OPTIMIZING WAREHOUSE FLOWS

A Business Optimizing Case Study using Deep Q Learning

MAY 20 2021

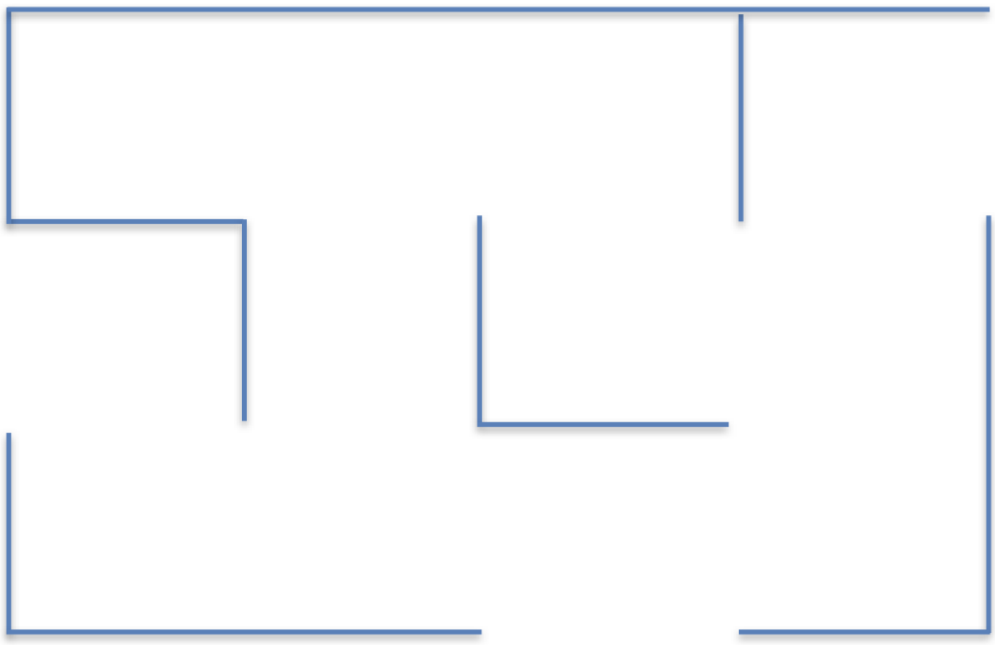
Authored by: Anuj Malkotia

OPTIMIZING PROCESSES

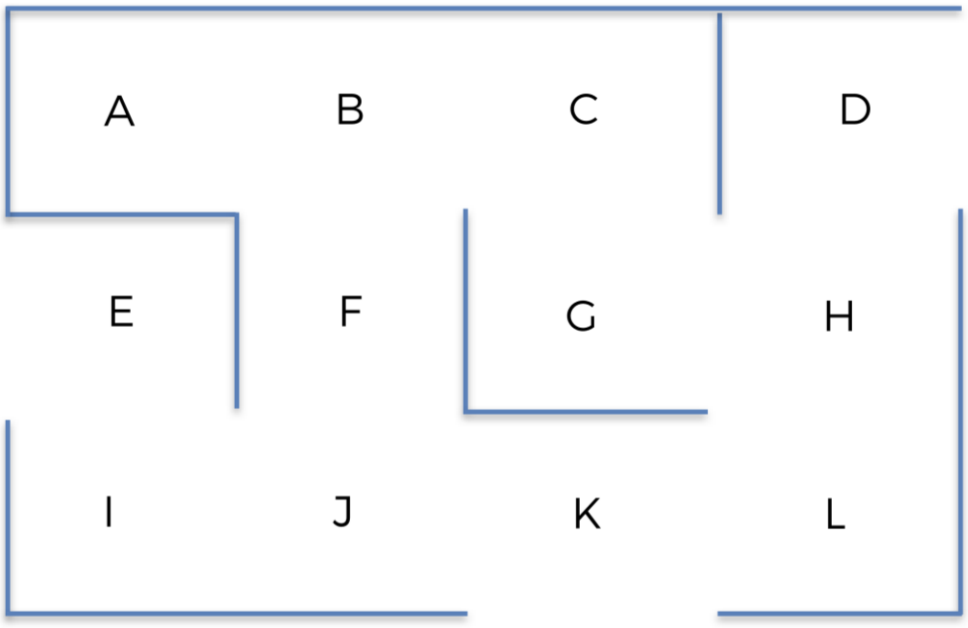
A machine has the possibility to identify problems the human eye would have no chance at. A camera can be far more sensitive than the iris of an eyeball. That's not to say that humans will be completely eliminated from the flaw-finding processes of development - judgments and corrections to design flaws still lies within the scope of only-human expertise. But overall, AI and computer vision will help humans through this process, allowing for much more in-depth and fastidious checking and testing processes, allowing for a much greater scale of design optimization.

Case Study: Optimizing the Flows in an E-Commerce Warehouse

The problem to solve will be to optimize the flows inside the following warehouse:



Above is the map of a warehouse, the warehouse belongs to an online retail company that sells products to a variety of customers. Inside this
Warehouse, the products are stored in 12 different locations, labeled by the following letters from A to L:



As the orders are placed by the customers online, an Autonomous Warehouse Robot is moving around the Warehouse to collect the products for future deliveries. Here is what it looks like:

The 12 locations are all connected to a computer system, which is ranking in real time the priorities of Product collection for these 12 locations. For example, at a specific time t, it will return the following Ranking:

Priority Rank	Location
1	G
2	K
3	L
4	J
5	A
6	I
7	H
8	C
9	B
10	D
11	F
12	E

PROBLEM STATEMENT

Location G has priority 1, which means it is the top priority, as it contains a product that must be collected and delivered immediately. Our Autonomous Warehouse Robot must move to location G by the shortest route depending on where it is. Our goal is to build an AI that will return that shortest route, wherever the robot is. But then as we see, locations K and L are in the Top 3 priorities. Hence we will want to implement an option for our Autonomous Warehouse Robot to go by some intermediary locations before reaching its final top priority location.

The way the system computes the priorities of the locations is out of the scope of this case study. The reason for this is that there can be many ways, from simple rules or algorithms, to deterministic computations, to machine learning. But most of these ways would not be artificial intelligence as we know it today. What we really want to focus on is core AI, encompassing Q-Learning, Deep Q-Learning and other branches of Reinforcement Learning. So we will just say for example that Location G is top priority because one of the most loyal platinum customers of the company placed an urgent order of a product stored in location G which therefore must be delivered as soon as possible. Therefore in conclusion, our mission is to build an AI that will always take the shortest route to the top

Priority location, whatever the location it starts from, and having the option to go by an intermediary location which is in the top 3 priorities.

ENVIRONMENT TO DEFINE

When building an AI, the first thing we always have to do is to define the environment. And defining an
Environment always requires the three following elements:

- Defining the states
- Defining the actions
- Defining the rewards

1. DEFINING THE STATES

Let's start with the states. The input state is simply the location where our Autonomous Warehouse Robot is at each time t. However since we will build our AI with mathematical equations, we will encode the Locations names (A, B, C,...) into index numbers, with respect to the following mapping:

Location	State
A	0
B	1
C	2
D	3
E	4
F	5
G	6
H	7
I	8
J	9
K	10
L	11

There is a specific reason why we encode the states with indexes from 0 to 11, instead of other integers. The reason is that we will work with matrices, a matrix of reward and a matrix of Q-Values, and each line/column of these matrices will correspond to a specific location. For example, the first line of each matrix, which has index 0, corresponds to location A. The second line/column, which has index 1, corresponds to

location B. Etc. We will see the purpose of working with matrices in more details a bit later.

```
location_to_state = {'A':0,'B':1,'C':2,'D':3,'E':4,'F':5,'G':6,'H':7,'I':8,'J':9,'K':10,'L':11}
```

2. DEFINING THE ACTIONS

Now let's define the possible actions to play. The actions are simply the next moves the robot can make to go from one location to the next. So for example, let's say the robot is in location J, the possible actions that the robot can play is to go to I, to F, or to K. And again, since we will work with mathematical equations, we will encode these actions with the same indexes as for the states. Hence, following our same example where the robot is in location J at a specific time, the possible actions that the robot can play are, according to our previous mapping above: 5, 8 and 10. Indeed, the index 5 corresponds to F, the index 8 corresponds to I, and the index 10 corresponds to K. Therefore eventually, the total list of actions that the AI can play overall is the following:

```
actions = [0,1,2,3,4,5,6,7,8,9,10,11]
```

Obviously, when being in a specific location, there are some actions that the robot cannot play. Taking the same previous example, if the robot is in location J, it can play the actions 5, 8 and 10, but it cannot play the other actions. We will make sure to specify that by attributing a 0 reward to the actions it cannot play, and a 1 reward to the actions it can play. And that brings us to the rewards.

```
actions = [0,1,2,3,4,5,6,7,8,9,10,11]
```

3. DEFINING THE REWARDS

The last thing that we have to do now to build our environment is to define a system of rewards. More specifically, we have to define a reward function R that takes as inputs a state s and an action a, and returns a numerical reward that the AI will get by playing the action a in the state s:

$$R : (\text{state}, \text{action}) \mapsto r \in \mathbb{R}$$

So how are we going to build such a function for our case study? Here this is simple. Since there is a discrete and finite number of states (the indexes from 0 to 11), as well as a discrete and finite number of actions (same indexes from 0 to 11), the best way to build our reward function R is to simply make a matrix. Our reward function will exactly be a matrix of 12 rows and 12 columns, where the rows correspond to the states, and the columns correspond to the actions. That way, in our function "R : (s; a) 7! r 2 R", s will be the row index of the matrix, a will be the column index of the matrix, and r will be the cell of indexes (s; a) in the matrix. Therefore the only thing that we have to do now to define our reward function is simply to populate this matrix with the numerical rewards. And as we just said in the previous paragraph, what we have to do first is to attribute, for each of the 12 locations, a 0 reward to the actions that the robot cannot play, and a 1 reward to the actions the robot can play. By doing that for each of the 12 locations, we will end up with a matrix of rewards. Let's build it step by step, starting with the first location: location A. When being in location A, the robot can only go to location B. Therefore, since location A has index 0 (first row of the matrix) and location B has index 1 (second column of the matrix), the first row of the matrix of rewards will get a 1 on the second column, and a 0 on all the other columns, just like so: **V**

Defining the Rewards:

	A	B	C	D	E	F	G	H	I	J	K	L
A	0	1	0	0	0	0	0	0	0	0	0	0
B												
C												
D												
E												
F												
G												
H												
I												
J												
K												
L												

Now let's move on to location B. When being in location B, the robot can only go to three different locations: A, C and F. Since B has index 1 (second row), and A, C, F have respective indexes 0, 2, 5 (1st, 3rd, and 6th column), then the second row of the matrix of rewards will get a 1 on the 1st, 3rd and 6th columns, and 0 on all the other columns. Hence we get:

Defining the Rewards:

	A	B	C	D	E	F	G	H	I	J	K	L
A	0	1	0	0	0	0	0	0	0	0	0	0
B	1	0	1	0	0	1	0	0	0	0	0	0
C												
D												
E												
F												
G												
H												
I												
J												
K												
L												

Then same, C (of index 2) is only connected to B and G (of indexes 1 and 6) By doing the same for all the other locations, we eventually get our final matrix of rewards:

Defining the Rewards:

	A	B	C	D	E	F	G	H	I	J	K	L
A	0	1	0	0	0	0	0	0	0	0	0	0
B	1	0	1	0	0	1	0	0	0	0	0	0
C	0	1	0	0	0	0	1	0	0	0	0	0
D	0	0	0	0	0	0	0	1	0	0	0	0
E	0	0	0	0	0	0	0	0	1	0	0	0
F	0	1	0	0	0	0	0	0	0	1	0	0
G	0	0	1	0	0	0	0	1	0	0	0	0
H	0	0	0	1	0	0	1	0	0	0	0	1
I	0	0	0	0	1	0	0	0	0	1	0	0
J	0	0	0	0	0	1	0	0	1	0	1	0
K	0	0	0	0	0	0	0	0	0	1	0	1
L	0	0	0	0	0	0	0	1	0	0	1	0

```
rewards = np.array([[0,1,0,0,0,0,0,0,0,0,0,0],
                    [1,0,1,0,0,1,0,0,0,0,0,0],
                    [0,1,0,0,0,0,1,0,0,0,0,0],
                    [0,0,0,0,0,0,0,1,0,0,0,0],
                    [0,0,0,0,0,0,0,0,1,0,0,0],
                    [0,1,0,0,0,0,0,0,0,1,0,0],
                    [0,0,1,0,0,0,1,1,0,0,0,0],
                    [0,0,0,1,0,0,1,0,0,0,0,1],
                    [0,0,0,0,1,0,0,0,0,1,0,0],
                    [0,0,0,0,0,1,0,0,1,0,1,0],
                    [0,0,0,0,0,0,0,0,0,1,0,1],
                    [0,0,0,0,0,0,0,1,0,0,1,0]])
```

AI SOLUTION

The Whole Q learning Algorithm:

Let's summarize the different steps of the whole Q-Learning process:

Initialization:

For all couples of states s and actions a, the Q-Values are initialized to 0:

$$\forall s \in S, a \in A, Q_0(s,a) = 0$$

We start in the initial state s0. We play a random possible action and we reach the first state s1.

Then for each t >=1, we will repeat for a certain number of times (1000 times in our code) the following:

1. We select a random state st from our 12 possible states:

st = random(0,1, 2, 3, 4, 5, 6, 7,8, 9, 10, 11)

2. We play a random action at that can lead to a next possible state, i.e., such that R(st; at) > 0:

at = random(0; 1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11) s.t. R(st; at) > 0

3. We reach the next state st+1 and we get the reward R(st; at)

4. We compute the Temporal Difference TDt(st , at):

$$TD_t(s_t, a_t) = R(s_t, a_t) + \gamma \max_a(Q(s_{t+1}, a)) - Q(s_t, a_t)$$

5. We update the Q-value by applying the Bellman equation:

$$Q_t(s_t, a_t) = Q_{t-1}(s_t, a_t) + \alpha TD_t(s_t, a_t)$$

CODE FOR ABOVE AI SOLUTION:

```
def route(starting_location,ending_location):
    rewards_new = np.copy(rewards)
    ending_state = location_to_state[ending_location]
    rewards_new[ending_state, ending_state] = 1000

    #Initializing the Q values
    Q = np.array(np.zeros([12,12]))

    #Implementing Q Learning Process
    for i in range(1000):
        current_state = np.random.randint(0,12)
        playable_actions = []
        for j in range(12):
            if rewards_new[current_state,j]>0:
                playable_actions.append(j)

        next_state = np.random.choice(playable_actions)
        TD = rewards_new[current_state,next_state] + gamma * Q[next_state,np.argmax(Q[next_state,])] - Q[current_state,next_state]
        Q[current_state,next_state] = Q[current_state,next_state] + alpha * TD

    route = [starting_location]
    next_location = starting_location
    while(next_location != ending_location):
        starting_state = location_to_state[starting_location]
        next_state = np.argmax(Q[starting_state,])
        next_location = state_to_locations[next_state]
        route.append(next_location)
        starting_location = next_location
    print( route )

#PART 3 GOING INTO PRODUCTION
#Printing the final route
print('Route:')
route('E','G')
```

WHOLE CODE:

```
#Importing Libraries
import numpy as np

#Setting parameters for alpha(learning factor) and gamma(discount factor)
gamma = 0.75
alpha = 0.9  #Low value = Slow Learning / High Value = Fast Learning

#PART 1 - DEFINING THE ENVIRONMENT

#Defining states
location_to_state = {'A':0,'B':1,'C':2,'D':3,'E':4,'F':5,'G':6,'H':7,'I':8,'J':9,'K':10,'l':11}


#defining actions
actions = [0,1,2,3,4,5,6,7,8,9,10,11]

#Defining rewards
rewards = np.array([[0,1,0,0,0,0,0,0,0,0,0,0],
                    [1,0,1,0,0,1,0,0,0,0,0,0],
                    [0,1,0,0,0,0,1,0,0,0,0,0],
                    [0,0,0,0,0,0,0,1,0,0,0,0],
                    [0,0,0,0,0,0,0,0,1,0,0,0],
                    [0,1,0,0,0,0,0,0,0,1,0,0],
                    [0,0,1,0,0,0,1,1,0,0,0,0],
                    [0,0,0,1,0,0,1,0,0,0,0,1],
                    [0,0,0,0,1,0,0,0,0,1,0,0],
                    [0,0,0,0,0,1,0,0,1,0,1,0],
                    [0,0,0,0,0,0,0,0,0,1,0,1],
                    [0,0,0,0,0,0,0,1,0,0,1,0]])


#PART 2 - BUILDING THE AI SOLUTION IE Q LEARNING

#Making a mapping from the states to the locations
```

```
state_to_locations = {state:location for location, state in location_to_state.items()}

#Making the final function that will return the optimal route

def route(starting_location,ending_location):
    rewards_new = np.copy(rewards)
    ending_state = location_to_state[ending_location]
    rewards_new[ending_state, ending_state] = 1000

    #Initializing the Q values
    Q = np.array(np.zeros([12,12]))

    #Implementing Q Learning Process
    for i in range(1000):
        current_state = np.random.randint(0,12)
        playable_actions = []
        for j in range(12):
            if rewards_new[current_state,j]>0:
                playable_actions.append(j)

        next_state = np.random.choice(playable_actions)
        TD = rewards_new[current_state,next_state] + gamma *
Q[next_state,np.argmax(Q[next_state,])] - Q[current_state,next_state]
        Q[current_state,next_state] = Q[current_state,next_state] + alpha * TD

    route = [starting_location]
    next_location = starting_location
    while(next_location != ending_location):
        starting_state = location_to_state[starting_location]
        next_state = np.argmax(Q[starting_state,])
        next_location = state_to_locations[next_state]
        route.append(next_location)
        starting_location = next_location
    print( route )

#PART 3 GOING INTO PRODUCTION
```

```
#Printing the final route
print('Route:')
route('E','G')
```