第 1 章

Eight Queen Problem

1.1 量子アニーリングで N-クィーン問題を解く

1.2 定式化

N-Queen 問題を考えてみる．$x_{i,j}$ は,i 行 j 列目のマスを, 0 にするか 1 にするかを表すものとする．例えば、
2 行 1 列目のマスを 1 にする場合, $x_{2,1} = 1$ に、
1 行 3 列目のマスを 1 にしない場合, $x_{1,3} = 0$ とする．
$(N \times N)$ の盤面における N-Queen 場合の場合, 変数の組み合わせは次の様になる．

$$
\begin{aligned}
\boldsymbol{x} = \{ & \\
& x_{1,1}\,, x_{1,2}, \ldots, x_{1,N} \\
& x_{2,1}\,, x_{2,2}, \ldots, x_{2,N} \\
\vdots & \\
& x_{N,1}, x_{N,2}, \ldots, x_{N,N} \}
\end{aligned}
$$

(ただしプログラムの実装時には, 配列の添字は 0 から $N-1$ までの N 次の配列になるので, このことを考慮する必要がある)

量子アニーリングで解くということは, 評価関数 f の値が, この組み合わせ $\boldsymbol{x}$ が正しい N-Queen の配置を構成できているときには小さい値になり, 間違った構成の場合には大きな値になる．その様な QUBO 行列 Q を決定していくことになる．

1.2.1 縦横どの 1 列の数値の総和も等しい

N-queen が正しい構成になるときには, 縦横のどの列についての総和も等しくなるという制約がある．

$$
\begin{aligned}
f_0(x) &= \sum_k \sum_l \left(\sum_{(i,j)\in L_k} x_{i,j} - \sum_{(i,j)\in L_l} x_{i,j} \right)^2 \\
&= \sum_k \sum_l \left(\sum_{(i_1,j_1)\in L_k} \sum_{(i_2,j_2)\in L_k} x_{i_1,j_1} x_{i_2,j_2} - 2 \sum_{(i_1,j_1)\in L_k} \sum_{(i_2,j_2)\in L_l} x_{i_1,j_1} x_{i_2,j_2} \right. \\
&\qquad \left. + \sum_{(i_1,j_1)\in L_l} \sum_{(i_2,j_2)\in L_l} x_{i_1,j_1} x_{i_2,j_2} \right)
\end{aligned}
$$

ただし,L_k は列に含まれるマスの集合のことで, 以下の通り. $L_1 \sim L_N$ は横の行, $L_{N+1} \sim L_{2N}$ は縦の列, を表している.

$$
\begin{aligned}
L_1 &= \{(1,1),(1,2),\ldots,(1,N)\} \\
L_2 &= \{(2,1),(2,2),\ldots,(2,N)\} \\
&\vdots \\
L_N &= \{(N,1),(N,2),\ldots,(N,N)\} \\
L_{N+1} &= \{(1,1),(2,1),\ldots,(N,1)\} \\
L_{N+2} &= \{(1,2),(2,2),\ldots,(N,2)\} \\
&\vdots \\
L_{2N} &= \{(1,N),(2,N),\ldots,(N,N)\}
\end{aligned}
$$

また, ある列 k の総和と, ある列 l の総和の差を d とすると,

$$
\left(\sum_{(i,j)\in L_k} x_{i,j} - \sum_{(i,j)\in L_l} x_{i,j} \right)^2 = d^2
$$

であることから, $d = 0$ の時（すべての列の総和が等しい時）に最小になる.
（N-queen の各行および各列における総和 S は単に 1 になる.）

各行（横方向）の値の総和は S である

$$
\begin{aligned}
f_1(\boldsymbol{x}) &= \sum_i \left(\sum_j x_{i,j} - S \right)^2 \\
&= \sum_i \left(\sum_{j_1} \sum_{j_2} x_{i,j_1} x_{i,j_2} - 2 \sum_j S x_{i,j} + S^2 \right)
\end{aligned}
$$

$x_{i,j} \in \{0,1\}$ に注意すると, $x_{i,j}$ は x を掛けた 2 次の形にしてもよい. また, 最小化問題では定数項は影響しないので無視できる.（以下同様に）

各列（縦方向）の値の総和は S である

$$
\begin{aligned}
f_2(\boldsymbol{x}) &= \sum_j \left(\sum_i x_{i,j} - S \right)^2 \\
&= \sum_j \left(\sum_{i_1} \sum_{i_2} x_{i_1,j} x_{i_2,j} - 2 \sum_i S x_{i,j} + S^2 \right)
\end{aligned}
$$

斜め方向の値の総和は 0 または 1 である

$$
f_3(\boldsymbol{x}) = \left(\sum_d x_{d,d} - S \right)^2 + \left(\sum_d x_{d,N-d+1} - S \right)^2
$$

左側の項は右下がりの斜めの列に対する制約, 右側の項は右上がりの斜めの列に対する制約.

1.2.2　盤面の数値の総和は N である

8-queen の場合, N は 8 である.

$$
\begin{aligned}
f_4(\boldsymbol{x}) = \left(\sum_i \sum_j x_{i,j} - N \right)^2 &= \left(\sum_i \sum_j x_{i,j} - N \right) \left(\sum_i \sum_j x_{i,j} - N \right) \\
= \sum_{i_1,i_2} \sum_{j_1,j_2} x_{i_1,j_1} x_{i_2,j_2} - 2 \sum_i \sum_j N x_{i,j} + N^2 \\
= \sum_{i_1,i_2} \sum_{j_1,j_2} x_{i_1,j_1} x_{i_2,j_2} - 2 \sum_i \sum_j N x_{i,j} x_{i,j}
\end{aligned}
$$

$x_{i,j} \in \{0, 1\}$ を考慮して, 二次形式に直している.

1.2.3　N-Queen 生成の評価関数

以上の制約を足し合わせることで N-Queen の生成に必要な評価関数を作ることができる.

$$
f(\boldsymbol{x}) = \lambda_1 f_1(\boldsymbol{x}) + \lambda_1 f_2(\boldsymbol{x}) + \lambda_2 f_3(\boldsymbol{x}) + \lambda_3 f_4(\boldsymbol{x})
$$

或いは、

$$
f(\boldsymbol{x}) = \lambda_1 f_0(\boldsymbol{x}) + \lambda_3 f_4(\boldsymbol{x})
$$

1.3　実装

```
pip install dwave-ocean-sdk
pip install openjij
```

1.3.1　初期化

```
from collections import defaultdict
import numpy as np
from dwave.optimization.symbols import Sum
from dwave.system import DWaveSampler, EmbeddingComposite
from openjij import SASampler, SQASampler
import time

rotate90 = lambda A: [list(x)[::-1] for x in zip(*A)]  # 90度回転用関数
transpose = lambda A: [list(x) for x in zip(*A)]  # 転置用関数

class EightQueen:
```

以下は EightQueen クラスにまとめてたもの．必要なライブラリのインポートや変数, 定数はパラメータを変更することで,N-Queen の大きさや使う量子アニーリングマシンなどを変更することができる．

```
  def __init__(self):
    self.N = 8  # 盤面の大きさはNxN
    self.S = 1.0  # 各列の総和
    self.l1 = 2.0  # 罰金項の強さ（各行、各列の総和は等しい）
    self.l2 = 1.5  # 罰金項の強さ（斜めの総和は0または1）
    self.l3 = 1.0  # 罰金項の強さ（盤面上はN個）
    self.num_reads = 10000  # アニーリングを実行する回数
    self.token = 'XXXX'  # API token(個人のものを使用)
    self.solver = 'Advantage_system6.4'  # 量子アニーリングマシン
    self.machine = False
    Sampler = [SASampler(), SQASampler()]
    self.sampler = Sampler[0]
    self.ij_to_idx = {}

  def myindex(self):
    Q = defaultdict(lambda: 0)  # Q_i,j  (i, j)に入れる値
    # x_i,jの通し番号を記録
    ij_to_idx = {}
    idx = 0
    for i in range(self.N):
        for j in range(self.N):
            ij_to_idx[(i, j)] = idx
            idx += 1
    self.ij_to_idx = ij_to_idx
    return Q, ij_to_idx
```

1.3.2　QUBO 行列を作る

各列の総和は等しい (f_1, f_2, f_3 の制約)

```
  def constraint(self, Q, ij_to_idx):
    Q = self.constraint_row(Q, ij_to_idx)
```

```
        Q = self.constraint_clmn(Q, ij_to_idx)
        Q = self.constraint_diagonal(Q, ij_to_idx)
        Q = self.constraint_N(Q, ij_to_idx)
        return Q

    def constraint_row(self, Q,  ij_to_idx):
        # 各行の総和をSに制限(f1)
        for i in range(self.N):
            for j1 in range(self.N):
                for j2 in range(self.N):
                    Q[(ij_to_idx[(i, j1)], ij_to_idx[(i, j2)])] += self.l1
                Q[(ij_to_idx[(i, j1)], ij_to_idx[(i, j1)])] -= 2 * self.S * self.l1
        return Q

    def constraint_clmn(self, Q,  ij_to_idx):
        # 各列の総和をSに制限(f2)
        for j in range(self.N):
            for i1 in range(self.N):
                for i2 in range(self.N):
                    Q[(ij_to_idx[(i1, j)], ij_to_idx[(i2, j)])] += self.l1
                Q[(ij_to_idx[(i1, j)], ij_to_idx[(i1, j)])] -= 2 * self.S * self.l1
        return Q

    def constraint_diagonal(self, Q, ij_to_idx):
        # 斜めの各列の総和は0または1
        # 左上から右下の斜め制約 (i - j が等しい)
        for k in range(-self.N + 1, self.N):  # 有効なダイアゴナルの範囲
            diagonal_indices = [(i, j) for i in range(self.N) for j in range(self.N) if
             i - j == k]
            for idx1 in diagonal_indices:
                for idx2 in diagonal_indices:
                    Q[(ij_to_idx[idx1], ij_to_idx[idx2])] += self.l2
                Q[(ij_to_idx[idx1], ij_to_idx[idx1])] -= 2 * self.S * self.l2

        # 右上から左下の斜め制約 (i + j が等しい)
        for k in range(2 * self.N - 1):  # 有効なダイアゴナルの範囲
            diagonal_indices = [(i, j) for i in range(self.N) for j in range(self.N) if
             i + j == k]
            for idx1 in diagonal_indices:
                for idx2 in diagonal_indices:
                    Q[(ij_to_idx[idx1], ij_to_idx[idx2])] += self.l2
                Q[(ij_to_idx[idx1], ij_to_idx[idx1])] -= 2 * self.S * self.l2
        return Q

    def constraint_N(self, Q, ij_to_idx):
        # NxNの盤面上の数値の総和はNである
        for i1 in range(self.N):
            for j1 in range(self.N):
                for i2 in range(self.N):
                    for j2 in range(self.N):
                        Q[(ij_to_idx[(i1, j1)], ij_to_idx[(i2, j2)])] += self.l3
                Q[(ij_to_idx[(i1, j1)], ij_to_idx[(i1, j1)])] -= 2 * self.N * self.l3
```

```
    return Q
```

1.3.3　量子アニーリングの実行

solver(Advantage_system6.4) という量子アニーリングマシンで, num_reads(今回は 1000) 回量子アニーリングを実行させせることも可能だが有料になるので, Openjij のシミュレータを使うことにする. openjij から SASampler または SQASampler のいずれかを使って, ここまでに作成した QUBO 行列 Q に基づいた量子アニーリングの実行をシミュレートすることができる.

```
def annealing(self, Q):
  if self.machine:
      # 量子アニーリングの実行
      endpoint = 'https://cloud.dwavesys.com/sapi/'
      dw_sampler = DWaveSampler(solver=self.solver, token=self.token, endpoint=
  endpoint)
      sampler = EmbeddingComposite(dw_sampler)
      sampleset = sampler.sample_qubo(Q, num_reads=self.num_reads)
  else:
      # 焼きなまし法の実行
      sampler = self.sampler
      sampleset = sampler.sample_qubo(Q, num_reads=self.num_reads)
  return sampleset
```

1.3.4　出力結果のフィルタ

量子アニーリングが出力する結果には, 最適化しきれずに制約条件を満たせないままのもの, あるいは同じ結果を重複して複数回出力しているもの, そして, 回転対称や鏡像の盤面も複数含まれて出力されてくる. まず制約条件を満たせないまま出力されているものを除外（check_valid 関数）し, 対称な形も含めた重複出力を除外（check_duplicate 関数）する様にしている.

```
def gen_mat(self, sset):
  mat = []
  for i in range(self.N):
      w = []
      for j in range(self.N):
          w.append(sset[(self.ij_to_idx[(i, j)])])
      mat.append(w)
  return mat

def same_p(self, sset1, sset2):
  s1 = np.array(sset1).flatten()
  s2 = np.array(sset2).flatten()
  if np.array_equal(s1, s2):
      return True
  return False

def check_row_clmn(self, mat):
  Sum = self.S
```

```
    for i in range(self.N):
        s = 0.0
        for j in range(self.N):
            s += mat[i][j]
        if Sum < s:
            return False
    return True

  def check_diagonal(self, mat):   # def check2(mat):
    Sum = self.S
    # 左上から右下の斜め (i - j が等しい)
    for k in range(-self.N + 1, self.N):  # 有効なダイアゴナルの範囲
        diagonal_indices = [(i, j) for i in range(self.N) for j in range(self.N) if
     i - j == k]
        s = 0
        for d in diagonal_indices:
            s += mat[d[0]][d[1]]
        if Sum < s:
            return False
    # 右上から左下の斜め (i + j が等しい)
    for k in range(2 * self.N - 1):  # 有効なダイアゴナルの範囲
        diagonal_indices = [(i, j) for i in range(self.N) for j in range(self.N) if
     i + j == k]
        s = 0
        for d in diagonal_indices:
            s += mat[d[0]][d[1]]
        if Sum < s:
            return False
    return True

  def check_valid(self, sampleset):
    removelist = []
    for nth, sset in enumerate(sampleset):
        # Convert the solution into the 2D matrix form
        mat = self.gen_mat(sset)  # Properly generate the matrix form
        # Conduct various checks (rows, columns, diagonals)
        ck1 = self.check_row_clmn(mat)  # Check rows
        if not ck1:
            removelist.append(nth)
            continue
        ck2 = self.check_row_clmn(transpose(mat))  # Check columns
        if not ck2:
            removelist.append(nth)
            continue
        ck3 = self.check_diagonal(mat)  # Check diagonals
        if not ck3:
            removelist.append(nth)
            continue
    return self.remove_from_sampleset(sampleset, removelist)

  def check_duplicate(self, sampleset):
    # Ensure no duplicates due to symmetry or rotations
    removelist = []
    for nth1, sset1 in enumerate(sampleset):
```

```
        if nth1 in removelist:
            continue
        sets1 = self.variable_mat(sset1)
        for nth2, sset2 in enumerate(sampleset):
            if nth1==nth2 or nth2 in removelist:
                continue
            sets2 = self.variable_mat(sset2)
            for elem1 in sets1:
                for elem2 in sets2:
                    if self.same_p(elem1, elem2):
                        if nth2 not in removelist:
                            removelist.append(nth2)
                            break
            else:
                continue
    return self.remove_from_sampleset(sampleset, removelist)

  def remove_from_sampleset(self, sampleset, removelist):
    sampleset1 = []
    for nth, sset in enumerate(sampleset):
        if nth not in removelist:
            sampleset1.append(sset)
    return sampleset1

  def variable_mat(self, sset):
    sets = []
    set = self.gen_mat(sset)  # Convert to matrix form
    set1 = rotate90(set)     # rotate 90
    set2 = rotate90(set1)    # rotate 180
    set3 = rotate90(set2)    # rotate 270
    set4 = np.fliplr(np.array(set)).copy().tolist()
    set5 = np.flipud(np.array(set)).copy().tolist()
    sets = [set, set1, set2, set3, set4, set5]
    return sets

  def check(self, sampleset):
    sampleset1 = self.check_valid(sampleset)
    tm1 = time.time()
    print("valid   =", len(sampleset1))
    sampleset2 = self.check_duplicate(sampleset1)
    tm2 = time.time()
    print("checked =", len(sampleset2))
    return sampleset2, tm1, tm2
```

1.3.5　結果出力（可視化）

num_reads 回のアニーリングの結果から, 重複, 回転対称, 鏡像などのチェックをクリアしたものを可視化している.

```
  def result(self, resultset):
    for nth, sset in enumerate(resultset):
        mat = self.gen_mat(sset)
```

```
        print(nth+1)
        for i in range(self.N):
            print(mat[i])
        print()
```

または, 各出力結果ごとに, 回転対称（90 度,180 度,270 度）のもの, 鏡像（左右, 上下）のもの, を同時に生成出力して, 確認できるようにしたものは以下.

```
def result_check_do(self, resultset):
  for nth, sset in enumerate(resultset):
        sset = self.gen_mat(sset)  # 行列の形に変換
        set1 = rotate90(sset)
        set2 = rotate90(set1)
        set3 = rotate90(set2)
        set4 = (np.array(sset)[:, ::-1]).tolist()
        set5 = (np.array(sset).T[:, ::-1].T).tolist()
        print(nth+1)
        for i in range(self.N):
            print(sset[i], set1[i], set2[i], set3[i], set4[i], set5[i])
        print()
```

1.3.6　実行（主処理）

```
if __name__ == '__main__':
  start = time.time()
  eq = EightQueen()
  Q, ij2idx = eq.myindex()
  Q = eq.constraint(Q, ij2idx)
  ckpt1 = time.time()
  sampleset = eq.annealing(Q)
  ckpt2 = time.time()
  print("annealed=",len(sampleset))
  resultset, ckpt3, ckpt4 = eq.check(sampleset)
  eq.result_check_do(resultset)
  #
  print("Prepare:{}".format(ckpt1 - start))
  print("Annealing:{}".format(ckpt2 - ckpt1))
  print("ValidateCheck:{}".format(ckpt3 - ckpt2))
  print("DuplicateCheck:{}".format(ckpt4 - ckpt3))
  print("Total:{}".format(ckpt4 - start))
```

1.4　まとめ

実行例. 出力は以下の通り. 8 クィーンの場合,10000 回の num_reads の内で, 制約条件を満たした出力だったのが 7074 個. その内, 重複や対称な恰好のものを除外した結果,8 クィーンの基本解である 12 個を全て出力できている. また, それ以上の数を出してくることはない（重複を総当たりで潰しているのですか

ら）.elapsed time は, インテルシリコンの Mac で 1 分弱で済んでいる. num_reads を小さくすると, 出力はだんだん 12 個の基本解を網羅できなくなってくる.

```
annealed= 10000
valid   = 7074
checked = 12
1
[0, 0, 0, 0, 1, 0, 0, 0]
[0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 1, 0, 0]
[1, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0]
[0, 0, 0, 1, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1]
[0, 0, 1, 0, 0, 0, 0, 0]

2
[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, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 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]

3
[0, 0, 0, 0, 0, 1, 0, 0]
[0, 0, 0, 1, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 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, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1]

4
[0, 0, 0, 0, 1, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1]
[0, 0, 0, 1, 0, 0, 0, 0]
[1, 0, 0, 0, 0, 0, 0, 0]
```

```
[0, 0, 0, 0, 0, 0, 1, 0]
[0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 1, 0, 0]
[0, 0, 1, 0, 0, 0, 0, 0]

5
[0, 0, 0, 0, 0, 0, 0, 1]
[0, 0, 0, 1, 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, 1, 0, 0]
[0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0]
[0, 0, 0, 0, 1, 0, 0, 0]

6
[0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 1, 0, 0]
[0, 0, 0, 1, 0, 0, 0, 0]
[1, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 1, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0]
[0, 1, 0, 0, 0, 0, 0, 0]

7
[0, 0, 0, 0, 1, 0, 0, 0]
[0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1]
[1, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 1, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0]
[0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 1, 0, 0]

8
[0, 0, 0, 1, 0, 0, 0, 0]
[0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 1, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0]
```

```
[1, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 1, 0, 0]

9
[0, 0, 0, 1, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 1, 0, 0]
[1, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 0, 0, 0]
[0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1]
[0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0]

10
[0, 0, 0, 0, 1, 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, 1]
[0, 0, 1, 0, 0, 0, 0, 0]
[1, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0]

11
[0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 1, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1]
[1, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 1, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0]
[0, 0, 0, 0, 1, 0, 0, 0]
[0, 1, 0, 0, 0, 0, 0, 0]

12
[0, 0, 0, 0, 0, 1, 0, 0]
[0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 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, 0, 1]
[0, 0, 0, 1, 0, 0, 0, 0]

Prepare:0.003168821334838867
Annealing:20.62755823135376
ValidateCheck:3.63771915435791
DuplicateCheck:33.6769118309021
Total:57.94535803794861

プロセスは終了コード 0 で終了しました
```