hw2

February 3, 2021

1 HW 2

Max Schrader

1.1 Declaring the variables

[1]:
```
MOVEMENTS = {"A": {"moves": ["B", "C", "D"], "reward": [1.0, 1.6, 1.9]},
             "B": {"moves": ["E", "F", "G"], "reward": [1.0, 1.5, 1.0]},
             "C": {"moves": ["E", "F", "G"], "reward": [1.8, 1.8, 1.7]},
             "D": {"moves": ["E", "F", "G"], "reward": [1.9, 1.6, 1.5]},
             "E": {"moves": ["H", "L", "O"], 'reward': [1.9, 1.6, 1.8]},
             "F": {"moves": ["H", "L", "O"], 'reward': [1.4, 1.4, 1.8]},
             "G": {"moves": ["H", "L", "O"], 'reward': [1.0, 1.1, 1.1]},
             "H": {"moves": ["P"], 'reward': [1.3]},
             "L": {"moves": ["P"], 'reward': [1.8]},
             "O": {"moves": ["P"], 'reward': [1.8]},
             "P": {"moves": [], 'reward': []}}
NUM_STATES = 4
```

1.2 Creating the Functions

[2]:
```
def compose_R(movements):
    """
    compose_R creates the reward matrix from the dictionary above. It is just a␣
↪helper function,
    as the R matrix is hard to create visually.

    :returns: 2d matrix, {2d matrix index: state name}, {state name: 2d matrix}
    """
    states = sorted(list(movements.keys()))
    R = [[0 for _ in states] for _ in states]
    for j, s_0 in enumerate(states):
        for s_1, reward in zip(movements[s_0]["moves"],␣
↪movements[s_0]["reward"]):
            position = [i for i, s in enumerate(states) if s == s_1]
            R[j][position[0]] = reward
```

```
    return R, {ind: letter for ind, letter in enumerate(states)}, {letter: ind␣
↪for ind, letter in enumerate(states)}


def backwards_algo(options, r, t, state_num=NUM_STATES - 1):
    """
    This is the main function, which implements the backward algorithm. It␣
↪searches recursively for
    the route that maximizes the score.

    INPUTS:
    options: [(current score, current index, [current sequence]), ...] , all of␣
↪the options with max scores
    r: reward matrix
    t: index of times ran
    state_num = # of iterations to make, (equal to the number of states - 1, as␣
↪the first iteration is made in the function call)

    OUTPUTS:
    :returns: [(final score, ending index, [state sequence])]
    """
    if t < state_num:
        option_container=[]
        for j, (reward, move, state_history) in enumerate(options): # this loop␣
↪handles multiple options with the same score
            local_options = [(r[i][move] + reward, i, state_history + [i]) for␣
↪i, row in enumerate(r) if row[move] > 0]
            max_option = get_max_options(local_options)
            option_container.extend(max_option)
        return backwards_algo(options=get_max_options(option_container), r=r,␣
↪t=t+1)
    return options


def get_max_options(reward_tuple):
    """
    get_max_options is a helper function to find the max score from a list of␣
↪options,
    while preserving the other information in the tuple. it returns a list of␣
↪all values == to the max reward,
    as there are occurances where reward via 2 different routes are the same

    INPUTS:
    reward_tuple: a list of reward tuples (same as options in bacckwards_algo)

    OUTPUTS:
```

```
    :returns: a list of tuples with the max reward
    """
    max_reward = max(reward_tuple, key=lambda item:item[0])
    return [(r, i, states) for r, i, states in reward_tuple if r ==␣
↪max_reward[0]]


def compose_pretty_state_sequence(reverse_sequence, state_dict):
    """
    compose_pretty_state_sequence is a helper function to reverse the sequence␣
↪returned by the backwards algorithm.
    It also converts the index sequnce into a state name sequnce

    INPUTS:
    reverse_sequence: the final state sequence returned by the backwards␣
↪algorithm
    state_dict: a mapping of indexes to state names, created by compose_R

    OUTPUTS:
    :return: the pretty state sequence
    """
    return [state_dict[val] for val in reverse_sequence[::-1]]
```

1.3 Initializing the Variables

[3]:
```
#  creating R
R, *state_index = compose_R(MOVEMENTS)
print("R = ")
[print(r) for r in R];

# getting the index of the final state (the starting state of the backwards␣
↪algorithm)
starting_state = [state_index[1]['P']]
```

```
R =
[0, 1.0, 1.6, 1.9, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 1.0, 1.5, 1.0, 0, 0, 0, 0]
[0, 0, 0, 0, 1.8, 1.8, 1.7, 0, 0, 0, 0]
[0, 0, 0, 0, 1.9, 1.6, 1.5, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1.9, 1.6, 1.8, 0]
[0, 0, 0, 0, 0, 0, 0, 1.4, 1.4, 1.8, 0]
[0, 0, 0, 0, 0, 0, 0, 1.0, 1.1, 1.1, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1.3]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1.8]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1.8]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

1.4 Calculating the route of max reward

[4]:
```
final_data = backwards_algo(options=get_max_options([(R[i][starting_state[-1]],␣
 ↪i, starting_state + [i]) for i, row in enumerate(R) if␣
 ↪row[starting_state[-1]] > 0]), r=R, t=0)
print(f"Raw Result:\t{final_data[0]}")
print(f"Score:\t\t{final_data[0][0]}\nSequence:
 ↪\t{compose_pretty_state_sequence(reverse_sequence=final_data[0][-1],␣
 ↪state_dict=state_index[0])}")
```

```
Raw Result:     (7.4, 0, [10, 9, 4, 3, 0])
Score:          7.4
Sequence:       ['A', 'D', 'E', 'O', 'P']
```