AppliedW10

May 3, 2024

1 Question 1

Discuss your assignment with your Applied Class demonstrators.

2 Question 2

The idea behind Stochastic Hill Climbing is to choose randomly from all uphill (Downhill) moves in the vicinity of the current solution. The probabilities of choosing may depend on steepness of the uphill move.

3 Question 3

The algorithm can be expressed as follows:

```
Input:
Max temperature $T_{max}$
cooling factor $\alpha$
objective function $f$
initial solution $x_0$
Result: A good enough solution $x^*$
$T \leftarrow T_{max}$;
$x^* \leftarrow x_0$;
while $T > \epsilon$ do
    $i \leftarrow 0$;
    while $i <$ number of perturbations do
        $\tilde{x} \leftarrow$ Perturbation($x^*$);
        if Metropolis criterion accepted then
            $x^* \leftarrow \tilde{x}$;
        end
        $i \leftarrow i + 1$;
    end
    $T \leftarrow \alpha T$
end
return $x^*$
```

Algorithm 1: Sketch for simulated anneling

Note that there are many ways to answer these questions, the solution provided here is just one example.

- Assign each item an index. Potential solution can be represented as a bitlist where the i-th item is 1 if we take the item and 0 if we do not.
- We could structure the fitness function to reward high value solutions but highly punish solutions that are too heavy.
- One possible fitness function, F, (that we would want to maximise):

$$F = \begin{cases} \text{Value} & \text{if Weight} \leq W \\ 0 & \text{if Weight} > W \end{cases}$$

- We can perturb a potential solution by randomly choosing a subset of items and swap whether we are taking those items or not. The size of the subset can be determine in different ways. For example, could always select a constant portion of items, or maybe just a single item.

[3]:
```python
import numpy as np
from matplotlib import pyplot as plt
from scipy.linalg import norm

def norm2(x):
    return norm(x, ord=2)

random_state = 0
```

3.1 Knapsack Generator

[33]:
```python
class knapsack_generator:
    def __init__(self, n=20, v_low=1, v_high=10, w_low=1, w_high=10,␣
↪random_state=None):
        self.n=n
        self.v_low=v_low
        self.v_high=v_high
        self.w_low=w_low
        self.w_high=w_high
        self.random_state=random_state
        np.random.seed(self.random_state)

    def __call__(self):
        V = np.random.randint(size=self.n, low=self.v_low, high=self.v_high)
        W = np.random.randint(size=self.n, low=self.w_low, high=self.w_high)
        items_forming_cap = np.random.choice(self.n, size=int(np.sqrt(self.n)))
        k = sum(W[items_forming_cap])
        return V, W, k

def knapsack(n=20, v_range=[1, 10], w_range=[1, 10], random_state=None):
    V = np.random.randint(size=n, low=v_range[0], high=v_range[1])
    W = np.random.randint(size=n, low=w_range[0], high=w_range[1])
    items_forming_cap = np.random.choice(n, size=int(np.sqrt(n)))
    k = sum(W[items_forming_cap])
    return V, W, k
```

[34]:
```
# V, W, k = knapsack(n=20)
# V, W, k
n=20
v_low=1
v_high=10
w_low=1
w_high=10
random_state=0
knap = knapsack_generator(n=n, v_low=v_low, v_high=v_high, w_low=w_low,␣
 ↪w_high=w_high, random_state=random_state)
V, W, k = knap()
V, W, k
```

[34]:
```
(array([6, 1, 4, 4, 8, 4, 6, 3, 5, 8, 7, 9, 9, 2, 7, 8, 8, 9, 2, 6]),
 array([9, 5, 4, 1, 4, 6, 1, 3, 4, 9, 2, 4, 4, 4, 8, 1, 2, 1, 5, 8]),
 14)
```

3.2 Simulated Annealing

[76]:
```
class SA_knapsack:
    def __init__(self, W, V, k, alpha=0.8, T0 = 1, T_min=0.1, per_T=10,␣
 ↪random_state=None):
        self.W = W
        self.V = V
        self.k = k
        self.T0 = T0
        self.T_min = T_min
        self.per_T = per_T
        self.alpha = alpha
        self.random_state = random_state
        np.random.seed(self.random_state)

    def peturb(self, x, n_flip=1):
        flip = np.random.choice(len(x), size=n_flip, replace=False)
        x_pert = x.copy()
        x_pert[flip] = 1 - x_pert[flip]
        return x_pert

    def sample(self, x):
        return np.zeros_like(x) # start with empty solution and peturb from␣
 ↪there. empty will always be valid!
```

```
        return np.random.choice(2, size=len(x))

    def fitness(self, x):
        if np.sum(self.W[x==1]) > self.k:
            return 0
        else:
            return np.sum(self.V[x==1])

    def metropolis_max(self, x, x_new, T):
        if self.fitness(x_new) > self.fitness(x):
            return True
        return np.random.random() > np.exp((self.fitness(x) - self.
↪fitness(x_new))/T)

    def solve(self, verbose=False):
        x = self.sample(W)
        T = self.T0
        t_count = 0
        while T > self.T_min:
            if verbose: print('T={0}, fitness={1}, x={2}'.format(T, self.
↪fitness(x), x))
            for i in range(self.per_T):
                x_new = self.peturb(x, n_flip=2)
                if self.metropolis_max(x, x_new, T):
                    x = x_new.copy()
            T = self.alpha*T
            t_count += 1
        if verbose: print('{0} solutions checked'.format(t_count*self.per_T))
        return x
```

[78]:
```
SA = SA_knapsack(W=W, V=V, k=k, alpha=0.98, T0 = 1, T_min=0.01, per_T=100,␣
↪random_state=random_state)
x = SA.solve(verbose=False)
x, SA.fitness(x)
```

```
/tmp/ipykernel_161530/1608091459.py:32: RuntimeWarning: overflow encountered in
exp
  return np.random.random() > np.exp((self.fitness(x) - self.fitness(x_new))/T)
```

[78]:
```
(array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0]), 50)
```

[79]:
```
np.dot(x, W), np.dot(x, V)
```

[79]:
```
(14, 50)
```

4 Question 4

[4]:
```
import numpy as np
from matplotlib import pyplot as plt
```

4.1 Transition Matrix

[5]:
```
p = 0.5
M = np.array([
    [0, p, 1-p, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], # (0, 0) - 0
    [0, 0, 0, p, 1-p, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], # (15, 0) - 1
    [0, 0, 0, 0, p, 1-p, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], # (0, 15) - 2
    [0, 0, 0, 0, 0, 0, p, 1-p, 0, 0, 0, 0, 0, 0, 0, 0, 0], # (30, 0) - 3
    [0, 0, 0, 0, 0, 0, 0, p, 1-p, 0, 0, 0, 0, 0, 0, 0, 0], # (15, 15) - 4
    [0, 0, 0, 0, 0, 0, 0, 0, p, 1-p, 0, 0, 0, 0, 0, 0, 0], # (0, 30) - 5
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1-p, 0, p, 0, 0, 0, 0], # (40, 0) - 6
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, p, 0, 0, 0, 1-p, 0, 0], # (30, 15) - 7
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1-p, 0, 0, p, 0, 0], # (15, 30) - 8
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, p, 0, 0, 0, 0, 1-p], # (0, 40) - 9
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, p, 1-p, 0, 0, 0], # (40, 15) - 10
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, p, 1-p], # (15, 40) - 11
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0], # (Win, 0) - 12
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, p, 0, 1-p, 0, 0], # (40, 30) - 13 <-␣
 ↪Advantage A
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, p, 0, 1-p, 0], # (30, 30) - 14 <-␣
 ↪Deuce
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, p, 0, 1-p], # (30, 40) - 15 <-␣
 ↪Advantage B
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1], # (0, Win) - 16
])
```

4.2 Canonical Form

[6]:
```
def extract_canonical_components(M):
    D = np.diag(M)
    absorbing, transient = np.where(D==1)[0], np.where(D!=1)[0]
    I = M[absorbing, :][:, absorbing]
    Q = M[transient, :][:, transient]
    R = M[transient, :][:, absorbing]
    O = M[absorbing, :][:, transient]
    return I, Q, R, O, absorbing, transient
```

[9]:
```
I, Q, R, O, absorbing, transient = extract_canonical_components(M)
Q, absorbing, transient
```

[9]:
```
(array([[0. , 0.5, 0.5, 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ,
         0. , 0. ],
        [0. , 0. , 0. , 0.5, 0.5, 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ,
         0. , 0. ],
        [0. , 0. , 0. , 0. , 0.5, 0.5, 0. , 0. , 0. , 0. , 0. , 0. , 0. ,
         0. , 0. ],
        [0. , 0. , 0. , 0. , 0. , 0. , 0.5, 0.5, 0. , 0. , 0. , 0. , 0. ,
         0. , 0. ],
        [0. , 0. , 0. , 0. , 0. , 0. , 0. , 0.5, 0.5, 0. , 0. , 0. , 0. ,
         0. , 0. ],
        [0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0.5, 0.5, 0. , 0. , 0. ,
         0. , 0. ],
        [0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0.5, 0. , 0. ,
         0. , 0. ],
        [0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0.5, 0. , 0. ,
         0.5, 0. ],
        [0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0.5, 0. ,
         0.5, 0. ],
        [0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0.5, 0. ,
         0. , 0. ],
        [0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0.5,
         0. , 0. ],
        [0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ,
         0. , 0.5],
        [0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ,
         0.5, 0. ],
        [0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0.5,
         0. , 0.5],
        [0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ,
         0.5, 0. ]]),
 array([12, 16]),
 array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 13, 14, 15]))
```

4.3 Fundamental Matrix

[10]:
```python
def fundamental_matrix(M):
    I, Q, R, O, absorbing, transient = extract_canonical_components(M)
    N = np.linalg.inv(np.eye(len(Q))-Q)
    return N, transient
```

[11]:
```python
N, transient = fundamental_matrix(M)
N
```

[11]:
```
array([[1.    , 0.5   , 0.5   , 0.25  , 0.5   , 0.25  , 0.125 , 0.375 ,
        0.375 , 0.125 , 0.25  , 0.25  , 0.625 , 1.    , 0.625 ],
       [0.    , 1.    , 0.    , 0.5   , 0.5   , 0.    , 0.25  , 0.5   ,
        0.25  , 0.    , 0.375 , 0.125 , 0.6875, 1.    , 0.5625],
       [0.    , 0.    , 1.    , 0.    , 0.5   , 0.5   , 0.    , 0.25  ,
        0.5   , 0.25  , 0.125 , 0.375 , 0.5625, 1.    , 0.6875],
       [0.    , 0.    , 0.    , 1.    , 0.    , 0.    , 0.5   , 0.5   ,
        0.    , 0.    , 0.5   , 0.    , 0.625 , 0.75  , 0.375 ],
       [0.    , 0.    , 0.    , 0.    , 1.    , 0.    , 0.    , 0.5   ,
        0.5   , 0.    , 0.25  , 0.25  , 0.75  , 1.25  , 0.75  ],
       [0.    , 0.    , 0.    , 0.    , 0.    , 1.    , 0.    , 0.    ,
        0.5   , 0.5   , 0.    , 0.5   , 0.375 , 0.75  , 0.625 ],
       [0.    , 0.    , 0.    , 0.    , 0.    , 0.    , 1.    , 0.    ,
        0.    , 0.    , 0.5   , 0.    , 0.375 , 0.25  , 0.125 ],
       [0.    , 0.    , 0.    , 0.    , 0.    , 0.    , 0.    , 1.    ,
        0.    , 0.    , 0.5   , 0.    , 0.875 , 1.25  , 0.625 ],
       [0.    , 0.    , 0.    , 0.    , 0.    , 0.    , 0.    , 0.    ,
        1.    , 0.    , 0.    , 0.5   , 0.625 , 1.25  , 0.875 ],
       [0.    , 0.    , 0.    , 0.    , 0.    , 0.    , 0.    , 0.    ,
        0.    , 1.    , 0.    , 0.5   , 0.125 , 0.25  , 0.375 ],
       [0.    , 0.    , 0.    , 0.    , 0.    , 0.    , 0.    , 0.    ,
        0.    , 0.    , 1.    , 0.    , 0.75  , 0.5   , 0.25  ],
       [0.    , 0.    , 0.    , 0.    , 0.    , 0.    , 0.    , 0.    ,
        0.    , 0.    , 0.    , 1.    , 0.25  , 0.5   , 0.75  ],
       [0.    , 0.    , 0.    , 0.    , 0.    , 0.    , 0.    , 0.    ,
        0.    , 0.    , 0.    , 0.    , 1.5   , 1.    , 0.5   ],
       [0.    , 0.    , 0.    , 0.    , 0.    , 0.    , 0.    , 0.    ,
        0.    , 0.    , 0.    , 0.    , 1.    , 2.    , 1.    ],
       [0.    , 0.    , 0.    , 0.    , 0.    , 0.    , 0.    , 0.    ,
        0.    , 0.    , 0.    , 0.    , 0.5   , 1.    , 1.5   ]])
```

4.4 Absorbtion Times and Probabilities

[12]:
```
absorbtion_times = np.dot(N, np.ones(len(N)))
total_time = np.sum(absorbtion_times)
absorbtion_times, total_time
```

[12]:
```
(array([6.75, 5.75, 5.75, 4.25, 5.25, 4.25, 2.25, 4.25, 4.25, 2.25, 2.5 ,
        2.5 , 3.  , 4.  , 3.  ]),
 60.0)
```

[13]:
```
absorbtion_probabilities = np.dot(N, R)
absorbtion_probabilities
```

[13]:
```
array([[0.5    , 0.5    ],
       [0.65625, 0.34375],
       [0.34375, 0.65625],
       [0.8125 , 0.1875 ],
       [0.5    , 0.5    ],
       [0.1875 , 0.8125 ],
       [0.9375 , 0.0625 ],
       [0.6875 , 0.3125 ],
       [0.3125 , 0.6875 ],
       [0.0625 , 0.9375 ],
       [0.875  , 0.125  ],
       [0.125  , 0.875  ],
       [0.75   , 0.25   ],
       [0.5    , 0.5    ],
       [0.25   , 0.75   ]])
```

[14]:
```
def simulate_game(M, p, max_iter=100):
    pos = [0]
    for i in range(max_iter):
        pos_next = np.random.choice(len(M), p=M[pos[-1]])
        if pos_next == pos[-1]:
            return pos + [pos_next]
        else:
            pos += [pos_next]
    return -1
```

[15]:
```
simulate_game(M, p=0.5)
```

[15]:
```
[0, 2, 5, 8, 14, 15, 14, 15, 14, 13, 12, 12]
```

[16]:
```
def monte_carlo(M, p, max_iter=100, reps=100, verbose=False):
    ties = 0
    winners = []
    final_transients = []
    game_lengths = []
    for i in range(reps):
        if verbose: print('Simulating game {0}'.format(i))
        game = simulate_game(M=M, p=p, max_iter=max_iter)
        tie = (type(game) in [int])
        if verbose: print('Game was {0} a tie'.format((1-tie)*'not'))
        ties += tie
        if not tie:
            winners += [game[-1]]
            final_transients += [game[-3]]
            if verbose: print('Winner was {0}, Second Last State was {1}'.
 ↪format(winners[-1], final_transients))
        game_lengths += [len(game)-2]
    winners = np.array(winners)
    final_transients = np.array(final_transients)
    return ties, winners, final_transients, game_lengths
```

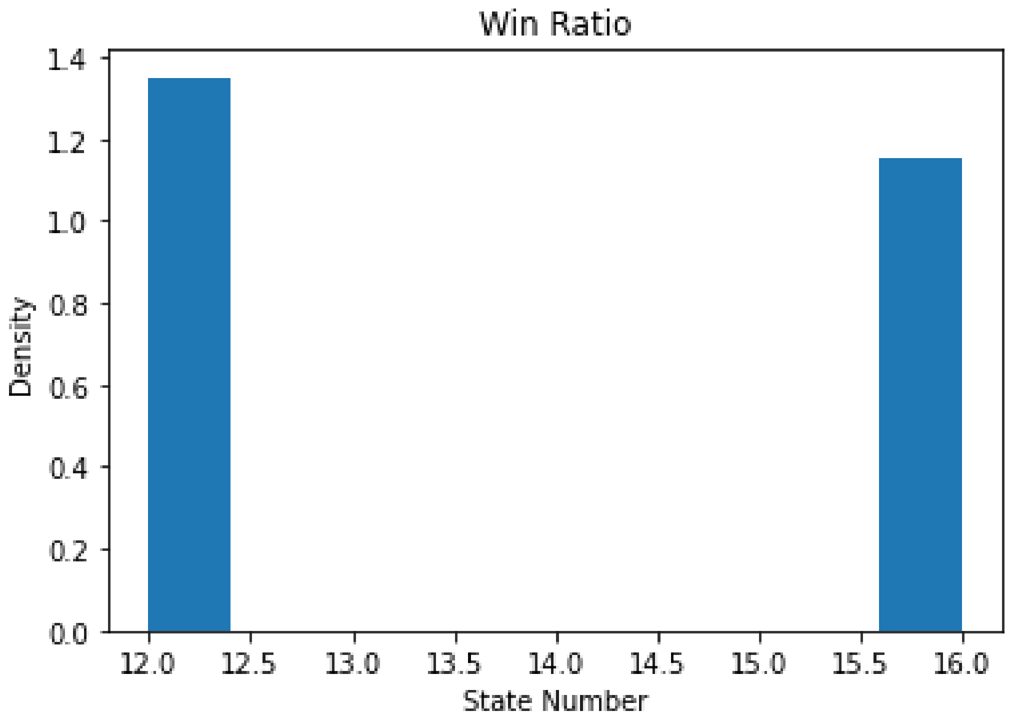
[17]:
```
ties, winners, final_transients, game_lengths = monte_carlo(M=M, p=p,␣
 ↪verbose=False)
```

[18]:
```
print('There were {0} ties'.format(ties))
```

```
There were 0 ties
```

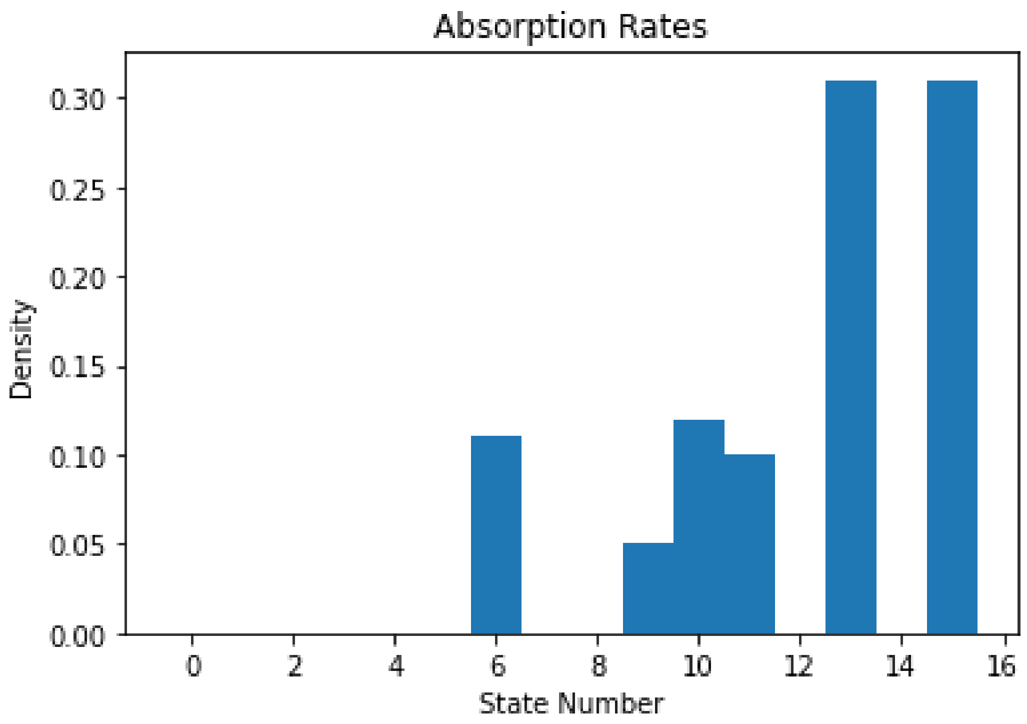
```
[19]: bins=np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13, 14, 15])-0.5
      bins=np.array(list(range(17)))-0.5

      fig = plt.hist(winners, density=True)
      plt.xlabel('State Number')
      plt.ylabel('Density')
      plt.title('Win Ratio')
```

```
[19]: Text(0.5, 1.0, 'Win Ratio')
```



```
[20]: fig = plt.hist(final_transients, bins=bins, density=True)
      plt.xlabel('State Number')
      plt.ylabel('Density')
      plt.title('Absorption Rates')
```

```
[20]: Text(0.5, 1.0, 'Absorption Rates')
```



[21]:
```
absorbtion_time_A = []
absorbtion_time_B = []

for i, game in enumerate(game_lengths):
    if winners[i] == 12:
        absorbtion_time_A += [game_lengths[i]]
    else:
        absorbtion_time_B += [game_lengths[i]]

absorbtion_time_A = np.array(absorbtion_time_A).mean()
absorbtion_time_B = np.array(absorbtion_time_B).mean()
absorbtion_time_A, absorbtion_time_B
```

[21]:
```
(6.074074074074074, 7.478260869565218)
```