Lab4

September 15, 2021

[13]:
```python
from math import exp
import random
from copy import deepcopy
import time
```

[14]:
```python
N_QUEENS = 8
temp = 2000
```

`make_board(n)` function creates a n rows and places a queen in one of the columns of each of the rows. It stores the row, column(position) of every queen in chess_board dictionary and returns that.

[15]:
```python
def make_board(n):
    '''Create a chess board with a queen on a row'''
    chess_board = {}
    temp = list(range(n))
    # shuffle to make sure it is random
    random.shuffle(temp)
    column = 0

    while len(temp) > 0:
        row = random.choice(temp)
        chess_board[column] = row
        temp.remove(row)
        column += 1
    del temp
    return chess_board
```

[16]:
```python
def calculate_threat(n):
    '''Combination formula. It is choosing two queens in n queens'''
    if n < 2:
        return 0
    if n == 2:
        return 1
    return (n - 1) * n / 2
```

`cost(chess_board)` calculates how many queens are not in same position

[17]:
```python
def cost(chess_board):
    '''Calculate how many pairs of threaten queen'''
    threat = 0
```

```
    m_chessboard = {}
    a_chessboard = {}

    for column in chess_board:
        temp_m = column - chess_board[column]
        temp_a = column + chess_board[column]
        if temp_m not in m_chessboard:
            m_chessboard[temp_m] = 1
        else:
            m_chessboard[temp_m] += 1
        if temp_a not in a_chessboard:
            a_chessboard[temp_a] = 1
        else:
            a_chessboard[temp_a] += 1

    for i in m_chessboard:
        threat += calculate_threat(m_chessboard[i])
    del m_chessboard

    for i in a_chessboard:
        threat += calculate_threat(a_chessboard[i])
    del a_chessboard

    return threat
```

Simulated annealing (SA) is a probabilistic technique for approximating the global optimum of a given function. Specifically, it is a metaheuristic to approximate global optimization in a large search space for an optimization problem.

At each step, the simulated annealing heuristic considers some neighboring state s* of the current state s, and probabilistically decides between moving the system to state s* or staying in-state s. These probabilities ultimately lead the system to move to states of lower energy. Typically this step is repeated until the system reaches a state that is good enough for the application, or until a given computation budget has been exhausted.

0.1 Pseudocode for general simulated annealing

The following pseudocode presents the simulated annealing heuristic as described above. It starts from a state s0 and continues until a maximum of kmax steps have been taken. In the process, the call neighbour(s) should generate a randomly chosen neighbour of a given state s; the call random(0, 1) should pick and return a value in the range [0, 1], uniformly at random. The annealing schedule is defined by the call temperature(r), which should yield the temperature to use, given the fraction r of the time budget that has been expended so far.

```
Let s = s0
For k = 0 through kmax (exclusive):
  T <- temperature( 1 - (k+1)/kmax )
  Pick a random neighbour, snew <- neighbour(s)
  If P(E(s), E(snew), T) >= random(0, 1):
```

```
    s <- snew
Output: the final state s
```

[18]:
```python
def simulated_annealing():
    '''Simulated Annealing'''
    solution_found = False
    answer = make_board(N_QUEENS)

    # To avoid recounting when can not find a better state
    cost_answer = cost(answer)

    t = temp
    sch = 0.99

    while t > 0:
        t *= sch
        successor = deepcopy(answer)
        while True:
            index_1 = random.randrange(0, N_QUEENS - 1)
            index_2 = random.randrange(0, N_QUEENS - 1)
            if index_1 != index_2:
                break
        successor[index_1], successor[index_2] = successor[index_2], \
            successor[index_1]  # swap two chosen queens
        delta = cost(successor) - cost_answer
        if delta < 0 or random.uniform(0, 1) < exp(-delta / t):
            answer = deepcopy(successor)
            cost_answer = cost(answer)
        if cost_answer == 0:
            solution_found = True
            print_chess_board(answer)
            break
    if solution_found is False:
        print("Failed")
```

[19]:
```python
def print_chess_board(board):
    '''Print the chess board'''
    brd = [['_' for j in range(0,N_QUEENS)] for i in range(0,N_QUEENS)]
    for column, row in board.items():
        brd[row][column] = 'Q'
        print("{} => {}".format(column, row))
    for row in brd:
        print(row)
```

[20]:
```python
def main():
    start = time.time()
    simulated_annealing()
    print("Runtime in second:", time.time() - start)
```

[21]:
```
main()
```

```
0 => 3
1 => 6
2 => 4
3 => 1
4 => 5
5 => 0
6 => 2
7 => 7
['_', '_', '_', '_', '_', 'Q', '_', '_']
['_', '_', '_', 'Q', '_', '_', '_', '_']
['_', '_', '_', '_', '_', '_', 'Q', '_']
['Q', '_', '_', '_', '_', '_', '_', '_']
['_', '_', 'Q', '_', '_', '_', '_', '_']
['_', '_', '_', '_', 'Q', '_', '_', '_']
['_', 'Q', '_', '_', '_', '_', '_', '_']
['_', '_', '_', '_', '_', '_', '_', 'Q']
Runtime in second: 0.01335453987121582
```

[ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

[ ]:
```
```