hillClimbing_and_Simulated_Annealing_Implementation

June 18, 2021

[44]:
```
import random
class HillClimb:
    def __init__(self , grid_size):
        self.grid_size = grid_size
        self.grid = [None] * grid_size
        for i in range(grid_size):
            self.grid[i] = [None] * grid_size

        self.fill_grid()
        self.show_grid()
        v = self.hill_climb()
        print(v)

    def fill_grid(self):
        self.grid = [
            [10, 3,  4,   6,  23],
             [9,  32, 12,  2,  34],
             [7,  8,  100, 21, 11],
             [18, 67, 55,  89, 90],
             [22, 33, 14,  44, 110]
        ]
        #for i in range(self.grid_size):
            #for j in range(self.grid_size):
                #self.grid[i][j] = random.randint(0,100)

    def show_grid(self):
        for i in range(self.grid_size):
            print(self.grid[i])

    def get_first(self,tu):
        return tu[0]

    def get_successor(self,index):
        x  = index[0]  # x-axis
        y  = index[1]  # y-axis

        neighbours = []
```

```
        up    = y - 1
        down  = y + 1
        left  = x + 1
        right = x - 1

        try:        # for down
            self.grid[x][down]

        except IndexError:
            pass

        else:
            neighbours.append((self.grid[x][down] , [x,down]))

            # ------------------------------------#

        try:
            self.grid[left][y]
        except IndexError:
            pass

        else:
            neighbours.append((self.grid[left][y] , [left,y]))

            # ------------------------------------#

        if up >= 0:   # if positive
            #index = (x,up)
            neighbours.append((self.grid[x][up] , [x,up]))

            # ------------------------------------#

        if right >= 0:   # if positive
            #index = (right,y)
            neighbours.append((self.grid[right][y] , [right,y]))

            # ------------------------------------#
        neighbours.sort(key = self.get_first , reverse = True)

        return neighbours[0][1]



    def hill_climb(self):
        i , j = 2,2    #initial node
        #initial = self.grid[2][2]
```

```
        visited = []
        queue = [[i,j]]
        while queue:
            current_index = queue.pop(0)
            #current_node = self.grid[current_index[0]][current_index[1]]
            print("Current State : ",self.
↪grid[current_index[0]][current_index[1]],"at",current_index)
            if current_index in visited:
                continue

            successor = self.get_successor(current_index)
            visited.append(current_index)
            current_index = successor
            queue.append(current_index)

        return "heighest value : " + str(self.
↪grid[current_index[0]][current_index[1]]) + " at "+str(current_index)
```

[45]:
```
print(HillClimb(5))
```

```
[10, 3, 4, 6, 23]
[9, 32, 12, 2, 34]
[7, 8, 100, 21, 11]
[18, 67, 55, 89, 90]
[22, 33, 14, 44, 110]
Current State :  100 at [2, 2]
Current State :  55 at [3, 2]
Current State :  100 at [2, 2]
heighest value : 100 at [2, 2]
<__main__.HillClimb object at 0x000002479AE27400>
```

[58]:
```
import random
class Simulated:
    def __init__(self , grid_size):
        self.grid_size = grid_size
        self.grid = [None] * grid_size
        for i in range(grid_size):
            self.grid[i] = [None] * grid_size

        self.fill_grid()
        self.show_grid()
        #v = self.hill_climb()
        v = self.simulated()
        print("Solution at ",v,"Value = ",self.grid[v[0]][v[1]])

    def fill_grid(self):
        for i in range(self.grid_size):
```

```
            for j in range(self.grid_size):
                self.grid[i][j] = random.randint(0,100)

    def show_grid(self):
        for i in range(self.grid_size):
            print(self.grid[i])

    def get_first(self,tu):
        return tu[0]


    def get_random_index(self):
        i = random.randint(0,self.grid_size-1)
        j = random.randint(0,self.grid_size-1)

        return (i,j)

    def schedule(self,t):
        return (pow(10,7) - t)

    def get_nieghbour_with_prob(self,index , T):
        x  = index[0]  # x-axis
        y  = index[1]  # y-axis

        neighbours = []

        current_val = self.grid[x][y]

        up    = y - 1
        down  = y + 1
        left  = x + 1
        right = x - 1

        try:        # for down
            self.grid[x][down]

        except IndexError:
            pass

        else:
            n = self.grid[x][down] #next
            d_E = n - current_val
            power = d_E / T
            prob = pow(2.71,power)
            neighbours.append((prob, [x,down]))

            # ------------------------------------#
```

```
        try:
            self.grid[left][y]
        except IndexError:
            pass

        else:
            n = self.grid[left][y] #next
            d_E = n - current_val
            power = d_E / T
            prob = pow(2.71,power)

            neighbours.append((prob , [left,y]))

            # ---------------------------------------#

        if up >= 0:   # if positive
            #index = (x,up)

            n = self.grid[x][up] #next
            d_E = n - current_val
            power = d_E / T
            prob = pow(2.71,power)

            neighbours.append((prob , [x,up]))

            # ---------------------------------------#

        if right >= 0:   # if positive
            #index = (right,y)
            n = self.grid[right][y] #next
            d_E = n - current_val
            power = d_E / T
            prob = pow(2.71,power)

            neighbours.append((prob , [right,y]))

            # ---------------------------------------#
        neighbours.sort(key = self.get_first , reverse = True)

        return neighbours[0][1]

    def simulated(self):
        current_index = [2,2]
        t = 0
        while True:
            T = self.schedule(t)
```

```
            if T == 0:
                return current_index

            _next = self.get_random_index()

            current_val = self.grid[current_index[0]][current_index[1]]
            _next_val   = self.grid[_next[0]][_next[1]]
            delta_E = _next_val - current_val

            if delta_E > 0:
                current_index = _next
            else:
                current_index = self.get_nieghbour_with_prob(current_index , T)

            t += 1
```

[59]:
```
Simulated(5)
```

```
[71, 86, 80, 91, 35]
[63, 81, 59, 29, 97]
[32, 65, 8, 95, 13]
[16, 69, 40, 4, 68]
[26, 96, 7, 99, 70]
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-59-ba7187a2e089> in <module>
----> 1 Simulated(5)

<ipython-input-58-60d8010c7221> in __init__(self, grid_size)
     10         self.show_grid()
     11         #v = self.hill_climb()
---> 12         v = self.simulated()
     13         print("Solution at ",v,"Value = ",self.grid[v[0]][v[1]])
     14

<ipython-input-58-60d8010c7221> in simulated(self)
    123                 current_index = self.
 ↪get_nieghbour_with_prob(current_index , T)
    124
--> 125             t += 1

KeyboardInterrupt:
```

[53]:
```
import random
```

```
#random.seed()
print(random.randint(1, 9))
print(random.randint(1, 9))
```

```
6
1
```

[ ]: