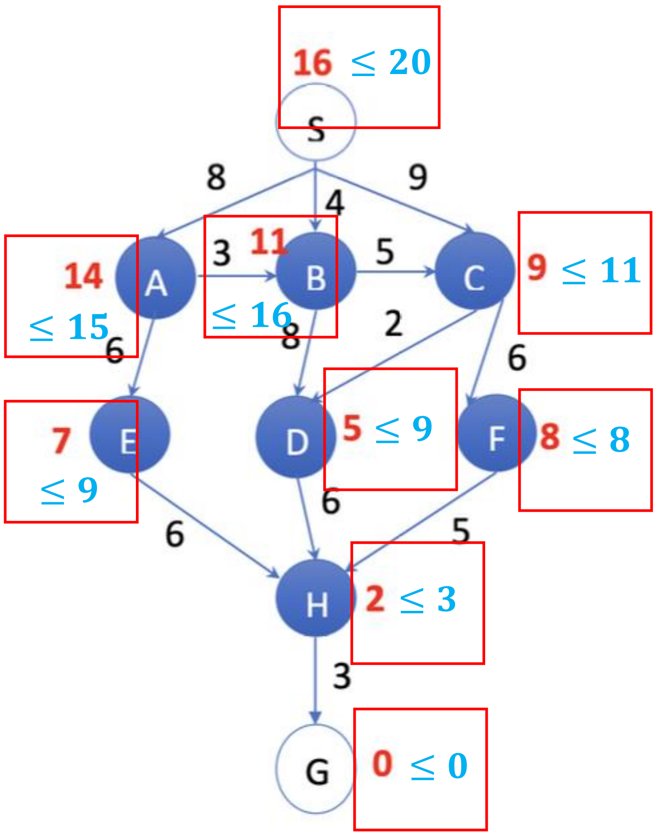
NAME: KANG CHIN SHEN (G1) MATRIC. NO: 01412921

CS420 ASSIGNMENT 2 SOLUTIONS

QUESTION 1



(a) Yes, the heuristic is admissible.

Justification:

- For a heuristic to be admissible, the following must be satisfied:

$$h(n) \leq TC^*(n, g)$$

- Since all the minimum cost to reach goal state from the current state (shown in light blue) is less than corresponding heuristic value (shown in red) of that state, the heuristic used is said to be admissible.

Node n	Minimum cost to reach goal from n, $TC^*(n, g)$	$h(n)$
S	20	16
A	15	14
B	16	11
C	11	9
E	9	7
D	9	5

F	8	8
H	3	2
G	0	0

(b) No, the heuristic is not consistent.

Justification:

- For a heuristic to be consistent, the following must be satisfied:

$$h(n) \leq c(n,p) + h(p)$$

- That is, the heuristic value to reach goal state, g from the current state, n is no greater than the step cost of getting to state p plus the heuristic cost of reaching g from p.

Node n	$c(n,p) + h(p)$	$h(n)$
S	- c(S,A) + h(A) = 8 + 14 = **22** - c(S,B) + h(B) = 4 + 11 = **15 (<16)** …	16

- Since the consistent heuristic must holds for every state and there is a violation, we conclude that the heuristic is not consistent.

(c) Part 1 : DFS Algorithm

- Final Solution Path : **S → A → E → H → G**
- Cost of Solution Path : **8 + 6 + 6 + 3 = 23**
- Open list's data structure : **Stack**

Step #	Open List (Stack)	Pop	Nodes to Add
1	S	S	C^S, B^S, A^S
2	A^S, B^S, C^S	A^S	E^A
3	E^A, B^S, C^S	E^A	H^E
4	H^E, B^S, C^S	H^E	G^H
5	G^H, B^S, C^S	G^H	Termination

Part 2 : A* Algorithm

- Final Solution Path : **S → C → D → H → G**
- Cost of Solution Path : **9 + 2 + 6 + 3 = 20**
- Open list's data structure : **Priority Queue**

Step #	Open List (Priority Queue)	Dequeue	Nodes to Add
1	S_{16}	S_{16}	$A^S_{22}, B^S_{15}, C^S_{18}$
2	$B^S_{15}, C^S_{18}, A^S_{22}$	B^S_{15}	D^B_{17}, C^B_{18} ** Do not replace C in open list*
3	$D^B_{17}, C^S_{18}, A^S_{22}$	D^B_{17}	H^D_{20}
4	$C^S_{18}, H^D_{20}, A^S_{22}$	C^S_{18}	D^C_{16}, F^C_{23} ** Re-open D in close list*
5	$D^C_{16}, H^D_{20}, A^S_{22}, F^C_{23}$	D^C_{16}	H^D_{19} ** Replace H in open list*
6	$H^D_{19}, A^S_{22}, F^C_{23}$	H^D_{19}	G^H_{20}
7	$G^H_{20}, A^S_{22}, F^C_{23}$	G^H_{20}	*Terminated*

- The priority of each state, n is based on the f-cost, f(n) which is defined as:

$$f(n) = g(n) + h(n)$$

That is, ***sum*** of ***g(n), cost from start state, s_0 to current state, n,*** and ***h(n), heuristic from current state, n to goal state, g.***

- Note: Since the heuristic is not consistent, re-opening of closed nodes is possible.

QUESTION 2

Clarification of term(s):

- 'First' refers to the integer at the front(leftmost element) of a list.

Part 1

I. **State representation**

Describe specifically what is a state in this problem:

- Given N discs, a state could be any legal combination (a bigger disc should never be put on top of a smaller disc) of discs on top of each of the three towers.

How you would store it in a computer using a data structure, and justify the correctness of your state representation:

- One of the possible data structures for this state representation would be using three lists, storing the information of which tower is having which discs (of specific size) being stacked appropriately on it. Since we have N discs of different sizes, each can be assigned with an integer value from 1,2…,N corresponding to smallest to biggest disc. This allows us to represent each tower as an ordered list of integers. For example, the start state, s_0 would be ([1,2…,N], [], []).

II. **Actions**

- Here, an action X_y, can be defined as **popping** the first integer(i.e., disc) from a list, X (i.e., tower) and **pushing** it to the front of another list, Y.
- An exhausted list of all possible actions for this search problem would be:

Operators, A = {A_B, A_C, B_A, B_C, C_A, C_B}

- For an action to be valid at a given state (not all actions are valid at a given each state):
  - The integer value being pushed to the front of another list must be strictly smaller than the integer currently located at the front of the stack be pushed to.
  - Suppose a list has no integer currently, then any of first integer from other lists is allowed to be popped and pushed to it.

III. **Cost of different actions**

- Cost is 1 for all actions, $X_y \epsilon$ A (uniform cost).

IV. **Successor State (for each action)**

- Suppose that an action, X_y is valid at a given state, s based on the constraints defined in (2) then the successor state, s' is defined as:
  - Let first integer of list X be k.
  - Then s' would be the state s, with k being popped from the front of list X, and pushed to the front of list Y.

V. **Objective**

- Find the least cost path (a **sequence** of **operator** applied on **each intermediate states**) from the start state, s_0 = ([1,2…,N],[],[]) to the goal state, s_g = ([],[],[1,2…,N]).

Part 2

- For a heuristic to be admissible, the following must be satisfied:

$$h(n) \leq TC^*(n, g)$$

- Now, let h(n) = 2 * (number of discs on tower C that is either smaller than discs on A or B) + (Total number of discs on tower A and B)
- Justification of admissibility:
  - For any state, we have p discs on A, q discs on B, and r discs on C.
  - We also know the goal state is to have all discs being stacked appropriately (increasing size) on C.
  - Before moving any discs from A or B to C, we must first remove those discs with smaller size than those on A and B first (let this be m discs), this takes m actions.
  - Next, we perform relaxation on the original problem:
    (1) Now we are allowed to remove any disc from any of the towers instead of only the top disc (but ONLY allowed to push to the top of the target tower) at any state.
    (2) We have one extra tower, T and this tower is allowed hold any discs without considering the size.
  - With this relaxation, now we can remove the m discs on C to T, then always select the next largest discs from any of the A, B and T and place them in correct order on top C, this takes (m + p + q) actions.
  - Hence, total cost of the relaxed problem = m + (m + p + q) = 2 * m + p + q
  - By the definition of original problem, this heuristic is trivially admissible as the total cost is lower bounded by the heuristic function.

Part 3

- The priority of each state, n is based on the f-cost, f(n) which is defined as:

$$f(n) = g(n) + h(n)$$

  That is, ***sum*** of ***g(n), cost from start state, s_0 to current state, n,*** and ***h(n), heuristic from current state, n to goal state, g.***
- Let the start state, $S0_3$ be ([1,2,3],[],[]) and f(S0) = 0 + (2*0 + 3) = 3
- Let the next valid successor states be:

- $S1_4^{S0}$ (after applying operator A_B on $S0$) : ([2,3],[1],[]) & f-cost = 1 + (2*0 + 3) = 4
- $S2_5^{S0}$ (after applying operator A_C on $S0$): ([2,3],[],[1]) & f-cost = 1 + (2*1 + 2) = 5

- $S3_6^{S1}$ (after applying operator A_C on $S1$): ([3],[1],[2]) & f-cost = 2 + (2*1 + 2) = 6
- $S0_5^{S1}$ (after applying operator B_A on $S1$): ([1,2,3],[],[]) & f-cost = 2 + (2*0 + 3) = 5
- $S2_6^{S1}$ (after applying operator B_C on $S1$): ([2,3],[],[1]) & f-cost = 2 + (2*1 + 2) = 6

- $S4_6^{S2}$ (after applying operator A_B on $S2$): ([3],[2],[1]) & f-cost = 2 + (2*1 + 2) = 6
- $S0_5^{S2}$ (after applying operator C_A on $S2$): ([1,2,3],[],[]) & f-cost = 2 + (2*0 + 3) = 5
- $S1_5^{S0}$ (after applying operator C_B on $S2$): ([2,3],[1],[]) & f-cost = 2 + (2*0 + 3) = 5

...

First three steps of A*star search in table representation:

Step #	Open List (Priority Queue)	Dequeue	Nodes to Add
1	$S0_3$	$S0_3$	$S1_4^{S0}, S2_5^{S0}$
2	$S1_4^{S0}, S2_5^{S0}$	$S1_4^{S0}$	$S3_6^{S1}, S0_5^{S1}, S2_6^{S1}$ ** Do not re-open S0 in the close list* ** Do not replace S2 in the open list*
3	$S2_5^{S0}, S3_6^{S1}$	$S2_5^{S0}$	$S4_6^{S2}, S0_5^{S2}, S1_5^{S0}$ ** Do not re-open S0 in the close list* ** Do not re-open S1 in the close list*
4	$S4_6^{S2}, S3_6^{S1}$	$S4_6^{S2}$	...

QUESTION 3

Part (a)

The values for different states V^t at iteration t are given by:

3	1.564	1.710	1.842	**Food**
2	1.451	blocked	1.270	**Tiger**
1	1.327	1.231	1.162	0.844
	1	**2**	**3**	**4**

Computation of values of different states V^{t+1} using Bellman Optimal Equation:

$$(1)\ Q^{t+1}(s,a) = \sum_{s'} P(s'|s,a)\,[R(s,a,s') + \gamma V^t(s')]$$

$$(2)\ V^{t+1}(s) = \max_a Q^{t+1}(s,a)$$

The available actions at each state are = {N,S,E,W}, each representing
N: North, S: South, E: East, W: West

For state (1,1):

$Q^{t+1}((1,1), N)$
= 0.7 * (-0.0025 + 0.95 * 1.451)
+ 0.15 * (-0.0025 + 0.95 * 1.327)
+ 0.15 * (-0.0025 + 0.95 * 1.231)
= 1.3269

$Q^{t+1}((1,1), S)$
= 0.7 * (-0.0025 + 0.95 * 1.327)
+ 0.15 * (-0.0025 + 0.95 * 1.327)
+ 0.15 * (-0.0025 + 0.95 * 1.231)
= 1.2445

$Q^{t+1}((1,1), W)$
= 0.7 * (-0.0025 + 0.95 * 1.327)
+ 0.15 * (-0.0025 + 0.95 * 1.451)
+ 0.15 * (-0.0025 + 0.95 * 1.327)
= 1.2758

$Q^{t+1}\big((1,1), E\big)$
= 0.7 * (-0.0025 + 0.95 * 1.231)
+ 0.15 * (-0.0025 + 0.95 * 1.451)
+ 0.15 * (-0.0025 + 0.95 * 1.327)
= 1.2120

$V^{t+1}\big((1,1)\big) = 1.3269$

For state (1,2):

$Q^{t+1}\big((1,2), N\big)$
= 0.7 * (-0.0025 + 0.95 * 1.231)
+ 0.15 * (-0.0025 + 0.95 * 1.327)
+ 0.15 * (-0.0025 + 0.95 * 1.162)
= 1.1708

$Q^{t+1}((1,2), S)$
= 0.7 * (-0.0025 + 0.95 * 1.231)
+ 0.15 * (-0.0025 + 0.95 * 1.327)
+ 0.15 * (-0.0025 + 0.95 * 1.162)
= 1.1708

$Q^{t+1}((1,2), W)$
= 0.7 * (-0.0025 + 0.95 * 1.327)
+ 0.15 * (-0.0025 + 0.95 * 1.231)
+ 0.15 * (-0.0025 + 0.95 * 1.231)
= 1.2308

$Q^{t+1}((1,2), E)$
= 0.7 * (-0.0025 + 0.95 * 1.162)
+ 0.15 * (-0.0025 + 0.95 * 1.231)
+ 0.15 * (-0.0025 + 0.95 * 1.231)
= 1.1211

$V^{t+1}((1,2)) = 1.2308$

For state (1,3):
$Q^{t+1}\left((1,3), N\right)$
= 0.7 * (-0.0025 + 0.95 * 1.270)
+ 0.15 * (-0.0025 + 0.95 * 1.231)
+ 0.15 * (-0.0025 + 0.95 * 0.844)
= 1.1377

$Q^{t+1}((1,3), S)$
= 0.7 * (-0.0025 + 0.95 * 1.162)
+ 0.15 * (-0.0025 + 0.95 * 1.231)
+ 0.15 * (-0.0025 + 0.95 * 0.844)
= 1.0659

$Q^{t+1}((1,3), W)$
= 0.7 * (-0.0025 + 0.95 * 1.231)
+ 0.15 * (-0.0025 + 0.95 * 1.270)
+ 0.15 * (-0.0025 + 0.95 * 1.162)
= 1.1627

$Q^{t+1}((1,3), E)$
= 0.7 * (-0.0025 + 0.95 * 0.844)
+ 0.15 * (-0.0025 + 0.95 * 1.270)
+ 0.15 * (-0.0025 + 0.95 * 1.162)
= 0.9053

$V^{t+1}((1,3)) = 1.1627$

For state (1,4):
$Q^{t+1}\left((1,4), N\right)$
= 0.7 * (-1 + 0)
+ 0.15 * (-0.0025 + 0.95 * 1.162)
+ 0.15 * (-0.0025 + 0.95 * 0.844)
= –0.4149

$Q^{t+1}((1,4), S)$
= 0.7 * (-0.0025 + 0.95 * 0.844)
+ 0.15 * (-0.0025 + 0.95 * 1.162)
+ 0.15 * (-0.0025 + 0.95 * 0.844)
= 0.8446

$Q^{t+1}((1,4), W)$
= 0.7 * (-0.0025 + 0.95 * 1.162)
+ 0.15 * (-1 + 0)
+ 0.15 * (-0.0025 + 0.95 * 0.844)
= 0.7409

$Q^{t+1}((1,4), E)$
= 0.7 * (-0.0025 + 0.95 * 0.844)
+ 0.15 * (-1 + 0)
+ 0.15 * (-0.0025 + 0.95 * 0.844)
= 0.5294

$V^{t+1}((1,4)) = 0.8466$

For state (2,1):
$Q^{t+1}\big((2,1), N\big)$
= 0.7 * (-0.0025 + 0.95 * 1.564)
+ 0.15 * (-0.0025 + 0.95 * 1.451)
+ 0.15 * (-0.0025 + 0.95 * 1.451)
= 1.4511

$Q^{t+1}((2,1), S)$
= 0.7 * (-0.0025 + 0.95 * 1.327)
+ 0.15 * (-0.0025 + 0.95 * 1.451)
+ 0.15 * (-0.0025 + 0.95 * 1.451)
= 1.2935

$Q^{t+1}((2,1), W)$
= 0.7 * (-0.0025 + 0.95 * 1.451)
+ 0.15 * (-0.0025 + 0.95 * 1.564)
+ 0.15 * (-0.0025 + 0.95 * 1.327)
= 1.3744

$Q^{t+1}((2,1), E)$
= 0.7 * (-0.0025 + 0.95 * 1.451)
+ 0.15 * (-0.0025 + 0.95 * 1.564)
+ 0.15 * (-0.0025 + 0.95 * 1.327)
= 1.3744

$V^{t+1}((2,1)) = 1.4511$

For state (2,3):
$Q^{t+1}\big((2,3), N\big)$
= 0.7 * (-0.0025 + 0.95 * 1.842)
+ 0.15 * (-0.0025 + 0.95 * 1.270)
+ 0.15 * (-1 + 0)
= 1.2538

$Q^{t+1}((2,3), S)$
= 0.7 * (-0.0025 + 0.95 * 1.162)
+ 0.15 * (-0.0025 + 0.95 * 1.270)
+ 0.15 * (-1 + 0)
= 0.8016

$Q^{t+1}((2,3), W)$
= 0.7 * (-0.0025 + 0.95 * 1.270)
+ 0.15 * (-0.0025 + 0.95 * 1.842)
+ 0.15 * (-0.0025 + 0.95 * 1.162)
= 1.2701

$Q^{t+1}((2,3), E)$
= 0.7 * (-1 + 0)
+ 0.15 * (-0.0025 + 0.95 * 1.842)
+ 0.15 * (-0.0025 + 0.95 * 1.162)
= −0.2727

$V^{t+1}((2,3)) = 1.2701$

For state (3,1):
$Q^{t+1}\big((3,1), N\big)$
= 0.7 * (-0.0025 + 0.95 * 1.564)
+ 0.15 * (-0.0025 + 0.95 * 1.564)
+ 0.15 * (-0.0025 + 0.95 * 1.710)
= 1.5041

$Q^{t+1}((3,1), S)$
= 0.7 * (-0.0025 + 0.95 * 1.451)
+ 0.15 * (-0.0025 + 0.95 * 1.564)
+ 0.15 * (-0.0025 + 0.95 * 1.710)
= 1.4290

$Q^{t+1}((3,1), W)$
= 0.7 * (-0.0025 + 0.95 * 1.564)
+ 0.15 * (-0.0025 + 0.95 * 1.564)
+ 0.15 * (-0.0025 + 0.95 * 1.451)
= 1.4672

$Q^{t+1}((3,1), E)$
= 0.7 * (-0.0025 + 0.95 * 1.710)
+ 0.15 * (-0.0025 + 0.95 * 1.564)
+ 0.15 * (-0.0025 + 0.95 * 1.451)
= 1.5643

$V^{t+1}((3,1)) = 1.5643$

For state (3,2):
$Q^{t+1}\big((3,2), N\big)$
= 0.7 * (-0.0025 + 0.95 * 1.710)
+ 0.15 * (-0.0025 + 0.95 * 1.564)
+ 0.15 * (-0.0025 + 0.95 * 1.842)
= 1.6200

$Q^{t+1}((3,2), S)$
$= 0.7 * (-0.0025 + 0.95 * 1.710)$
$+ 0.15 * (-0.0025 + 0.95 * 1.564)$
$+ 0.15 * (-0.0025 + 0.95 * 1.842)$
$= 1.6200$

$Q^{t+1}((3,2), W)$
$= 0.7 * (-0.0025 + 0.95 * 1.564)$
$+ 0.15 * (-0.0025 + 0.95 * 1.710)$
$+ 0.15 * (-0.0025 + 0.95 * 1.710)$
$= 1.5249$

$Q^{t+1}((3,2), E)$
$= 0.7 * (-0.0025 + 0.95 * 1.842)$
$+ 0.15 * (-0.0025 + 0.95 * 1.710)$
$+ 0.15 * (-0.0025 + 0.95 * 1.710)$
$= 1.7098$

$V^{t+1}((3,2)) = 1.7098$

For state (3,3):
$Q^{t+1}((3,3), N)$
$= 0.7 * (-0.0025 + 0.95 * 1.842)$
$+ 0.15 * (-0.0025 + 0.95 * 1.710)$
$+ 0.15 * (2 + 0)$
$= 1.7665$

$Q^{t+1}((3,3), S)$
$= 0.7 * (-0.0025 + 0.95 * 1.270)$
$+ 0.15 * (-0.0025 + 0.95 * 1.710)$
$+ 0.15 * (2 + 0)$
$= 1.3861$

$Q^{t+1}((3,3), W)$
$= 0.7 * (-0.0025 + 0.95 * 1.710)$
$+ 0.15 * (-0.0025 + 0.95 * 1.842)$
$+ 0.15 * (-0.0025 + 0.95 * 1.270)$
$= 1.5781$

$Q^{t+1}((3,3), E)$
$= 0.7 * (2 + 0)$
$+ 0.15 * (-0.0025 + 0.95 * 1.842)$
$+ 0.15 * (-0.0025 + 0.95 * 1.270)$
$= 1.8427$

$V^{t+1}((3,3)) = 1.8427$

Hence, the numerical answer for each state of V^{t+1} is summarized as below:

3	1.5643	1.7098	1.8427	**Food**
2	1.4511	blocked	1.2701	**Tiger**
1	1.3269	1.2308	1.1627	0.8446
	1	**2**	**3**	**4**

Part (b)
Optimal policy is chosen by :
$\pi^{t+1}(s) = \arg\max_{a\in\{N,S,W,S\}} Q^{t+1}(s,a)$, which gives the best action (of that state) that maximize the expected utility over all states.

For state (1,1):
$\pi^{t+1}((1,1)) = \arg\max_{a\in\{N,S,W,S\}} Q^{t+1}((1,1),a)$
According to computation of $Q^{t+1}\big((1,1),a\big), \forall a \in \{N,S,W,S\}$ in part (a), the action to be chosen for (1,1) should be N (North), corresponding to $V^{t+1}\big((1,1)\big) = 1.3269$.

For state (1,2):
$\pi^{t+1}((1,2)) = \arg\max_{a\in\{N,S,W,S\}} Q^{t+1}((1,2),a)$
According to computation of $Q^{t+1}\big((1,2),a\big), \forall a \in \{N,S,W,S\}$ in part (a), the action to be chosen for (1,2) should be W(West), corresponding to $V^{t+1}\big((1,2)\big) = 1.\,2308$.

For state (1,3):
$\pi^{t+1}((1,3)) = \arg\max_{a\in\{N,S,W,S\}} Q^{t+1}((1,3),a)$
According to computation of $Q^{t+1}\big((1,3),a\big), \forall a \in \{N,S,W,S\}$ in part (a), the action to be chosen for (1,3) should be W (West), corresponding to $V^{t+1}\big((1,3)\big) = 1.\,1627$.

For state (1,4):
$\pi^{t+1}((1,4)) = \arg\max_{a\in\{N,S,W,S\}} Q^{t+1}((1,4),a)$
According to computation of $Q^{t+1}\big((1,4),a\big), \forall a \in \{N,S,W,S\}$ in part (a), the action to be chosen for (1,4) should be S (South), corresponding to $V^{t+1}\big((1,4)\big) = 0.8466$.

For state (2,1):
$\pi^{t+1}((2,1)) = \arg\max_{a\in\{N,S,W,S\}} Q^{t+1}((2,1),a)$
According to computation of $Q^{t+1}\big((2,1),a\big), \forall a \in \{N,S,W,S\}$ in part (a), the action to be chosen for (2,1) should be N (North), corresponding to $V^{t+1}\big((2,1)\big) = 1.4511$.

For state (2,3):
$\pi^{t+1}((2,3)) = \arg\max_{a\in\{N,S,W,S\}} Q^{t+1}((2,3),a)$
According to computation of $Q^{t+1}\big((2,3),a\big), \forall a \in \{N,S,W,S\}$ in part (a), the action to be chosen for (2,3) should be W (West), corresponding to $V^{t+1}\big((2,3)\big) = 1.\,2701$.

For state (3,1):
$\pi^{t+1}((3,1)) = \arg\max_{a\in\{N,S,W,S\}} Q^{t+1}((3,1),a)$
According to computation of $Q^{t+1}\big((3,1),a\big), \forall a \in \{N,S,W,S\}$ in part (a), the action to be chosen for (3,1) should be E (East), corresponding to $V^{t+1}\big((3,1)\big) = 1.\,5643$.

For state (3,2):
$\pi^{t+1}((3,2)) = \arg\max_{a\in\{N,S,W,S\}} Q^{t+1}((3,2),a)$
According to computation of $Q^{t+1}\big((3,2),a\big), \forall a \in \{N,S,W,S\}$ in part (a), the action to be chosen for (3,2) should be E(East), corresponding to $V^{t+1}\big((3,2)\big) = 1.\,7098$.

For state (3,3):
(9) $\pi^{t+1}((3,3)) = \arg\max_{a\in\{N,S,W,S\}} Q^{t+1}((3,3),a)$

According to computation of $Q^{t+1}\big((3,3),a\big), \forall a \in \{N,S,W,S\}$ in part (a), the action to be chosen for (3,3) should be E(East), corresponding to $V^{t+1}\big((3,3)\big) = 1.8427$.

Hence, the final policy is summarized as follows:

3	East	East	East	**Food**
2	North	blocked	West	**Tiger**
1	North	West	West	South
	1	**2**	**3**	**4**

QUESTION 4

Part (a)

```
[8] # Import packages
    import numpy as np
    import gym
    import random
    import matplotlib.pyplot as plt
    import seaborn as sn
    %matplotlib inline
```

```
[10] # Create the Frozen Lake Environment
     env = gym.make("FrozenLake8x8-v0")
     env.render()

     SFFFFFFF
     FFFFFFFF
     FFFHFFFF
     FFFFFHFF
     FFFHFFFF
     FHHFFFHF
     FHFFHFHF
     FFFHFFFG
```

```
[11] # Determine state space
     action_size = env.action_space.n
     print("Action size: ", action_size)

     # Determine action space
     state_size = env.observation_space.n
     print("State size: ", state_size)

     # Create Q-table
     qtable = np.zeros((state_size, action_size))
     qtable_history = []
     score_history = []
     print("Table shape: ", qtable.shape)

     Action size:  4
     State size:  64
     Table shape:  (64, 4)
```

- Shows the size of the state space, action space
- Shows the appropriate data structure – Q-Table

```
[12] # Hyperparameters

     gamma = 0.9                  # Discount factor
     total_episodes = 250000      # Total episodes
     learning_rate = 0.8          # Learning rate
     max_steps = 400              # Max steps per episode = action_size * state_size * 2

     # Exploration parameters
     epsilon = 1.0                # Exploration rate
     max_epsilon = 1.0            # Exploration probability at start
     min_epsilon = 0.001          # Minimum exploration probability
     decay_rate = 0.00005         # Exponential decay rate for exploration prob
```

- Set the hyperparameters for Q-learning, and discount factor to 0.9

```
[13] # List of rewards
     rewards = []

     # Run the Q-Learning with pre-defined number of episodes
     for episode in range(total_episodes):
     # for episode in range(10000):

         # Reset the environment in a new episode
         state = env.reset()

         step = 0

         done = False

         total_rewards = 0

         # In a episode (Start to Finish)
         for step in range(max_steps):

             # Choose an action, a in the current world state
             # Get a randomized number
             exp_exp_tradeoff = random.uniform(0, 1)

             # If this number > greater than epsilon : exploitation (taking the biggest Q value for this state)
             if exp_exp_tradeoff > epsilon:
                 action = np.argmax(qtable[state,:])

             # Else doing a random choice : exploration
             else:
                 action = env.action_space.sample()

             # Take the action (a) and observe the outcome state(s') and reward (r)
             new_state, reward, done, info = env.step(action)

             # Update Q(s,a):= Q(s,a) + lr [R(s,a) + gamma * max Q(s',a') - Q(s,a)]
             qtable[state, action] = qtable[state, action] + learning_rate * (reward + gamma * np.max(qtable[new_state, :]) - qtable[state, action])

             # Add the reward to total_rewards of this episode
             total_rewards += reward

             # Our new state is state
             state = new_state

             # If done (we're dead) : finish episode
             if done == True:
                 break

         # Reduce epsilon after each iteration : exploitation over exploration
         epsilon = min_epsilon + (max_epsilon - min_epsilon)*np.exp(-decay_rate*episode)


         # Append the reward of this eposide to reward list
         rewards.append(total_rewards)

         # Increase the episode_count
         episode_count = episode + 1
```

- Code that implements Q-learning
- (**For Part b**) Keep track of total accumulated reward for each episode (episode return), assigned to a list, *rewards*

Part (b)

```
[56] # Show the total accumulated reward for each episode
     print(rewards)

     [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
```

- Above shows the total accumulated reward for each episode

```
# Check the output
print ("Reward over time: " +  str(sum(rewards)/total_episodes))
print(qtable)

Reward over time: 0.39144
[[1.32525222e-04 4.31942840e-03 1.52473354e-04 7.97957168e-05]
 [2.54019009e-04 2.76173527e-04 8.13561853e-03 2.72442824e-04]
 [2.50588281e-03 1.07765109e-02 3.68550900e-04 3.57161705e-04]
 [5.16094251e-04 3.86486095e-04 8.13010243e-03 4.38916780e-04]
 [3.57892979e-03 3.42679710e-03 1.13285416e-02 9.35404889e-04]
 [1.65969646e-03 2.61644607e-02 1.69018776e-03 1.65799474e-03]
 [2.19857469e-03 7.71242660e-03 2.02568016e-02 1.98186868e-03]
 [2.33347791e-03 2.27945155e-03 3.46931761e-02 2.32059681e-03]
 [7.15363818e-05 1.04280500e-03 6.02439213e-04 4.31249930e-03]
 [7.37808995e-04 1.97554772e-04 1.23526902e-03 6.16685351e-03]
 [4.25812570e-04 3.18293807e-04 1.27517574e-04 7.27002862e-03]
 [2.09309828e-05 2.40066609e-04 6.07451796e-06 2.41039591e-02]
 [7.18272159e-04 9.96163072e-04 7.61527154e-04 3.34240640e-02]
 [1.03244440e-03 6.77987005e-02 9.36417455e-04 9.26199500e-04]
 [2.29843912e-03 2.03073617e-03 4.04647474e-02 2.41583682e-03]
 [5.32120985e-02 2.41269698e-03 2.42619515e-03 2.41074152e-03]
 [3.54617861e-03 5.99360674e-05 5.58661702e-05 6.22782165e-05]
 [6.24484729e-05 5.99889227e-05 3.01379426e-03 6.32381072e-05]
 [8.50978621e-03 1.20445978e-05 2.11742736e-05 3.28044602e-05]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [5.23439746e-05 7.36388902e-07 2.12221622e-02 1.99714464e-05]
 [2.65990421e-06 9.54520021e-04 3.15561012e-04 3.33340875e-02]
 [2.48227368e-03 4.12791193e-03 2.36338960e-02 3.96264380e-03]
 [4.29093917e-03 1.11398914e-01 3.85206780e-03 4.08329360e-03]
 [3.42464398e-05 2.82535004e-05 1.47946691e-03 2.72407402e-05]
 [1.57186418e-05 1.56584697e-05 4.09285915e-05 9.29312120e-04]
 [1.64224229e-03 1.65814361e-06 5.65632875e-06 1.16044500e-05]
 [1.18104281e-13 1.14976793e-04 1.21396743e-12 7.18917359e-10]
 [5.83319143e-03 8.92278102e-07 3.87931550e-05 3.50434992e-07]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [3.74682348e-03 3.44878922e-03 2.25232501e-02 9.89821617e-04]
 [1.67624044e-02 1.87795652e-02 2.53619093e-01 6.67343425e-03]
 [8.05965576e-06 7.03766197e-06 4.22088735e-04 7.21340405e-06]
 [7.11913913e-08 6.65118312e-07 7.56575848e-07 9.28903043e-04]
 [1.43458307e-10 3.00578877e-12 1.53945541e-11 7.22753834e-05]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [2.91065749e-05 1.57979777e-05 7.74933376e-03 3.30221826e-04]
[2.91065749e-05 1.57979777e-05 7.74933376e-03 3.30221826e-04]
[1.41450741e-04 1.52899897e-02 6.71881741e-09 5.05595640e-05]
[4.62267563e-04 7.70281260e-05 9.60538953e-04 1.87097799e-02]
[1.62271290e-02 3.86382286e-02 3.83346448e-01 4.24571598e-02]
[1.27297332e-04 5.68704303e-09 1.11768299e-07 1.07709174e-07]
[0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
[0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
[8.53009309e-11 6.73272634e-11 1.37975400e-04 1.30283909e-10]
[1.45400384e-06 7.73825706e-06 1.57460716e-08 3.49094649e-03]
[3.13979336e-02 3.78351383e-05 1.00443811e-05 1.22401937e-06]
[0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
[8.02945115e-03 2.52081948e-02 8.00296409e-01 4.12773168e-04]
[1.41051346e-05 9.90211209e-08 1.69954709e-07 2.42558092e-07]
[0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
[1.95370768e-11 1.05863426e-04 7.97998443e-10 1.81226199e-11]
[5.18318452e-04 2.81835229e-11 1.86779784e-12 1.89859086e-11]
[0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
[9.02736641e-07 1.70615867e-08 1.99329085e-01 1.80838524e-08]
[0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
[1.05851556e-01 3.35999976e-02 9.99358283e-01 9.84906847e-02]
[8.73090959e-06 3.20746948e-07 2.67769648e-07 3.00135794e-07]
[4.17568791e-08 7.44329442e-06 1.44254852e-09 1.19647408e-07]
[3.92702467e-05 1.29317822e-09 2.18544943e-10 7.41115685e-10]
[0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
[0.00000000e+00 4.51566400e-06 3.07676481e-06 0.00000000e+00]
[7.05127244e-03 9.22108642e-03 7.14477613e-01 1.75486714e-02]
[1.33110377e-01 9.71083813e-01 7.94735696e-03 1.30649327e-01]
[0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]]
```

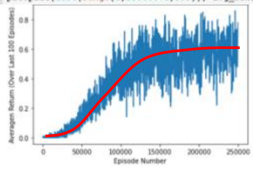
- Each of the values above is the Average Reward (Sum of Rewards of All Episodes / Total Number of Episodes)

```
count = 0
reward_sum = 0
avg_rewards = []

for val in rewards:
    reward_sum += val
    if count % 100 == 0:
        avg_rewards.append(reward_sum/100)
        reward_sum = 0
    count += 1
```

```
font = {'size'   : 10}
plt.xticks(fontsize=10)
plt.yticks(fontsize=10)

plt.xlabel('Episode Number', **font)
plt.ylabel('Averagen Return (Over Last 100 Episodes)', **font)
plt.plot(list(range(1,250000+1,100)), avg_rewards)
```



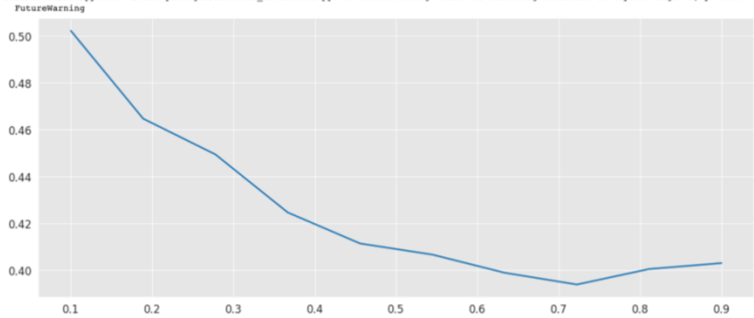
- Plot the graph – average return (over the last 100 episodes while the agent is learning) on y-axis and episode number on x-axis
- The number of episodes is sufficient to ensure convergence as shown above

Part (c)

Graph of Average Reward VS Learning Rate

```
# Set learning rate to values in [0.1, 0.9]
learning_rates = np.linspace(0.1, 0.9, 10)
test_rewards = []
for   list: test_rewards
      (10 items) [0.502172, 0.464696, 0.44942, 0.424676, 0.41144,
    print(str(learning_rate) + ": " + str(sum(rewards)/total_episodes))
# Plot the
plot_results(learning_rates, test_rewards)
```

```
0.1: 0.502172
0.18888888888888888: 0.464696
0.2777777777777778: 0.44942
0.3666666666666667: 0.424676
0.4555555555555556: 0.41144
0.5444444444444445: 0.406696
0.6333333333333333: 0.398948
0.7222222222222222: 0.393928
0.8111111111111111: 0.400544
0.9: 0.403064
/usr/local/lib/python3.7/dist-packages/seaborn/_decorators.py:43: FutureWarning: Pass the following variables as keyword args: x, y. From version 0.12, the only valid positional argument will be `data
  FutureWarning
```



- Plot the graph – Average Reward (Sum of Rewards of All Episodes / Total Number of Episodes) on y-axis and Learning Rate on x-axis
- As shown above, the Average Reward (as a metric to measure performance) drops when the Learning Rate increases
- According to wikipedia, '*A too high learning rate will make the learning jump over minima but a too low learning rate will either take too long to converge or get stuck in an undesirable local minimum*'
- In conclusion, after testing learning rates with 10 different values in [0.1,0.9], I believe 0.45556 is a decent learning rate to use, after considering both factors mentioned above
- As the average reward between using learning rate of 0.81111 and 0.45556 differ only approximately 0.01, and due to limitation of computation power of machine, I used **0.8** for this Q-Learning

QUESTION 5

Part (a)

```python
import nltk

# import the stop words
nltk.download('stopwords')
from nltk.corpus import stopwords

# import the reuters dataset
nltk.download('reuters')
from nltk.corpus import reuters

file_loc = '/root/nltk_data/corpora/reuters.zip'
from zipfile import ZipFile
with ZipFile(file_loc, 'r') as z:
    z.extractall('/root/nltk_data/corpora/')

import numpy as np
from gensim.models import word2vec
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package reuters to /root/nltk_data...
[nltk_data]   Package reuters is already up-to-date!
```

```python
def load_data(category="money-fx"):

    """ Read files from the specified Reuter's category.
        Params:
            category (string): category name
        Return:
            list of lists, with words from each of the processed files/documents
    """

    files = reuters.fileids(category)

    list_words = []

    ### iterate over all documents/files from the reuters dataset
    for f in files:
        words = []

        ### iterate over all words of a document/file
        for w in list(reuters.words(f)):

            ## TODO: remove the stop words, convert a word (w) to the lowercase, and append to the list

            # (1) Convert w to lowercase and assign it to word
            word = w.lower()

            # (2) If the word is not a stopword, append it to list_words
            if word not in stopwords.words('english'):
                words.append(word)

        list_words.append(words)

    return list_words

# check a few samples of reuters corpus
reuters_corpus = load_data()
print(reuters_corpus[:3])
```

```
[['bundesbank', 'allocates', '6', '.', '1', 'billion', 'marks', 'tender', 'bundesbank', 'accepted', 'bids', '6', '.', '1', 'billion', 'marks', 'today', "'", 'tender', '28', '-', 'day', 'securities', '
```

- Process words into the lowercase
- Perform stop words removal

Part (b)

```python
from nltk.corpus.reader.wordnet import lin_similarity
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

    # (1) Create a list of distinct words in the corpus, sorted using 'sorted' function
    for document in corpus:
        for word in document:
            corpus_words.append(word)

    corpus_words = list(dict.fromkeys(corpus_words))
    corpus_words = sorted(corpus_words)

    # (2) Find the number of distinct words across the corpus
    num_corpus_words = len(corpus_words)

    return corpus_words, num_corpus_words

words, num_words = distinct_words(reuters_corpus)
print('words    (unique words): ', words)
print('num_words(unique words): ', num_words)
```

```
words    (unique words):  ['"', '"(', '",', '".', '&', "'", '\',"', '(', ')', '),', '),"', ').', ').\"', ',', ',"', ',\'"', '-', '--', '---', '----', '.', '."', ".'", '.\'"', '.(', '.)', '.,', '.-', '.
num_words(unique words):  7157
```

- Number of unique words (size of vocabulary) : 7157

```
[ ] def compute_co_occurrence_matrix(corpus, window_size=7):

        """ Compute co-occurrence matrix for the given corpus and window_size (default of 7).
            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.
        """

        # (1) Create numpy matrix of shape = [number of corpus words x number of corpus words]
        M = np.array([[0 for i in range(num_words)] for i in range (num_words)])

        # (2) word2Ind (dict): dictionary that maps word to index (i.e. row/column number) for matrix M.
        word2Ind = {}
        idx = 0
        for str in words:
            word2Ind[str] = idx
            idx += 1

        # (3) Compute the co-occurence matrix of word counts
        #    Ordering of the words in the rows/columns should be the same as the ordering of the words given by the distinct_words function
        for document in corpus:
            for i in range(len(document)):
                for left_pointer in range(i-1, i-1-window_size, -1):
                    if left_pointer < 0:
                        break
                    M[word2Ind[document[i]]][word2Ind[document[left_pointer]]] += 1


                for right_pointer in range(i+1, i+1+window_size, 1):
                    if right_pointer > (len(document) - 1):
                        break
                    M[word2Ind[document[i]]][word2Ind[document[right_pointer]]] += 1


        return M, word2Ind

    M, word2Ind = compute_co_occurrence_matrix(reuters_corpus)
```

```
[ ] #Check the dimension of Co-occurence Matrix
    print(len(M[0]))
    print(len(M))
    print(M)
```

```
7157
7157
[[240   0   1 ...   0   0   1]
 [  0   0   0 ...   0   0   0]
 [  1   0   0 ...   0   0   0]
 ...
 [  0   0   0 ...   0   0   0]
 [  0   0   0 ...   0   0   0]
 [  1   0   0 ...   0   0   0]]
```

- Set the window size to 7 on either size of the word

Part (c)

```
[ ] # -----------------------------
    # Run SVD
    # Note: This may take several minutes
    # ------------------------------

    # Perform SVD using
    la = np.linalg

    # Extract the first 75 components with highest variance
    U, s, Vh = la.svd(M, full_matrices=False)

    # Create an array of word embeddings (size of 75)
    word_embedding_75dim = U[:,:75]
```

```
[ ] # Check dimension of word_embedding_75dim
    print(len(word_embedding_75dim))
    print(len(word_embedding_75dim[0]))
```

```
7157
75
```

```
[ ] # Check the entries in word_embedding_75dim
    print(word_embedding_75dim[0])
```

```
[-0.09452258 -0.07720011 -0.12735456  0.05468409 -0.07876138 -0.04031832
 -0.22225215  0.09948283 -0.08399988  0.19380147 -0.31778021 -0.03067155
  0.02684725  0.13678115 -0.02323169 -0.03050785 -0.01286921 -0.06073952
  0.01030706 -0.07832958  0.04890193 -0.04293866 -0.03067227 -0.16071788
  0.11480833  0.11068655  0.16673438 -0.00143427 -0.39310589 -0.05267965
 -0.06656377 -0.02629117  0.10618979 -0.0766517   0.01791283 -0.25378902
 -0.02198051  0.01126623 -0.15405202 -0.1794979   0.11116424  0.28674949
  0.15648581 -0.14059331 -0.04498939  0.03706441 -0.10200549  0.09349053
 -0.00913624  0.03514359  0.00233887  0.03652442 -0.04197862 -0.10038659
  0.03754252  0.03747057 -0.05015903 -0.06717924  0.04048411  0.0154382
  0.02732327  0.04784751 -0.03090008 -0.06238556 -0.02026827  0.01941879
  0.04099045 -0.02403856  0.00823303  0.03884228  0.07178907 -0.03300502
  0.03370903  0.03295775 -0.02111137]
```

- Apply SVD and obtain word embedding of size 75

Part (d)

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

# Creating the model and setting values for the various parameters
num_features = 75   # Set word vector dimensionality to 75 for consistency with previous implementation
min_word_count = 0  # Set minimum word count to 0 for consistency with previous implementation
num_workers = 4     # Number of parallel threads
context = 7         # Set ontext window size to 7 for consistency with previous implementation
downsampling = 1e-3 # (0.001) Downsample setting for frequent words

# Initializing the train model
from gensim.models import word2vec
print("Training model....")
model = word2vec.Word2Vec(reuters_corpus,\
                          workers=num_workers,\
                          size=num_features,\
                          min_count=min_word_count,\
                          window=context,
                          sample=downsampling)

# To make the model memory efficient
model.init_sims(replace=True)

# Saving the model for later use ( Word2Vec.load() )
model_name = "75_features_0minwords_7context"
model_path = F"/content/drive/MyDrive/colab_notebooks/asgn2/{model_name}"
model.save(model_path)
print("model saved")
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
Training model....
model saved
```

- Generate word embedding of size 75 using Word2Vec.pynb on the same Reuters dataset

```
from sklearn.metrics.pairwise import cosine_similarity

def svd_most_similar(query_word, n=10):
    """ return 'n' most similar words of a query word using the SVD word embeddings similar to word2vec's most_smilar
        Params:
            query_word (strings): a query word
        Return:
            most_similar (list of strings): the list of 'n' most similar words
    """

    cosine_sim = cosine_similarity(word_embedding_75dim, np.array([word_embedding_75dim[word2Ind[query_word]]]))
    cosine_sim = cosine_sim.flatten()

    indices = (-cosine_sim).argsort()[1:n+1]

    most_similar = []

    for i in indices:
        most_similar.append((words[i], cosine_sim[i]))

    for val in most_similar:
        print(val)
```

- To compare and determine which method (Word2Vec OR SVD) works better, a few examples are shown below:

```
[95] model = word2vec.Word2Vec.load(model_path)
     model.wv.most_similar('money')

     [('assistance', 0.9967169761657715),
      ('operated', 0.9960709810256958),
      ('revised', 0.9957497715950012),
      ('forecast', 0.9947110414505005),
      ('provided', 0.9942721128463745),
      ('700', 0.9939642548561096),
      ('350', 0.993422269821167),
      ('receives', 0.9923965930938721),
      ('compares', 0.9910964369773865),
      ('market', 0.9906169772148132)]

[98] svd_most_similar('money')

     ('gets', 0.4576669125840092)
     ('442', 0.4521483187068887)
     ('given', 0.4492420270849863)
     ('210', 0.4318722984079865)
     ('205', 0.4255437853268562 7)
     ('receives', 0.4222603958836185)
     ('186', 0.3913454693981322 5)
     ('injects', 0.3900390422575899)
     ('570', 0.3863047941647073)
     ('166', 0.3796084071326851)

[101] model.wv.most_similar('bank')

     [('help', 0.9957151412963867),
      ('given', 0.9939113259315491),
      ('market', 0.9924650192260742),
      ('affecting', 0.9912904500961304),
      ('afternoon', 0.9896146655082703),
      ('700', 0.9865050315856934),
      ('invited', 0.9858958721160889),
      ('takes', 0.9856358766555786),
      ('191', 0.9854189157485962),
      ('750', 0.9851484298706055)]

[102] svd_most_similar('bank')

     ('stereotype', 0.3016644191084184)
     ('tide', 0.2533390082709267 6)
     ('jokes', 0.2401325332775271 8)
     ('sportsman', 0.2376865089794874 3)
     ('beer', 0.2338818589957177)
     ('glass', 0.2321270869130946 7)
     ('scored', 0.2319843415752417 3)
     ('sufficed', 0.2278668671375087 3)
     ('meguerditch', 0.22402487622726294)
     ('likes', 0.2237786692341388 3)
```

- Based on the observation above, on overall, Word2Vec approach works better for this reuters_corpus:
    - For the word ‘money’, 5 out of 10 of the returned results of SVD approach are numerical values, which is relatively less meaningful for this case as compared to Word2Vec approach.

- For the word ‘bank’, we can see that one of the top 3 returned results of Word2Vec approach is ‘market’, which by meaning, could be related. However, for SVD approach, almost all returned results have no direct meaning with the word ‘bank’.

- One possible reason for this observation is that the dataset provided is not huge enough to capture the actual ‘meaning’ of each word
- Furthermore, for this specific example, we are required to set the hyperparameter values based on Q5 instructions, e.g., the window size is set to 7, which could be adjusted to larger size to better capture the relationship between words.

```
[ ] # Test : Consider removing these non-english words

    non_english_words = ['"', '"(', '",', '".', '&', "'", '\',"', '(', ')', '),', '),"', ').', ').', ',', ',"', ',\'"', '-', '--', '---', '----', '.', '."', ".'", '.\'"', '.(', '.)', '.,', '.-', '..', '
    print(len(non_english_words))

    645

[ ] # Test
    print(645/7157)

    0.09012155931256113
```

- Another possible reason is the corpus contains relative high number of non-English words. Although these special characters may be meaningful to certain extend, but with word counts approaching 10% of the entire corpus may affects, and possibly reduce the accuracy of SVD computation approach.

- Finish -