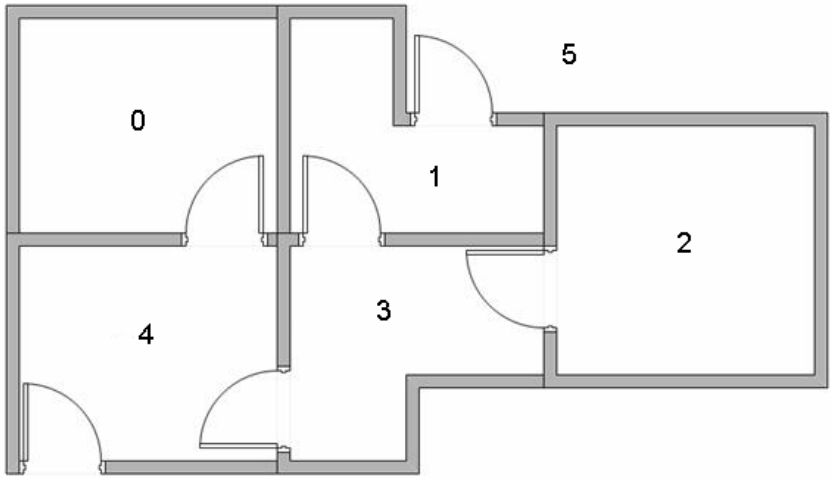
Data mining for networks

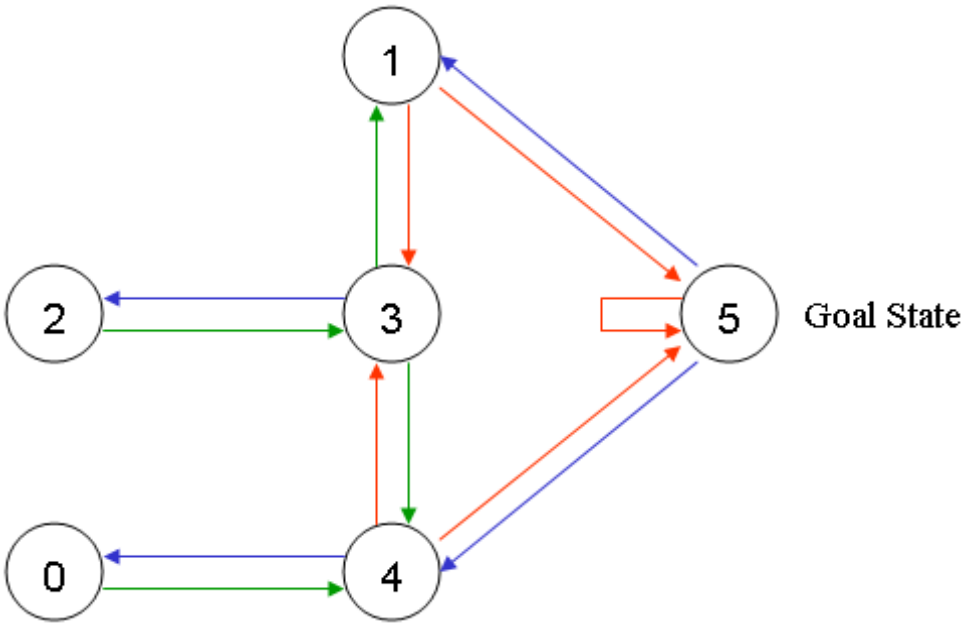
Reinforcement Learning : The Q-learning algorithm

Escape game

Suppose we have 5 rooms in a building connected by doors as shown in the figure below. We'll number each room 0 through 4. The outside of the building can be thought of as one big room (5). Notice that doors 1 and 4 lead into the building from room 5 (outside).

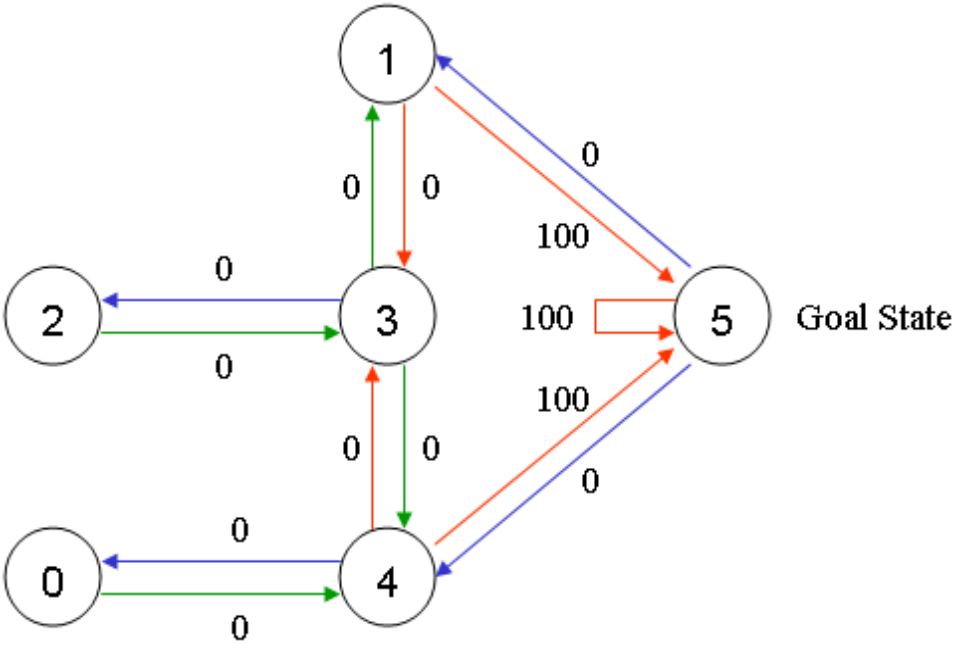


We can represent the rooms on a graph, each room as a node, and each door as a link.



For this example, we'd like to put an agent in any room, and from that room, go outside the building (this will be our target room). In other words, the goal room is number 5. To set this room as a goal, we'll associate a reward value to each door (i.e. link between nodes). The

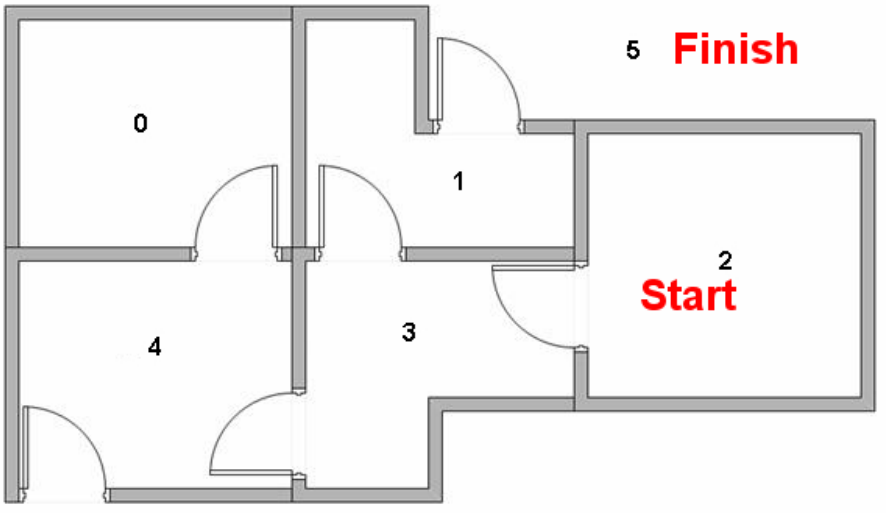
doors that lead immediately to the goal have an instant reward of 100. Other doors not directly connected to the target room have zero reward. Because doors are two-way (0 leads to 4, and 4 leads back to 0), two arrows are assigned to each room. Each arrow contains an instant reward value, as shown below:



Of course, Room 5 loops back to itself with a reward of 100, and all other direct connections to the goal room carry a reward of 100. In Q-learning, the goal is to reach the state with the highest reward, so that if the agent arrives at the goal, it will remain there forever. This type of goal is called an "absorbing goal".

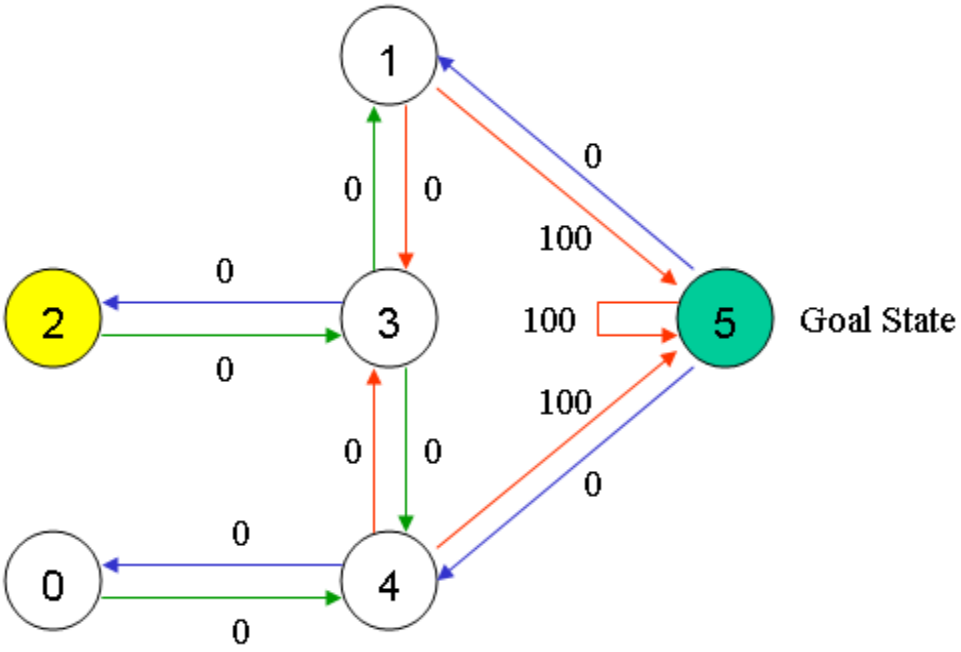
Imagine our agent as a dumb virtual robot that can learn through experience. The agent can pass from one room to another but has no knowledge of the environment, and doesn't know which sequence of doors lead to the outside.

Suppose we want to model some kind of simple evacuation of an agent from any room in the building. Now suppose we have an agent in Room 2 and we want the agent to learn to reach outside the house (5).



The terminology in Q-Learning includes the terms "state" and "action".

We'll call each room, including outside, a "state", and the agent's movement from one room to another will be an "action". In our diagram, a "state" is depicted as a node, while "action" is represented by the arrows.



Suppose the agent is in state 2. From state 2, it can go to state 3 because state 2 is connected to 3. From state 2, however, the agent cannot directly go to state 1 because there is no direct door connecting room 1 and 2 (thus, no arrows). From state 3, it can go either to state 1 or 4 or back to 2 (look at all the arrows about state 3). If the agent is in state 4, then the three possible actions are to go to state 0, 5 or 3. If the agent is in state 1, it can go either to state 5 or 3. From state 0, it can only go back to state 4.

We can put the state diagram and the instant reward values into the following reward table, "matrix R".

$$R = \begin{array}{c} \\ \text{State} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \text{Action} \\ \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} -1 & -1 & -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 & -1 & 100 \\ -1 & -1 & -1 & 0 & -1 & -1 \\ -1 & 0 & 0 & -1 & 0 & -1 \\ 0 & -1 & -1 & 0 & -1 & 100 \\ -1 & 0 & -1 & -1 & 0 & 100 \end{bmatrix} \end{array}$$

The -1's in the table represent null values (i.e.; where there isn't a link between nodes). For example, State 0 cannot go to State 1.

Now we'll add a similar matrix, "Q", to the brain of our agent, representing the memory of what the agent has learned through experience. The rows of matrix Q represent the current state of the agent, and the columns represent the possible actions leading to the next state (the links between the nodes).

The agent starts out knowing nothing, the matrix Q is initialized to zero. The transition rule of Q learning used here is simplified as follows:

Q(state, action) = R(state, action) + Gamma * Max[Q(next state, all actions)]

According to this formula, a value assigned to a specific element of matrix Q, is equal to the sum of the corresponding value in matrix R and the discount parameter Gamma, multiplied by the maximum value of Q for all possible actions in the next state.

The learning parameter alpha is set to 1 in order to focus only on the exploration phase.

The agent will explore from state to state until it reaches the goal. We'll call each exploration an episode. Each episode consists of the agent moving from the initial state to the goal state. Each time the agent arrives at the goal state, the program goes to the next episode.

The Q-Learning algorithm goes as follows:

---

1. Set the gamma parameter, and environment rewards in matrix R.
2. Initialize matrix Q to zero.
3. For each episode:

- Select a random initial state.
- Do While the goal state hasn't been reached.
    - Select one among all possible actions for the current state.
    - Using this possible action, consider going to the next state.
    - Get maximum Q value for this next state based on all possible actions.
    - Compute: Q(state, action) = R(state, action) + Gamma * Max[Q(next state, all actions)]
    - Set the next state as the current state.
- End Do

End For

The algorithm above is used by the agent to learn from experience. Each episode is equivalent to one training session. In each training session, the agent explores the environment (represented by matrix R), receives the reward (if any) until it reaches the goal state. The purpose of the training is to enhance the 'brain' of our agent, represented by matrix

Q. More training results in a more optimized matrix Q. In this case, if the matrix Q has been enhanced, instead of exploring around, and going back and forth to the same rooms, the agent will find the fastest route to the goal state.

The Gamma parameter has a range of 0 to 1 (0 <= Gamma > 1). If Gamma is closer to zero, the agent will tend to consider only immediate rewards. If Gamma is closer to one, the agent will consider future rewards with greater weight, willing to delay the reward.

To use the matrix Q, the agent simply traces the sequence of states, from the initial state to goal state. The algorithm finds the actions with the highest reward values recorded in matrix Q for current state:

Algorithm to utilize the Q matrix:

---

1. Set current state = initial state.
2. From current state, find the action with the highest Q value.
3. Set current state = next state.
4. Repeat Steps 2 and 3 until current state = goal state.

The algorithm above will return the sequence of states from the initial state to the goal state.

The goal of the exercise is to run few episodes of the Q-learning algorithm by hand, step by step. We'll start by setting the value of the learning parameter Gamma = 0.8, and the initial state as Room 1.

1. **Initialize the Q-table.**
2. **First episode : Look at the second row (state 1) of matrix R. There are two possible actions for the current state 1: go to state 3, or go to state 5. By random selection, we select to go to 5 as our action.**
    a. **Using Bellman equation, compute the new value of Q(1,5)**
    b. **Update the Q-table accordingly.**

    The next state, 5, now becomes the current state. Because 5 is the goal state, we've finished one episode. Our agent's brain now contains an updated matrix Q
3. **Second episode : For the next episode, we start with a randomly chosen initial state. This time, we have state 3 as our initial state and by random selection, we select to go to state 1 as our action. Update the Q(3,1) value.**

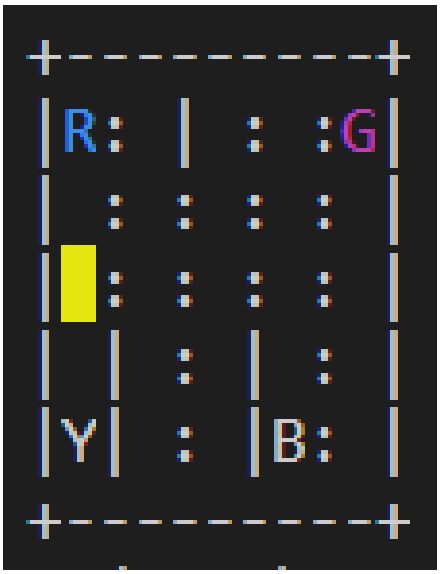
    The next state, 1, now becomes the current state. We repeat the inner loop of the Q learning algorithm because state 1 is not the goal state. Assume we select again randomly state 5. What is the new updated Q-table at the end of this episode ?
4. **Try to explore more episodes to find the Q-table reaching the convergence values.**
5. **Once the matrix Q gets close enough to a state of convergence, we know our agent has learned the most optimal paths to the goal state. Tracing the best sequences of states is as simple as following the links with the highest values at each state.**

    What is the best sequence to escape the building from room 2 ?

Working with OpenAI Gym

https://gym.openai.com/envs/Taxi-v3/

For this exercise, we'll use the Gym toolkit to create an agent that is able to play a simple taxi game. I encourage you to read their introduction first to get comfortable with Gym. With its easy API we can dive right into writing our first RL algorithm.



The game consists of a 5×5 matrix containing our taxi (green if manned) and four different cabstands labeled with letters. Also, there are some walls in the environment that our taxi can't pass. The task of the game is to pick up passengers at one of the cabstands and carry them to their destinations.

To do that, our agent has six possible actions to choose from. He can go north, south, east or west and he can try to pick up or drop off a passenger. This is called the **action space** of our taxi. Besides the action space we also have to define the **state space**. As we have 5*5 taxi locations, 5 different passenger locations (because we have to include the passenger being in our taxi) and 4 different destinations, the total number of states is 5*5*5*4 = 500.

Performing actions **rewards** the agent with points. He receives 20 points for a successful drop-off and loses 1 point for every time-step it takes. The latter results in our agent trying to solve the task fairly quick and prevents him from wandering around. There is also a -10 point penalty for illegal pick-up and drop-off actions and -1 penalty for driving against a wall.

When the Taxi environment is created, there is an initial Reward table that's also created, called `P`. We can think of it like a matrix that has the number of states as rows and number of actions as columns, i.e. a $states \times actions$ matrix.

Epsilon-greedy strategy

The Q-values are updated using the general formula :

$$Q^{new}(s_t, a_t) \leftarrow (1-\alpha) \cdot Q(s_t, a_t) + \alpha \cdot (r + \gamma \cdot max_a Q(s_{t+1}, a))$$

Where:

- α (alpha) is the learning rate ($0<\alpha\leq1$) - Just like in supervised learning settings, α is the extent to which our Q-values are being updated in every iteration.

- γ (gamma) is the discount factor ($0\leq\gamma\leq1$) - determines how much importance we want to give to future rewards. A high value for the discount factor (close to **1**) captures the long-term effective award, whereas, a discount factor of **0** makes our agent consider only immediate reward, hence making it greedy.

Since we initialize the Q-table with zeros, there is no best action in the start. Thus we have to choose randomly. This becomes problematic once one positive Q-value is found. That leads to the Q-function always returning that specific action. We wouldn't take the optimal strategy as we'd not get to know whether there is an even higher Q-value. That's where the epsilon parameter comes in to play. It decides whether we are using the Q-function to determine our next action or take a random sample of the action space. This has the advantage of not stopping to explore after we found a Q-value greater zero. Instead we are starting off exploring the action space and after every game played we decrease epsilon until reaching minimum. With enough exploration done, we can start exploiting the learnt. We call that **exploration-exploitation trade-off**, which is necessary to control the agent's greed.

The learning agent in Python using Gym

First of all, we need to install the Gym environment. Executing the following in a Jupyter notebook should work:

```
!pip install cmake 'gym[atari]' scipy
```

We can now initialize the gym environment.

```
import gym.spaces

env = gym.make("Taxi-v2")
```

We continue by creating the Q-table as numpy array. The size of the spaces can be accessed as seen below and np.zeros() just creates an array of the given shape filled with zeros.

```
import numpy as np

state_space = env.observation_space.n
action_space = env.action_space.n

qtable = np.zeros((state_space, action_space))
```

The last thing that needs to be predefined are the hyper-parameters. The learning-rate and discount-factor in our Q-function can be tweaked to improve the learning process. You can leave them unchanged for now and deal with them later.

```
epsilon = 1.0           #Greed 100%

epsilon_min = 0.005     #Minimum greed 0.05%

epsilon_decay = 0.99993 #Decay multiplied with epsilon after each episode

episodes = 50000        #Amount of games

max_steps = 100         #Maximum steps per episode

learning_rate = 0.65

gamma = 0.65
```

All that's left to do is implementing the procedure of playing games over and over again. In every episode we reset the state. After resetting we choose an action, step the game forward and update our Q-table until the game is over or we reach the maximum steps allowed. Finally we decrease our epsilon each episode.

```
for episode in range(episodes):

    # Reset the game state, done and score before every episode/game
    state = env.reset() #Gets current game state
    done = False        #decides whether the game is over
    score = 0

    for _ in range(max_steps):

        # With the probability of (1 - epsilon) take the best action in our Q-table
        if random.uniform(0, 1) > epsilon:
            action = np.argmax(qtable[state, :])
        # Else take a random action
        else:
            action = env.action_space.sample()

        # Step the game forward
        next_state, reward, done, _ = env.step(action)

        # Add up the score
        score += reward

        # Update our Q-table with our Q-function
        qtable[state, action] = (1 - learning_rate) * qtable[state, action] \
            + learning_rate * (reward + gamma * np.max(qtable[next_state,:]))

        # Set the next state as the current state
        state = next_state

        if done:
            break

    # Reducing our epsilon each episode (Exploration-Exploitation trade-off)
    if epsilon >= epsilon_min:
        epsilon *= epsilon_decay
```

Comparing with a random agent

Another attempt to solve the environment is an agent that just takes random actions. Neither does he learn, nor remember anything. We are only restricting the allowed amount of moves as before. The implementation is a slimmed version of the Q-learning agent – we are leaving the whole Q-table aspect out.

Extend the code by implementing the random agent and comparing with the learning one.

More information/Code for this exercise can be found at :

https://www.learndatasci.com/tutorials/reinforcement-q-learning-scratch-python-openai-gym/

https://www.novatec-gmbh.de/en/blog/introduction-to-q-learning/