksek boyutta kapılar gerektirmektedir. Daha önce işlenen algoritmalarda bulunmayan yeni kapıları tanılamak gerekmektedir. Aşağıda sınıf isimleri verilen **MultiControlQutritX** ve **MultiControlQutritZ** kapıları birden fazla kontrol kutriti ve bir adet hedef kutriti olan kapıları tanımlamaktadır. Bu kapılar *Oracle* içerisinde gerekmektedir. Juptyter-notebooks içindeki alakalı *.ipynb* dosyası içerisinde sınıfın tamamı bulunmaktadır.

```
1  class MultiContolQutritX(cirq.Gate):
2    def __init__(self, qudit_count):
3        super(MultiContolQutritX, self)
4        self.qudit_count = qudit_count
5    ...
6
7  class MultiContolZ(cirq.Gate):
8    def __init__(self, qudit_count):
9      super(MultiContolZ, self)
10       self.qudit_count = qudit_count
11   ...
```

Aşağıdaki kod parçası ile aramak istediğimiz hedefler seçilir ve devre başlatılmış olur.

```
# Aranmak istenen hedefler seçilir.
# Hedefteki herhangi bir rakamın değeri kuditin
  boyutuna eşit veya büyük olamaz.
search_targets = ("0112", "2111", "0012")

# Kutritleri oluşturur.
qudit_count = len(search_targets[0])
qudits = cirq.LineQid.range(qudit_count, dimension=3)
circuit = cirq.Circuit()
```

4.5.2 Adım 1: Hadamard

İki boyutta olduğu gibi yüksek boyutta Grover algoritmasında da tüm kutritlere Hadamard uygulanarak başlanır.

```
# Tüm kutritlere Hadamard uygular.
circuit.append(QutritHadamard().on_each(*qudits))
```

4.5.3 Adım 2: Oracle

İki boyutta olduğu gibi yüksek boyutta Grover algoritmasında da bir *Oracle* bulunmaktadır. Oracle devresini aşağıdaki gibi bir yordam ile kolayca çağırıp kullanabiliriz.

```
def oracle(circuit, targets):
    """ Tüm kutritler üzerinde Oracle oluşturur. """

      # Kutrit sayısına göre ID matrix oluşturulur.
    qudit_count = targets[0].__len__()
    sh = np.power(3, qudit_count)
    id_matrix = np.eye(sh)

    # Kutritler başlatılır.
    qids = cirq.LineQid.range(qudit_count, dimension=3)
    zero = np.array([[1, 0, 0]], dtype=complex)
    one = np.array([[0, 1, 0]], dtype=complex)
    two = np.array([[0, 0, 1]], dtype=complex)
```

```
    # Hedeflere göre Oracle şekillenir.
    # np.kron() = Matrisler arasında tensor çarpım
    for target in targets:
        new = np.array([[1]], dtype=complex)
        total_matrix = copy(id_matrix)
        for qid in reversed(target):
            if qid == "0":
                new = np.kron(zero, new)
            elif qid == "1":
                new = np.kron(one, new)
            elif qid == "2":
                new = np.kron(two, new)
            else:
                raise("?")

        index = None
        for m in range(new.shape[1]):
            if new[0][m] != complex(0, 0):
                index = m
                break

        if index is not None:
            total_matrix[index][index] += -2*new[0][
index]
        else:
            raise("?")

        # Devreye Oracle eklenir.
        circuit.append(Oracle(copy(total_matrix),
qudit_count).on(*qids))
```

4.5.4 Adım 3: Diffission

İki boyutta olduğu gibi yüksek boyutta Grover algoritmasında da bir Diffission bulunmaktadır.

```
def diffision(circuit, qudit_count):
    # Hadamard + Oracle + HermitikHadamard
    qids = cirq.LineQid.range(qudit_count, dimension=3)
    circuit.append(QutritHadamardHermitik().on_each(*
qids))
    oracle(circuit, ["0"*qudit_count])
    circuit.append(QutritHadamard().on_each(*qids))
```

4.5.5 Adım 4: Tekrarlama

İki boyutta olduğu gibi yüksek boyutta Grover algoritmasında da Oracle ve Diffison işlemi $\sqrt{\pi}$ adet tekrarlanmaktadır.

```
# Adım sayısı hesaplanır.
adim_sayisi = int(np.sqrt(len(search_targets[0])))

# Adım sayısı kadar oracle + diffusion.
for i in range(adim_sayisi):
    oracle(circuit, search_targets)
    diffision(circuit, qudit_count)
```

4.5.6 Adım 5: Ölçüm

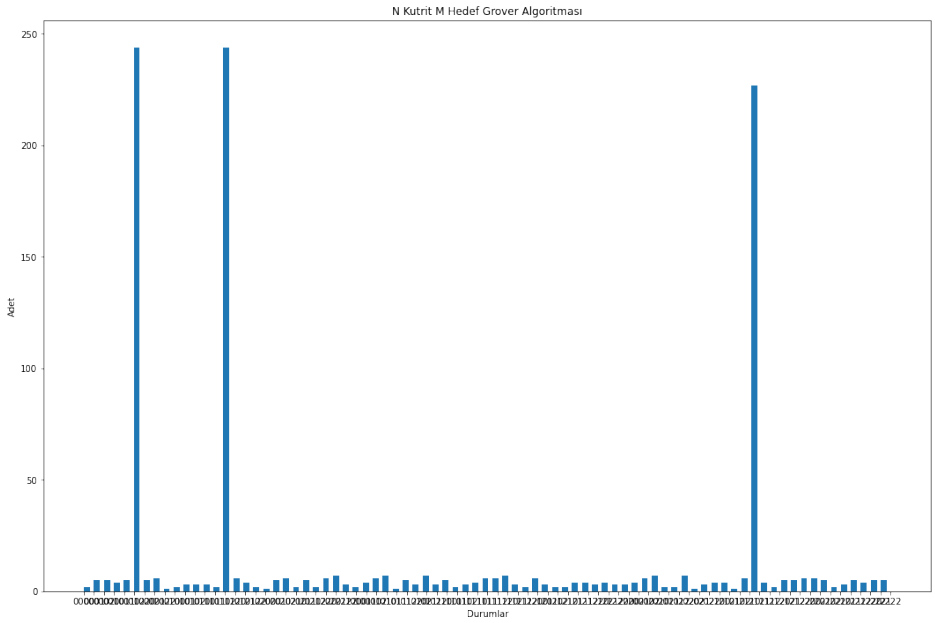
Bu adımda tüm kutritlere ölçüm eklenir.

```
# Ölçümler eklenir.
for i in range(qudit_count):
  circuit.append(cirq.measure(cirq.LineQid(i, dimension
  =3), key="q{}".format(i)))
print(circuit)
```

```
0 (d=3): ──H──O──O──O──Ht──O──H──O──O──O──Ht──O──H──M('q0')──
                │  │  │     │     │  │  │     │
1 (d=3): ──H──O──O──O──Ht──O──H──O──O──O──Ht──O──H──M('q1')──
                │  │  │     │     │  │  │     │
2 (d=3): ──H──O──O──O──Ht──O──H──O──O──O──Ht──O──H──M('q2')──
                │  │  │     │     │  │  │     │
3 (d=3): ──H──O──O──O──Ht──O──H──O──O──O──Ht──O──H──M('q3')──
```

4.5.7 Simülasyon

```
search_targets = ("0112", "2111", "0012")
plot_data, qudit_count = do_grover(1024,search_targets)
plot_data.sort()

fig = plt.figure(figsize=(14, 9))
axes = fig.add_axes([0, 0, 1, 1])
axes.hist(plot_data, np.power(3, qudit_count), width
  =0.6)
plt.title("N Kutrit M Hedef Grover Algoritması")
plt.xlabel("Durumlar")
plt.ylabel("Adet")
plt.show()
```



Yukarıdaki simülasyondan gelen ölçüm sonuçlarına baktığımıza örnekte aranan değerlerin ("0012", "0112", "2111") genliklerinin en fazla oluğunu görülmektedir.

4.6 Yüksek Boyutta Shor Algoritması

Her ne kadar Shor algoritmasının yüksek boyutta işleyişi iki boyutta olduğu gibi olsa da gereken Oracle karakutu fonksiyonu çalışılan boyuta uyarlamak karmaşık oluyor. Bu bölümde 4 boyutta ve mod15 için çalışan bir Shor algoritması örneği incelenecektir. Bu bölümde kodlanacak olan yüksek boyutta Shor algoritması juptyter-notebooks klasörünün içerisinde **hdim_shor.ipynb** dosyasında bulunmaktadır.

4.6.1 Kurulum

İlk olarak Shor algoritması için gereken *mod* algoritması tanımlanmadır. Bu örnek için yapılan mod15 algoritmasını sayacak kadar kapasitede olacak şekilde toplam 2 adet kuqrite ihtiyaç vardır. 2 adet sayma kuqrit ile $4^2 - 1 = 15$'e kadar sayma yapılabilir. Bu nedenle örnek için gereken $Amod15$ dönüşümü 2 kuqritlik kapı olmalıdır. Bu kapı sayma kuqritleri üzerinde gelen girdinin 15 tabanında modunu alarak yeni bir çıktı üretecek şekilde olamalıdır.

```
class QuqritMod15Gate(cirq.Gate):
    def __init__(self, count):
        super(QuqritMod15Gate, self).__init__()
        self.count = count

    def _qid_shape_(self):
        return 4, 4,

    def _unitary_(self):
        arr = np.zeros(shape=(16, 16), dtype=complex)
        arr[15][0] = 1
        arr[14][2] = 1
        arr[13][4] = 1
        arr[12][6] = 1
        arr[11][8] = 1
        arr[10][10] = 1
        arr[9][12] = 1
        arr[8][14] = 1
        arr[7][1] = 1
        arr[6][3] = 1
        arr[5][5] = 1
        arr[4][7] = 1
        arr[3][9] = 1
        arr[2][11] = 1
        arr[1][13] = 1
        arr[0][15] = 1

        if self.count == 1:
            return arr

        for i in range(self.count - 1):
            arr = arr.dot(arr)
        return arr

    def _circuit_diagram_info_(self, args):
        return protocols.CircuitDiagramInfo(
            wire_symbols=('W{}'.format(self.count), 'W'
))

    @property
    def transform_matrix(self) -> np.ndarray:
        return self._unitary_()
```

Gereken kuqrit sayısı hesaplanır ve devre hazırlanır.

```
quqrit_count = 6  #(4 + 2 sayma için)
quqrits = cirq.LineQid.range(quqrit_count, dimension=4)

# Devre hazırlanıyor.
circuit = cirq.Circuit()
```

4.6.2 Adım 1: Hadamard

Bu adımda tüm girdi kuqritlere Hadamard uygulanarak süperpozisyona getirilir. Ayrıca son sayma kubiti üzerinde boyutun artı operatörü ile 1 artırılmalıdır.

```
quqrit_count = 6  #(4 + 2 sayma için)
quqrits = cirq.LineQid.range(quqrit_count, dimension=4)

# Devre hazırlanıyor.
circuit = cirq.Circuit()
circuit.append(Id(4).on(quqrits[4]))

# Sayma kuqriteleri (01) = (0001) = 1 durumuna
  getirilir.
circuit.append(PlusOneGate(4).on(quqrits[5]))
```

4.6.3 Adım 2: Kontrollü Amod15

Bu adımda devre üzerinde defalarca kontrollü Amod15 kapısı eklenir. Her kayıtçı kuqrit hedefi sayma kuqritleri olan bir set kontrollü Amod15 kapısı uygular. Her set üzerinde ise kontrol rakamının değerlerin hepsi ikili olarak eklenecek kadar bu kontrollü kapıyı içerir. Setten farksız olarak her bir kontrollü kapıda hedef sayma kuqritlerine uygulanacak olan Amod15 kapısının üstü alınmalıdır.

```
# MOD15 Devresi kontrollü şekilde ekleniyor. Toplamda 8
  adet.
circuit.append(QuqritMod15Gate(1).controlled(1,
  control_values=[(1, 3)], control_qid_shape=(4,)).on(
  quqrits[0], quqrits[4], quqrits[5]))
circuit.append(QuqritMod15Gate(2).controlled(1,
  control_values=[(2, 3)], control_qid_shape=(4,)).on(
  quqrits[0], quqrits[4], quqrits[5]))
```

```
4 circuit.append(QuqritMod15Gate(4).controlled(1,
   control_values=[(1, 3)], control_qid_shape=(4,)).on(
   quqrits[1], quqrits[4], quqrits[5]))
5 circuit.append(QuqritMod15Gate(8).controlled(1,
   control_values=[(2, 3)], control_qid_shape=(4,)).on(
   quqrits[1], quqrits[4], quqrits[5]))
6 circuit.append(QuqritMod15Gate(16).controlled(1,
   control_values=[(1, 3)], control_qid_shape=(4,)).on(
   quqrits[2], quqrits[4], quqrits[5]))
7 circuit.append(QuqritMod15Gate(32).controlled(1,
   control_values=[(2, 3)], control_qid_shape=(4,)).on(
   quqrits[2], quqrits[4], quqrits[5]))
8 circuit.append(QuqritMod15Gate(64).controlled(1,
   control_values=[(1, 3)], control_qid_shape=(4,)).on(
   quqrits[3], quqrits[4], quqrits[5]))
9 circuit.append(QuqritMod15Gate(128).controlled(1,
   control_values=[(2, 3)], control_qid_shape=(4,)).on(
   quqrits[3], quqrits[4], quqrits[5]))
```

4.6.4 Adım 3: Ters Kuantum Fourier Dönüşümü

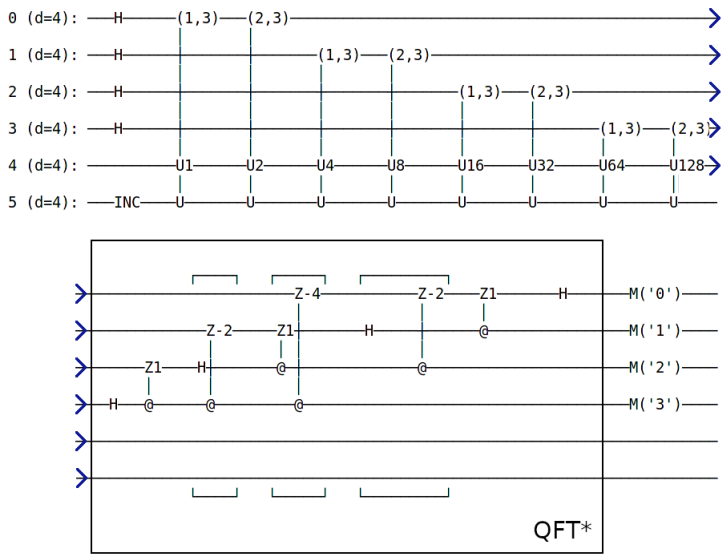
Bu adımda devreye ters kuantum fourier dönüşümü eklenir.

```
1 # QFT Tersi ekleniyor.
2 circuit.append(QuqritHadamard().on(quqrits[3]))
3 circuit.append(ControlledQuqritPhaseGate(1).on(quqrits
   [3], quqrits[2]))
4 circuit.append(QuqritHadamard().on(quqrits[2]))
5 circuit.append(ControlledQuqritPhaseGate(-2).on(quqrits
   [3], quqrits[1]))
6 circuit.append(ControlledQuqritPhaseGate(1).on(quqrits
   [2], quqrits[1]))
7 circuit.append(QuqritHadamard().on(quqrits[1]))
8 circuit.append(ControlledQuqritPhaseGate(-4).on(quqrits
   [3], quqrits[0]))
9 circuit.append(ControlledQuqritPhaseGate(-2).on(quqrits
   [2], quqrits[0]))
10 circuit.append(ControlledQuqritPhaseGate(1).on(quqrits
   [1], quqrits[0]))
11 circuit.append(QuqritHadamard().on(quqrits[0]))
```

4.6.5 Adım 4: Ölçüm

Bu adımda kayıtçı kuqritlere ölçüm eklenir.

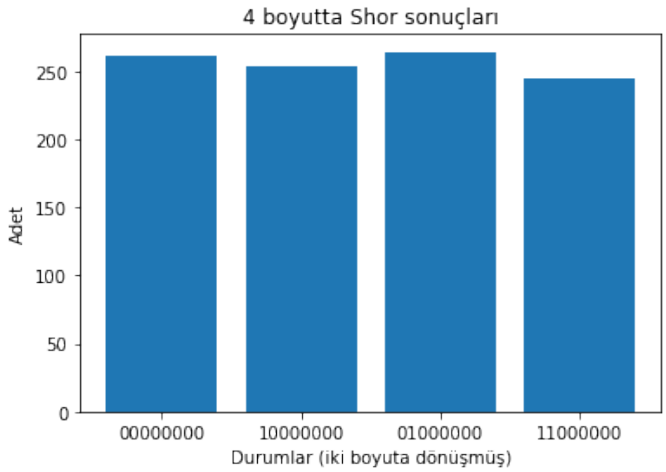
```
# Ölçümler ekleniyor.
circuit.append(cirq.measure(quqrits[0], key="{}".format
    (0)))
circuit.append(cirq.measure(quqrits[1], key="{}".format
    (1)))
circuit.append(cirq.measure(quqrits[2], key="{}".format
    (2)))
circuit.append(cirq.measure(quqrits[3], key="{}".format
    (3)))
print(circuit)
```



4.6.6 Simülasyon

```
# Plotlama için hazırlanıyor.
result_histogram = list()
plot = dict()

# 4 boyutlu sonuçları iki boyuta döndürür.
for i in range(shots):
    new_string = str()
```

```
8       for j in range(4):
9           value = res.measurements["{}".format(j)][i][0]
10          if value == 0:
11              new_string += "00"
12          elif value == 1:
13              new_string += "01"
14          elif value == 2:
15              new_string += "10"
16          else:
17              new_string += "11"
18      try:
19          plot[new_string] += 1
20      except KeyError:
21          plot[new_string] = 1
22      result_histogram.append(new_string)
23
24  pyplot.bar(plot.keys(), plot.values())
25  pyplot.title("4 boyutta Shor sonuçları")
26  pyplot.xlabel("Durumlar (iki boyuta dönüşmüş)")
27  pyplot.ylabel("Adet")
```



4.6.7 Adım 5: Periyot Bulma

Shor algoritmasının simülasyonundan gelen sonuçlar 4 boyuttan 2 boyuta indirgendiği için bu adım iki boyutlu Shor algoritması ile aynı devam eder. Klasik işlemler içeren bu adımda ölçüm sonuçları incelenerek öncelikle periyot

bulunur. Bulunan periyot üzerinden en büyük ortak bölenler incelenerek asal çarpanlar bulunur.

```
                 8Bit çıktı             Faz
0  00000000(bin) =   0(dec)    0/256 = 0.00
1  10000000(bin) = 128(dec)  128/256 = 0.50
2  01000000(bin) =  64(dec)   64/256 = 0.25
3  11000000(bin) = 192(dec)  192/256 = 0.75

Bulunan frekans:  4

    Faz Kesit  R
0  0.00   0/1  1
1  0.50   1/2  2
2  0.25   1/4  4
3  0.75   3/4  4
Tahminler:  [3, 5]
```

SON