CS420 Introduction to Artificial Intelligence

Assignment 2

Bharat Gangwani

Question 1

a

Yes, the heuristic is admissible.

Node n	Min cost to G	h(n)
S	16	12
A	13	8
B	11	7
C	9	9
D	11	6
E	8	5
F	5	5
H	3	1

b

No, it is not consistent. For a consistent estimator $h(n) \leq c(n, n') + h(n)$ for all nodes n and their successors n'. However, $h(F) > c(F, H) + h(H) \Leftrightarrow 5 > 2 + 1$.

c

BFS: Queue

Step #	Open list	DEQUEUE	Nodes to add
1	[S]	S	[A, B, C]
2	[A, B, C]	A	[D, E]
3	[B, C, D, E]	B	
4	[C, D, E]	C	[F, H]
5	[D, E, F, H]	D	
6	[E, F, H]	E	
7	[F, H]	F	
8	[H]	H	[G]

Path: [S, C, H, G], Cost: 18

A* Search: Priority Queue

Step #	Open list	DEQUEUE	Nodes to add
1	$[S^{12}]$	S^{12}	$[A_S^{12}, B_S^{12}, C_S^{18}]$
2	$[A_S^{12}, B_S^{12}, C_S^{18}]$	A_S^{12}	$[D_A^{13}, E_A^{14}]$
3	$[B_S^{12}, D_A^{13}, E_A^{14}, C_S^{18}]$	B_S^{12}	$[C_B^{16}]$
4	$[D_A^{13}, E_A^{14}, C_B^{16}]$	D_A^{13}	$[H_D^{16}]$
5	$[E_A^{14}, C_B^{16}, H_D^{16}]$	E_A^{14}	$[H_A^{15}]$
6	$[H_A^{15}, C_B^{16}]$	H_A^{15}	$[G_H^{17}]$
7	$[C_B^{16}, G_H^{17}]$	C_B^{16}	$[F_C^{16}]$
8	$[F_C^{16}, G_H^{17}]$	F_C^{16}	$[H_F^{14}]$
9	$[H_F^{14}, G_H^{17}]$	H_F^{14}	$[G_H^{16}]$
10	$[G_H^{16}]$	G_H^{16}	

Path: [S, B, C, F, H, G], Cost: 16

Question 2

a

A state is a a specific stacking of the N discs along the 4 towers in the appropriate order (smallest at the top). It can be stored in a (4 X N) array with each row acting as a stack as follows:

```
class Operator:
  t1 = -1
  t2 = -1

  def __init__(self, t1, t2):
      self.t1 = t1
      self.t2 = t2

class State:

  towers = -1
  towerpos = dict()
  previous = None
  gval = 0
  n = -1

  def __init__(self, n, prevtowers = None, prevtowerpos = None):
      self.towers = np.zeros([4, n])
      self.n = n
      self.towers[0] = [i for i in range(1, n+1)]
      self.towerpos = {0: 0, 1 : self.n, 2: self.n, 3: self.n}

      if prevtowers is not None:
          self.towers = np.zeros([4, n])
          for i in range(4):
              for j in range(n):
                  self.towers[i, j] = prevtowers[i][j]

      if prevtowerpos is not None:
          for key in prevtowerpos:
              self.towerpos[key] = prevtowerpos[key]

  def isEqual(self, s):
      for i in range(4):
```

```
            for j in range(self.n):
                if (self.towers[i][j] != s.towers[i][j]):
                    return 0
        return 1

    def applyOperator(self, o):
        s1 = State(self.n, self.towers, self.towerpos)
        s1.previous = self
        s1.gval = self.gval + 1

        if self.towerpos[o.t1] < self.n:
            k = self.towers[o.t1, self.towerpos[o.t1]]

            if self.towerpos[o.t2] == self.n:
                s1.towers[o.t2, self.towerpos[o.t2] - 1] = k
                s1.towers[o.t1, self.towerpos[o.t1]] = 0

                s1.towerpos[o.t1] += 1
                s1.towerpos[o.t2] -= 1

            elif (self.towers[o.t2, self.towerpos[o.t2]] > k):
                s1.towers[o.t2, self.towerpos[o.t2] - 1] = k
                s1.towers[o.t1, self.towerpos[o.t1]] = 0

                s1.towerpos[o.t1] += 1
                s1.towerpos[o.t2] -= 1

        return s1

    def printState(self):
        for i, char in enumerate(["A", "B", "C", "D"]):
            print(f"Tower {char}: {self.towers[i]}")
```

The (4 x N) array representation with each row as an ordered N-element stack allows me to check the top element of each row, pop it, compare it with the top element of other rows and insert it if it is smaller than the number to the right. Numbers to the left (nearer to index 0) have to be smaller than the numbers to their right in each row. This preserves the stacking rules required of the problem.

Example state (starting state):

$$\begin{bmatrix} 1 & 2 & 3 & ... & N \\ 0 & 0 & 0 & ... & 0 \\ 0 & 0 & 0 & ... & 0 \\ 0 & 0 & 0 & ... & 0 \end{bmatrix}$$

An action involves popping an element from the top of a stack and inserting it into another stack. Each action has a cost of 1 (tracked by "gval" in the code).

The successor state is the new 4 x N array with the top element from the popped stack inserted into the top of the new stack. In my implementation, it involves the first number from one row (nearest to index 0 but with nonzero elements to its right) being placed at the first place of another row (to the left of the first nonzero element in the row).

The objective in my implementation is to achieve this state:

$$\begin{bmatrix} 0 & 0 & 0 & ... & 0 \\ 0 & 0 & 0 & ... & 0 \\ 0 & 0 & 0 & ... & 0 \\ 1 & 2 & 3 & ... & N \end{bmatrix}$$

In other words, the state where all numbered elements are stacked appropriately on the last tower/row.

b

An admissible heuristic is counting the number of discs that are yet to be placed in the final tower. My implementation is below:

```
def h(state):
  zero_count = 0
  for i in state.towers[3]:
      if i == 0:
          zero_count += 1
  return zero_count
```

It is admissible as the minimum cost (number of moves) of placing k discs on the final tower is bound to be greater than or equal to k since we can only move one disc at a time.

c

```
STEP 1
----------------------------------------
OPEN LIST
Tower A: [1. 2. 3.]
Tower B: [0. 0. 0.]
Tower C: [0. 0. 0.]
Tower D: [0. 0. 0.]
--------------------
POP
Tower A: [1. 2. 3.]
Tower B: [0. 0. 0.]
Tower C: [0. 0. 0.]
Tower D: [0. 0. 0.]
--------------------
NODES ADDED
Tower A: [0. 2. 3.]
Tower B: [0. 0. 1.]
Tower C: [0. 0. 0.]
Tower D: [0. 0. 0.]
----------
Tower A: [0. 2. 3.]
Tower B: [0. 0. 0.]
Tower C: [0. 0. 1.]
Tower D: [0. 0. 0.]
----------
Tower A: [0. 2. 3.]
Tower B: [0. 0. 0.]
Tower C: [0. 0. 0.]
Tower D: [0. 0. 1.]
----------
----------------------------------------
STEP 2
----------------------------------------
OPEN LIST
Tower A: [0. 2. 3.]
```

```
Tower B: [0. 0. 0.]
Tower C: [0. 0. 0.]
Tower D: [0. 0. 1.]
--------------------
Tower A: [0. 2. 3.]
Tower B: [0. 0. 0.]
Tower C: [0. 0. 1.]
Tower D: [0. 0. 0.]
--------------------
Tower A: [0. 2. 3.]
Tower B: [0. 0. 1.]
Tower C: [0. 0. 0.]
Tower D: [0. 0. 0.]
--------------------
POP
Tower A: [0. 2. 3.]
Tower B: [0. 0. 0.]
Tower C: [0. 0. 0.]
Tower D: [0. 0. 1.]
--------------------
NODES ADDED
Tower A: [0. 0. 3.]
Tower B: [0. 0. 2.]
Tower C: [0. 0. 0.]
Tower D: [0. 0. 1.]
----------
Tower A: [0. 0. 3.]
Tower B: [0. 0. 0.]
Tower C: [0. 0. 2.]
Tower D: [0. 0. 1.]
----------
----------------------------------------
STEP 3
----------------------------------------
OPEN LIST
Tower A: [0. 0. 3.]
Tower B: [0. 0. 0.]
Tower C: [0. 0. 2.]
Tower D: [0. 0. 1.]
--------------------
Tower A: [0. 0. 3.]
Tower B: [0. 0. 2.]
Tower C: [0. 0. 0.]
```

```
Tower D: [0. 0. 1.]
--------------------
Tower A: [0. 2. 3.]
Tower B: [0. 0. 0.]
Tower C: [0. 0. 1.]
Tower D: [0. 0. 0.]
--------------------
Tower A: [0. 2. 3.]
Tower B: [0. 0. 1.]
Tower C: [0. 0. 0.]
Tower D: [0. 0. 0.]
--------------------
POP
Tower A: [0. 0. 3.]
Tower B: [0. 0. 0.]
Tower C: [0. 0. 2.]
Tower D: [0. 0. 1.]
--------------------
NODES ADDED
Tower A: [0. 0. 0.]
Tower B: [0. 0. 3.]
Tower C: [0. 0. 2.]
Tower D: [0. 0. 1.]
----------
Tower A: [0. 1. 3.]
Tower B: [0. 0. 0.]
Tower C: [0. 0. 2.]
Tower D: [0. 0. 0.]
----------
Tower A: [0. 0. 3.]
Tower B: [0. 0. 1.]
Tower C: [0. 0. 2.]
Tower D: [0. 0. 0.]
----------
Tower A: [0. 0. 3.]
Tower B: [0. 0. 0.]
Tower C: [0. 1. 2.]
Tower D: [0. 0. 0.]
```

Question 3

a

$$
\begin{aligned}
V((1,1))^{t+1} &= Q((1,1),N)^{t+1} \\
&= 0.96(-0.0025 + 0.95 * 1.688) + 0.02(-0.0025 + 0.95 * 1.598) \\
&+ 0.002(-0.0025 + 0.95 * 1.634) \\
&= 1.598 \\
V((1,2))^{t+1} &= Q((1,2),N)^{t+1} \\
&= 0.96(-0.0025 + 0.95 * 1.783) + 0.04 * (-0.0025 + 0.95 * 1.688) \\
&= 1.688 \\
V((1,3))^{t+1} &= Q((1,3),E)^{t+1} \\
&= 0.96(-0.0025 + 0.95 * 1.886) + 0.02(-0.0025 + 0.95 * 1.783)) \\
&+ 0.02(-0.0025 + 0.95 * 1.688) \\
&= 1.783 \\
V((2,1))^{t+1} &= Q((2,1),E)^{t+1} \\
&= 0.96(-0.0025 + 0.95 * 1.727) + 0.04(-0.0025 + 0.95 * 1.634) \\
&= 1.635 \\
V((2,3))^{t+1} &= Q((2,3),E)^{t+1} \\
&= 0.96(-0.0025 + 0.95 * 1.992) + 0.04(-0.0025 + 0.95 * 1.886) \\
&= 1.886 \\
V((3,1))^{t+1} &= Q((3,1),N)^{t+1} \\
&= 0.96(-0.0025 + 0.95 * 1.829) + 0.02 * (-0.0025 + 0.95 * 1.634) \\
&+ 0.02 * (-0.0025 + 0.95 * 1.582) \\
&= 1.727 \\
V((3,2))^{t+1} &= Q((3,2),N)^{t+1} \\
&= 0.96(-0.0025 + 0.95 * 1.992) + 0.02 * -1 + 0.02 * (-0.0025 + 0.95 * 1.829) \\
&= 1.83 \\
V((3,3))^{t+1} &= Q((3,3),E)^{t+1} \\
&= 0.96 * 2 + 0.02(-0.0025 + 0.95 * 1.886) + 0.002(-0.0025 + 0.95 * 1.829) \\
&= 1.99 \\
V((4,1))^{t+1} &= Q((4,1),W)^{t+1} \\
&= 0.96(-0.0025 + 0.95 * 1.727) + 0.02(-0.0025 + 0.95 * 1.582) + 0.02 * -1 \\
&= 1.583
\end{aligned}
$$

3	1.783	1.886	1.99	**Food**
2	1.688		1.83	**Tiger**
1	1.598	1.635	1.727	1.583
	1	**2**	**3**	**4**

b

3	E	E	E	**Food**
2	N		N	**Tiger**
1	N	E	N	W
	1	**2**	**3**	**4**

In each state, the agent should pick the action which maximises their expected utility. On the grid, that means prioritising movement towards a cell which has a higher maximum expected utility.

Question 4

a

```
import gym
import numpy as np
env = gym.make('FrozenLake-v1', desc=None, map_name="4x4",
    is_slippery=True)

s, info = env.reset()

samatrix = np.zeros([16, 4])
gamma = 0.90
alpha = 0.075
epsilon = 1

nep = 10000
sumr = 0
rrec = []
```

```
while nep > 0:

    if np.random.rand() < epsilon:
        options = np.where(samatrix[s] != max(samatrix[s]))[0]
        if len(options) > 0:
            act = np.random.randint(0, len(options))
            act = options[act]
            s1, r, terminated, truncated, info = env.step(act)
        else:
            act = np.random.randint(0, 4)
            s1, r, terminated, truncated, info = env.step(act)
    else:
        optoption = np.where(samatrix[s] == max(samatrix[s]))[0]
        if len(optoption) > 0:
            act = optoption[0]
            s1, r, terminated, truncated, info = env.step(act)
        else:
            act = np.random.randint(0, 4)
            s1, r, terminated, truncated, info = env.step(act)


    maxqs1 = max(samatrix[s1])
    qs = samatrix[s][act]
    samatrix[s][act] = qs + alpha*(r + gamma*maxqs1 - qs)
    sumr += r

    if terminated or truncated:
        s, info = env.reset()
        nep -= 1
        if nep % 100 == 0:
            epsilon *= 0.8
            epsilon = max(epsilon, 0.01)
        rrec.append(sumr)
        sumr = 0

    else:
        s = s1

import matplotlib.pyplot as plt

avgr = []
for i in range(0, 9900):
  avgr.append(sum(rrec[i:i+100])/100)
```

```
fig = plt.figure()
ax = fig.gca()
ax.plot(avgr)
ax.set_xlabel("N Episodes")
ax.set_ylabel("Average Reward")
plt.show()
```

b

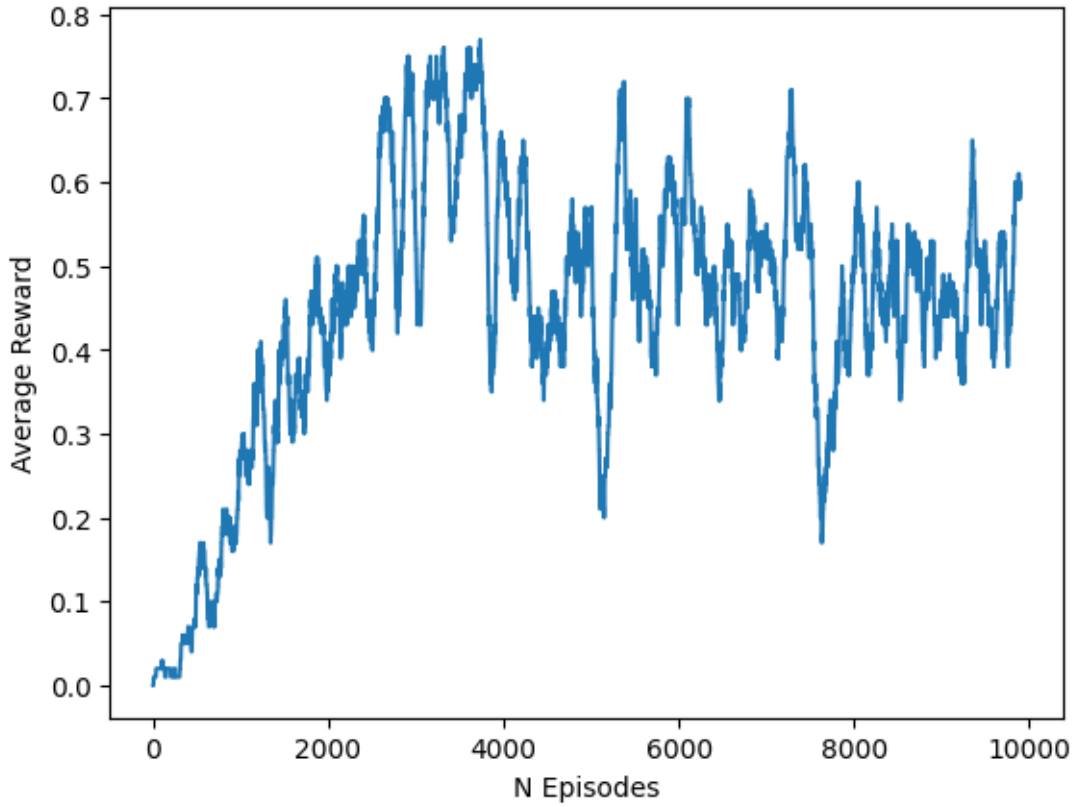

Figure 1: 100 Episode Moving Average Reward over 10,000 Episodes

c

$alpha = 0.075$; I realised that the average reward over episodes would plummet if I set the learning rate too high or too low. If it was too low, I believe it's because it was taking too long to converge. I selected the learning rate by adjusting it slightly to check how it affected the cumulative episode reward over the number of episodes.

Question 5

```
import nltk
nltk.download('stopwords')
from nltk.corpus import stopwords
nltk.download('punkt')

import numpy as np
import pandas as pd
from gensim.models import word2vec

import re # For regular expressions

# a
def load_data():
  """ Read tweets from the file.
      Return:
          list of lists (list_words), with words from each of the
              processed tweets
  """
  tweets = pd.read_csv('tweets.csv', names=['text'])
  list_words = []
  ### iterate over all tweets from the dataset
  for i in tweets.index:
    ### remove non-letter.
    text = re.sub("[^a-zA-Z ]", "", tweets.iloc[i].text)
    ### tokenize
    words = text.split()

    stop_words = set(stopwords.words("english"))

    new_words = []
```

```
    ### iterate over all words of a tweet
    for w in words:
      ## TODO: remove the stop words and convert a word (w) to
          the lower case
      if w.lower() not in stop_words:
        new_words.append(w.lower())

    list_words.append(new_words)
  return list_words

# check a few samples of twitter corpus
twitter_corpus = load_data()
print(twitter_corpus[:3])

# b
def distinct_words(corpus):
  """ get a list of distinct words for the corpus.
      Params:
          corpus (list of list of strings): corpus of documents
      Return:
          corpus_words (list of strings): list of distinct words
              across the corpus, sorted (using python 'sorted'
              function)
          num_corpus_words (integer): number of distinct words
              across the corpus
  """
  corpus_words = []
  num_corpus_words = -1
  # ------------------
  # TODO:
  for tweet in corpus:
      for word in tweet:
          if word not in corpus_words:
              corpus_words.append(word)

  num_corpus_words = len(corpus_words)
  corpus_words = sorted(corpus_words)

  # ------------------
  return corpus_words, num_corpus_words

words, num_words = distinct_words(twitter_corpus)
print(words[:10], num_words)
```

```
def compute_co_occurrence_matrix(corpus, window_size=5):
  """ Compute co-occurrence matrix for the given corpus and
      window_size (default of 5).
      Params:
          corpus (list of list of strings): corpus of documents
          window_size (int): size of context window
      Return:
          M (numpy matrix of shape = [number of corpus words x
              number of corpus words]):
              Co-occurence matrix of word counts.
              The ordering of the words in the rows/columns should
                  be the same as the ordering of the words given
                  by the distinct_words function.
          word2Ind (dict): dictionary that maps word to index
              (i.e. row/column number) for matrix M.
  """
  M = np.zeros((num_words, num_words))
  word2Ind = {word:i for i, word in enumerate(words)}

  # ------------------
  # TODO:
  for tweet in corpus:
      for idx, word in enumerate(tweet):
          snippet = tweet[idx-window_size:idx+window_size]
          for winword in snippet:
              if word == winword:
                  continue
              M[word2Ind[word]][word2Ind[winword]] += 1

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

  return M, word2Ind

M, word2Ind = compute_co_occurrence_matrix(twitter_corpus)
```

b

There are 10,841 unique words in the corpus.