Reiknigreind - Reikniverkefni 2

Matthías Páll Gissurarson

27. apríl 2014

Monte Carlo Spilari

Niðurstöður: Þegar `Monte Carlo` spilaði við Random, þá vann `Monte Carlo` 46 skipti af 60, svo `Monte Carlop` hefur töluvert mikið forskot á móti random spilara.

Hinsvegar, þá fer `Monte Carlo` vs. `Monte Carlo Plus` 15−5−40, þ.e. `Monte Carlo Plus` vann 40 skipti af 60, eða 2/3, og 5 jafntefli. Það er töluvert forskot, en oftast þegar `Monte Carlo` vann, þá var það vegna þess að hann var kominn með svikamyllu, þ.e. þannig að sama hvað `Monte Carlo Plus` gerði, þá mundi `Monte Carlo` vinna.

Heuristic

Ég ákvað að prófa probabilisitc heuristic, sem metur hag gildi (utility value) ástands útfrá því hve langt það er frá sigri/tapi, þ.e. gefur hátt utility fyrir skjótan sigur, (og því ætti hann að velja það að blokka/sigra ef það er í boði), en lægra fyrir sigur sem er lengra í burtu. Einnig er gefin mikil refsing ef núverandi spilari er búinn að tapa, en mikill ávinningur ef hann er búinn að vinna.

Þegar `Heuristic` keppir við `Monte Carlo` þá sigrar það í 54 skipti af 60, sem er betra en gengið hjá `Monte Carlo Plus` Þegar það keppir við `Monte Carlo Plus`, þá er staðan hinsvegar jöfn, $30 - 30$, svo það virðist sem það sé jafn gott og `Monte Carlo Plus`, en það kemur heim og saman við það að `Heuristic` er að hegða sér mjög líkt `Monte Carlo Plus`.

Þegar `Heuristic` keppir við `Board Inversion Heursitic`, þá fer það $57 - 3$ fyrir `Heuristic`, svo `Heuristic` virðist vera betra mat á stöðunni en board inversion, með w sem w sem gefið er í kynningunni.

Þegar `Board Inversion Heuristic` keppir við `Monte Carlo`, þá fer það $49-11$ fyrir `Monte Carlo`.

Reinforcement Learning

Við forritið var bætt við $TD(\lambda)$ útfærslu, sem var unnin út frá $TD(0)$ matlab lausninni sem við fengum gefna. Þegar $\lambda = 0$, þá hagar það sér eins og $TD(0)$, og skilar álíka win-rate og það gerði fyrir $TD(0)$, sem bendir til þess að útfærslan sé rétt. $TD(1)$ hagar sér eins og `Monte Carlo` skv. fræðunum, en með $\lambda = 1$ fæ ég mjög álíkt winrate og `Monte-Carlo` gegn random, en hinsvegar þegar að ég læt það keppa við `Monte Carlo`, þá virðist það ganga mun verr, en það sigrar aðeins 13 skipti af 60. Það er sennilega vegna þess að í `Monte-Carlo` er ég að taka meðaltal útfrá stöðunni eins og hún er núna, en í `Board Inversion Heuristic` með $\lambda = 1$ er ég í raun ekki að taka mið út frá stöðunni eins og hún er, heldur er ég bara að byggja á því sem ég hef séð áður. Þegar $\lambda = 0.7$, þá er þetta ennþá verra, en á móti `Monte Carlo`, þá fer leikurinn $11-3-46$, þ.e. `Board Inversion Heuristic` sigrar 11 leiki af 60, og það eru 3 jafntefli. Það verður að teljas frekar slappt, svo hugsanlega er galli í implementationinu á $TD(\lambda)$

`Board Inversion Heuristic` mundi sennilega standa sig betur ef það væri þjálfað í fleiri episode, en hjá mér eru það aðeins 2000 epsiode sem það fer í gegnum, en það er sökum þess hve lengi þetta tekur að keyra. Einnig væri hægt að tune-a gildin á λ, α og γ í forritinu til þess að fá betri niðurstöður.

`N-tuples` voru implementuð á svipaðan hátt og þau voru gerð í sýnidæmi, þ.e. þau voru generateu-ð þannig að þau væru eins og snákar í borðinu, en ekki random eins og stungið var uppá. Þau voru síðan notuð til að þjálfa vigtir fyrir þau tuple.

Þjálfun á spilaranum fyrir `N-tuples` tók rosalega langan tíma, en það er sennilega vegna þess að það var ekki útfært með fylkjum, heldur með einfaldari hlutum í python. Það var því ekki þjálfað fyrir fleiri en 2000 episode.

Þar sem að þetta eru heldur fá episode (talað er um 100 milljón episode í greininni), þá stóð `N-tuples` sig frekar illa á móti `Monte-Carlo Plus` (selective útgáfunni), en staðan var $4-56$ `Monte-Carlo Plus` í vil. Þar sem að `Monte-Carlo Plus` var mjög álíkt mínu Heuristic, þá áætlum við að það muni fara svipað gegn því.

Það verður að teljast heldur slappt, en eins og bent hefur verið á, þá mundi niðurstaðan vera mjög ólík ef þjálfuð væru fleiri episode. Sennilega væri hægt að bæta þessa niðurstöðu með því að láta vigtirnar vera þjálfaðar gegn `Monte-Carlo Plus` spilara, frekar en random eins og gert er, því þá mundi `N-tuples` miða frekar að að vinna þann spilara.

Hvað ég hef lært

Eftir að hafa unnið þetta verkefni hef ég núna mun betri skilning á hvernig TD og Monte Carlo hermanirnar virka, sérstaklega eftir að maður útfærði það sjálfur. Einnig veitti útfærslan á N-tuples mér innsýn yfir í hvernig má búa til sniðugt subspace af featurespace, til þess að geta ráðið við hversu stórt það verður.

Viðauki

../connect4.py

```
1  from random import choice
2  from pylab import *
3  from util import *
4
5  class connect4(object):
6      #Creates new game
7      def __init__(self, state = None, currPlayer = None,
           alpha=0.5, w = givenw, ntups = None, verbose = False
           , shape = (7,6)):
8          self.shape = shape
9          self.state = zeros(shape) if state is  None else
               state
10         self.win = None
11         self.alpha = alpha
12         self.w = w
13         self.currPlayer = 1 if currPlayer is None else
               currPlayer
14         self.color = True
15         self.ntups = ntups
16         self.verbose = verbose
17
18     def getFeatures(self):
19         if self.ntups is not None:
20             ks = []
21             N = 0
22             for tup in self.ntups:
23                 k = N
24                 N += 3**len(tup)
```

```
25                  for i,(x,y) in enumerate(tup):
26                      s = self.state[x,y]
27                      s = 1 if s < 0 else 2 if s > 0 else 0
28                      k += s*3**i #3 possible states
29                  ks.append(k)
30              phi = zeros((N,))
31              phi[ks] = 1
32          else:
33              cop = copy(self.state)
34              phi = cop.flatten()
35          return phi
36
37      def makeNtups(self,n=8,numtups=70):
38          #if self.verbose:
39          print("Creating %d %d-tuples" % (numtups,n))
40          ntups = []
41          for i in range(numtups):
42              if self.verbose:
43                  print("Creating tuple %d" %(i+1,))
44              x,y = randint(0,self.shape[0]), randint(0,self.
                    shape[1])
45              snake = []
46              for _ in range(n):
47                  snake.append((x,y))
48                  while (x,y) in snake:
49                      minx = -1 if x-1 >= 0 else 0 #Inclusive
50                      maxx = 2 if x+1 < self.shape[0] else 1
                            #Exclusive
51                      miny = -1 if y-1 >= 0 else 0
52                      maxy = 2 if y+1 < self.shape[1] else 1
53                      x += randint(minx,maxx)
54                      y += randint(miny,maxy)
55              ntups.append(snake)
56          self.ntups = ntups
57
58
59      def copy(self):
60          return connect4(state = copy(self.state),
61                          currPlayer = self.currPlayer,
62                          shape = self.shape,
```

```
63                          alpha = self.alpha,
64                          w = self.w,
65                          ntups = self.ntups,
66                          verbose = self.verbose)
67
68
69      #Make a legal move
70      def makeRandomMove(self):
71          legalMoves = self.getLegalMoves()
72          move = choice(legalMoves)
73          return self.simulate(move)
74
75      def getLegalMoves(self):
76          legal = lambda x: self.isLegal(x)
77          return list(filter(legal,range(0,self.shape[1])))
78
79
80      def isLegal(self,move):
81          t = transpose(self.state)
82          fi = find(t[move]==0)
83          return len(fi) > 0
84
85
86      # 0 $\leq$ action $\leq$ shape[0]
87      # colour = -1 $\wedge$ 1
88      def simulate(self,action,colour = None):
89          if colour is None:
90              colour = self.currPlayer
91              self.currPlayer *= -1
92
93          t = transpose(self.state)
94          indicesWhereEmpty = find(t[action]==0)
95          row = argmax(indicesWhereEmpty)
96          column = action
97          self.state[row][column] = colour
98          self.state = transpose(t)
99          self.win = self.checkwin(self.state)
100         return self.state, self.win
101
102     def rollout(self,move):
```

```
103             sumReward = 0
104             simmove = self.copy()
105             state,win = simmove.simulate(move)
106             for i in range(5):
107                 simulation = simmove.copy()
108                 while win is None:
109                     state, win = simulation.makeRandomMove()
110                 sumReward += win
111             return sumReward * currPlayer
112 
113         def rolloutPlus(self,move):
114             sumReward = 0
115             player = self.currPlayer
116             simmove = self.copy()
117             state,win = simmove.simulate(move)
118             for i in range(5):
119                 simulation = simmove.copy()
120                 while win is None:
121                     if simulation.currPlayer == player:
122                         obvMove = simulation.obviousMove()
123                         if  obvMove is not None:
124                             state,win = simulation.simulate(
                                    obvMove)
125                         else:
126                             state, win = simulation.
                                    makeRandomMove()
127                     else:
128                         state, win = simulation.makeRandomMove
                            ()
129                 sumReward += win
130             return sumReward * currPlayer
131 
132 
133         def winningMove(self):
134             legalMoves = self.getLegalMoves()
135             for move in legalMoves:
136                 simulation = self.copy()
137                 state, win = simulation.simulate(move)
138                 if win == self.currPlayer:
139                     return move
```

```
140         return None
141
142     def blockMove(self):
143         legalMoves = self.getLegalMoves()
144         for move in legalMoves:
145             simulation = self.copy()
146             simulation.currPlayer = -1*self.currPlayer
147             state,win = simulation.simulate(move)
148             if win == -1*self.currPlayer:
149                 return move
150         return None
151
152
153     def obviousMove(self):
154         winMove = self.winningMove()
155         if winMove is not None:
156             return winMove
157         blockMove = self.blockMove()
158         if blockMove is not None:
159             return blockMove
160         return None
161
162     def learnWFromTd(self, eps = 0.1, alpha = 0.01,
163                      numEpisodes = 10000, lamb =0, gamma=
                             1, useNtups = True):
164         if useNtups:
165             self.makeNtups()
166         n = len(self.getFeatures())
167         w = zeros(n,)
168         ntups = self.ntups
169         e = zeros((n,2))
170         for episode in range(numEpisodes):
171             trainee = connect4(w = w,ntups=ntups)
172             phi = zeros((n,2))
173             counter = 0
174             while trainee.win is None:
175                 if self.verbose:
176                     print(trainee)
177                     print("Learning episode %d of %d" %(
                                episode, numEpisodes))
```

```
178                 counter += 1
179                 legalMoves = trainee.getLegalMoves()
180                 if (rand(1)[0] < eps):
181                     move = choice(legalMoves)
182                 else:
183                     move = trainee.boardInversionHeuristic(
                            trainee)
184                 player = trainee.currPlayer
185                 playerInd = (player+1)//2
186                 trainee.simulate(move)
187                 phinew = player*trainee.getFeatures()
188                 delta = 0 + gamma*logsig(dot(w,phinew)) -
                        logsig((dot(w,phi[:,playerInd])))
189                 e[:,playerInd] = gamma*lamb*e[:,playerInd]\
190                                  +dlogsig(dot(w,phi[:,
                                      playerInd]))*transpose(
                                      phi[:,playerInd])
191                 if(counter > 2):
192                     w +=  alpha*delta*e[:,playerInd]
193                 phi[:,playerInd] = phinew
194             if trainee.win == 1:
195                 reward = 1
196             elif trainee.win == -1:
197                 reward = 0
198             else:
199                 reward = 0.5
200
201             deltapos = reward - logsig(dot(w,phi[:,0]))
202             deltaneg = (1-reward) - logsig(dot(w,phi[:,1]))
203             w += alpha*deltapos*e[:,0]
204             w += alpha*deltaneg*e[:,1]
205             if not self.verbose:
206                 print("Learning episode %d of %d" %(episode
                        , numEpisodes),end="\r")
207         self.w = w
208         return w
209
210
211
212
```

```
213     def boardInversionHeuristic(self,sim):
214         features = sim.getFeatures()
215         w = self.w
216         legalMoves = sim.getLegalMoves()
217         if len(legalMoves) == 0:
218             return float("-inf") #Don't want to make
                    illegal move.
219         phi = zeros((len(w),len(legalMoves)))
220         for i,move in enumerate(legalMoves):
221             cp = sim.copy()
222             cp.simulate(move)
223             phi[:,i] = cp.getFeatures()
224         bestMove= legalMoves[argmax(sim.currPlayer*tansig(
                dot(w,phi)))]
225         return bestMove
226 
227 
228     def monteCarloPlay(self):
229         legalMoves = self.getLegalMoves()
230         rewards = [self.rollout(move) for move in
                legalMoves]
231         bestMove = legalMoves[argmax(rewards)]
232         return self.simulate(bestMove)
233 
234     def monteCarloPlayPlus(self):
235         obvMove = self.obviousMove()
236         if obvMove is not None:
237             return self.simulate(obvMove)
238         legalMoves = self.getLegalMoves()
239         rewards = [self.rolloutPlus(move) for move in
                legalMoves]
240         bestMove = legalMoves[argmax(rewards)]
241         return self.simulate(bestMove)
242 
243 
244     def heuristic(self,sim):
245         #Our heuristic here is very similar to
246         #The one used by the monte carlo player,
247         #Except that we use distance from the win
248         #As a metric as well, such that a
```

```
249             #win in the next move is much better than a win
250             #Two moves later, and a loss in the next move
251             #Is worse than a loss two moves later
252             #And we average over 7 games.
253             #This would cause us to block if possible,
254             #and win if possible (how much depends on alpha)
255             #but also block in the better way if possible
256             #alpha controls how fast the function drops
257             alpha = self.alpha
258             #alpha = 0.9
259             reward = 0
260             player = currPlayer
261             if sim.win is not None:
262                 #Large win or punishment for win or loss
263                 return sim.win*currPlayer*100
264             for i in range(5):
265                 simulation = sim.copy()
266                 win = None
267                 count = 0
268                 while win is None:
269                     state, win = simulation.makeRandomMove()
270                     count += 1
271                 #We exponent alpha to the power of moves.
272                 #Since it is < 1, it becomes smaller
273                 #The further away we are.
274                 reward += 10*win*(alpha**count)
275             return reward*currPlayer
276 
277 
278         def heuristicPlay(self, heuristicFunc = None):
279             if heuristicFunc is None:
280                 heuristicFunc = self.heuristic
281             legalMoves = self.getLegalMoves()
282             simulation = self.copy()
283             heuristicVals = []
284             for move in legalMoves:
285                 sim = simulation.copy()
286                 state,win = sim.simulate(move)
287                 heuristicVals.append(heuristicFunc(sim))
288 
```

```
289         bestMove = legalMoves[argmax(heuristicVals)]
290         return self.simulate(bestMove)
291
292
293
294
295
296     def __str__(self):
297         oldOptions = get_printoptions()
298         if self.color:
299             set_printoptions(linewidth = 100,formatter = {"
                    float":colorPrinter})
300             BLUE = '\033[34m'
301             ENDC = '\033[0m'
302             header = "\n ["
303             header += BLUE +" 0"
304             for i in range(1,self.shape[1]-1):
305                 header += "   %d" %(i,)
306             header += "   " + str(self.shape[1]-1) + ENDC
307             header += " ]\n\n"
308             end = "\n"
309         else:
310             header = ""
311             end = ""
312         s = header + str(self.state) + end
313         set_printoptions(**oldOptions)
314         return  s
315
316
317     # 0 is draw, 1,-1 is win for that colour, None is game
            not over
318     def checkwin(self, state):
319         s = state.flatten()
320         for winpos in winningLocations:
321             posSum = sum(s[winpos])
322             if abs(posSum) == 4:
323                 return sign(posSum)
324
325         if all(state,1)[0]:
326             return 0
```

```
327             return None
328
329     def resultsToString(results):
330         st = ""
331         for color in [-1,0,1]:
332             st += colors[color] + ": "+ str(results[color]) +\
333                   " "  if color in results else ""
334         return st
335
336     def colorPrinter(x):
337         GREEN = '\033[32m'
338         YELLOW = '\033[33m'
339         BLACK = '\033[30m'
340         BLUE = '\033[34m'
341         RED = '\033[31m'
342         ENDC = '\033[0m'
343
344         START = "" if x ==0 else BLUE if x < 0 else RED
345         ADDSPACE = ' ' if x >= 0 else ""
346         if x == 0:
347             return "   "
348         return START + ADDSPACE+ ("%.0f." % x) + ENDC
349
350
351     if __name__=="__main__":
352         color = True
353         GREEN = '\033[32m'
354         BLUE = '\033[34m'
355         RED = '\033[31m'
356         ENDC = '\033[0m'
357
358         colors = {-1: BLUE+"Blue"+ENDC, 0: "Ties", 1: RED+"Red"
               +ENDC}
359         def getInput(currPlayer):
360             col = int(input(colors[currPlayer] + " player, 
                   enter column: "))
361             while col not in range(7):
362                 print("Incorrect move!")
363                 col = int(input(colors[currPlayer] + " player, 
                       enter column: "))
```

```
364             return col
365
366
367         play = True
368         numPlayers = -1
369         verbose = bool(int(input("Display board? 1/0: ").split
                ()[-1]))
370         dispTrain = bool(int(input("Display Training? 1/0: ").
                split()[-1]))
371         while numPlayers not in range(3):
372             numPlayers = int(input("Enter number of players: ")
                    .split()[-1])
373         if numPlayers < 2:
374             posopponents = ["random","MC","MCPlus","Heuristic",
                    "BoardInv","N-tups"]
375             opponentChoices = {}
376             print("Available opponents:")
377             for i,opponent in enumerate(posopponents):
378                 print("%d. %s" % (i, opponent))
379             opponentChoices[-1] = posopponents[int(input("Pick
                    opponent for "+\
380                                                     colors
                                                        [-1]
                                                         +"
                                                        : "
                                                        ).
                                                        split
                                                        ()
                                                        [-1])
                                                        ]
381             print(opponentChoices[-1]+" chosen")
382             if numPlayers == 0:
383                 opponentChoices[1] = posopponents[int(input("
                        Pick opponent for "+\
384                                                     colors
                                                        [1]

                                                        +
                                                        "
                                                        :
```

```
                                                                    ␣"
                                                                    )
                                                                    .
                                                                    split
                                                                    ()
                                                                    [-1])
                                                                    ]
385             print(opponentChoices[1]+"␣chosen")
386
387     results = {-1: 0, 0: 0, 1: 0}
388     totalRoundsToPlay = 60
389     playedRounds = 0
390     interactive = False
391     startingPlayer = 1
392     w = None
393     ntups = None
394     while play:
395         c4 = connect4(currPlayer = startingPlayer,verbose=
                dispTrain)
396         if w is not None:
397             c4.w = w
398             c4.ntups = ntups
399         elif "BoardInv" or "N-tups" in opponentChoices.
                values():
400             print("TD␣chosen,␣learning␣w")
401             episodes = int(input("Enter␣episodes␣for␣TD:␣")
                    .split()[-1])
402             print(episodes)
403             lamb = float(input("Enter␣lambda␣for␣TD:␣").
                    split()[-1])
404             print(lamb)
405             if "N-tups" in opponentChoices.values():
406                 w = c4.learnWFromTd(numEpisodes = episodes,
                        lamb = lamb)
407                 ntups = c4.ntups
408             else:
409                 w = c4.learnWFromTd(numEpisodes = episodes,
```

```
410                                         lamb = lamb,useNtups =
                                                False)
411         #c4.w = givenw
412         boardInv = lambda : c4.heuristicPlay(heuristicFunc
               = c4.boardInversionHeuristic)
413         opponentActions = {"random": c4.makeRandomMove,
414                            "MC": c4.monteCarloPlay,
415                            "MCPlus": c4.monteCarloPlayPlus,
416                            "Heuristic": c4.heuristicPlay,
417                            "BoardInv": boardInv,
418                            "N-tups":boardInv}
419         win = None
420         while win is None:
421             if verbose:
422                 print(c4)
423                 print("Score: ")
424                 print(resultsToString(results))
425             currPlayer = c4.currPlayer
426             if numPlayers == 0:
427                 if verbose:
428                     print("Method: ")
429                     print(resultsToString(opponentChoices))
430                 state, win = opponentActions[
                       opponentChoices[currPlayer]]()
431             elif numPlayers == 1:
432                 if currPlayer == 1:
433                     col = getInput(currPlayer)
434                     state, win = c4.simulate(col)
435                 else:
436                     state, win = opponentActions[
                           opponentChoices[currPlayer]]()
437             else:
438                 col = getInput(currPlayer)
439                 state, win = c4.simulate(col)
440
441
442
443         if verbose:
444             print(c4)
445         else:
```

```
446
447             print(resultsToString(results))
448         if win != 0:
449             print( colors[win] +" player wins!")
450         else:
451             print(RED + "Tie!" +ENDC)
452         results[win] += 1
453         playedRounds += 1
454         if interactive:
455             play = input("Play Again, Y/N? ") not in ["N","
                    n","no"]
456         else:
457             play = playedRounds < totalRoundsToPlay
458             if playedRounds == totalRoundsToPlay/2:
459                 print("Swapping players")
460                 startingPlayer = -1
461     print("Final score: ")
462     print(resultsToString(results))
463     if opponentChoices:
464         print("Methods: ")
465         print(resultsToString(opponentChoices))
```

../util.py

```
1  from pylab import *
2
3  def logsig(n):
4      return 1/(1+exp(-n))
5
6  def dlogsig(n):
7      a = logsig(n)
8      return a*(1-a)
9
10 def tansig(n):
11     return tanh(n)
12
13 givenw = [3,4,5,5,4,3,4,6,8,8,6,4,5,8,11,11,8,5,7,10,13,
14       13,10,7,5,8,11,11,8,5,4,6,8,8,6,4,3,4,5,5,4,3]
15
16 winningLocations = array(
17 [[1, 2, 3, 4],
```

```
18 |[  2,  3,  4,  5],
19 |[  3,  4,  5,  6],
20 |[  7,  8,  9, 10],
21 |[  8,  9, 10, 11],
22 |[  9, 10, 11, 12],
23 |[ 13, 14, 15, 16],
24 |[ 14, 15, 16, 17],
25 |[ 15, 16, 17, 18],
26 |[ 19, 20, 21, 22],
27 |[ 20, 21, 22, 23],
28 |[ 21, 22, 23, 24],
29 |[ 25, 26, 27, 28],
30 |[ 26, 27, 28, 29],
31 |[ 27, 28, 29, 30],
32 |[ 31, 32, 33, 34],
33 |[ 32, 33, 34, 35],
34 |[ 33, 34, 35, 36],
35 |[ 37, 38, 39, 40],
36 |[ 38, 39, 40, 41],
37 |[ 39, 40, 41, 42],
38 |[  1,  7, 13, 19],
39 |[  7, 13, 19, 25],
40 |[ 13, 19, 25, 31],
41 |[ 19, 25, 31, 37],
42 |[  2,  8, 14, 20],
43 |[  8, 14, 20, 26],
44 |[ 14, 20, 26, 32],
45 |[ 20, 26, 32, 38],
46 |[  3,  9, 15, 21],
47 |[  9, 15, 21, 27],
48 |[ 15, 21, 27, 33],
49 |[ 21, 27, 33, 39],
50 |[  4, 10, 16, 22],
51 |[ 10, 16, 22, 28],
52 |[ 16, 22, 28, 34],
53 |[ 22, 28, 34, 40],
54 |[  5, 11, 17, 23],
55 |[ 11, 17, 23, 29],
56 |[ 17, 23, 29, 35],
57 |[ 23, 29, 35, 41],
```

```
58 [ 6,12,18,24],
59 [12,18,24,30],
60 [18,24,30,36],
61 [24,30,36,42],
62 [ 4, 9,14,19],
63 [ 5,10,15,20],
64 [10,15,20,25],
65 [ 6,11,16,21],
66 [11,16,21,26],
67 [16,21,26,31],
68 [12,17,22,27],
69 [17,22,27,32],
70 [22,27,32,37],
71 [18,23,28,33],
72 [23,28,33,38],
73 [24,29,34,39],
74 [ 3,10,17,24],
75 [ 2, 9,16,23],
76 [ 9,16,23,30],
77 [ 1, 8,15,22],
78 [ 8,15,22,29],
79 [15,22,29,36],
80 [ 7,14,21,28],
81 [14,21,28,35],
82 [21,28,35,42],
83 [13,20,27,34],
84 [20,27,34,41],
85 [19,26,33,40]]) - 1
86
87
88 def winningPosistion():
89     C = zeros((69,42));
90     for i in range(size(location,0)):
91         C[i,winningLocations[i,:]] = 1;
92     return C,winningLocations
```