IA - Reinforcement Learning Homework 20/21

Leonardo Serilli (Mat 274426)

Part A

(a)

A state is represented **a triple (x, y, egg_beater)** where **x, y** are the coordinates of the chef and **egg_beater** says if the chef own the tool he need in order to cook.

$$S = \{(x, y, egg\_beater) \mid x \in \{1, 2, 3, 4\},\ y \in \{1, 2, 3, 4\},\ egg\_beater \in \{0, 1\}\}$$

In order to compute the cardinality **|S|** we must take into account that the chef can be in any of the **sixteen position with or without the cooking's tool**, so it is:

$$|S| = 32$$

(b)

There are four possible moves possible for the agent

$$A = \{left, right, top, down\},\ |A| = 4$$

(c)

The **transition function**

$$P : S \times S \times A \to \{0,\ 1\}$$

which maps

$$(s',\ s,\ a)\ to\ P(s'\mid s,\ a)$$

has dimensionality **32 * 32 * 4 = 4096**

(d)

The transition function P for any state s and action a is reported in the **attached spreadsheet** named **transFunct.xlxs**.

> Here is an entry in the spreadsheet:

x	y	egg_beater	x'	y'	egg_beater'	a	P
1	1	0	1	1	0	left	0

Here is the **python3 code** used in order to **generate all the instances** of the function and **append them to the spreadsheet.**
It has a method which check for the validity of a transition in order to **compute the output** for the function.

```
#!/usr/bin/python3
import csv

# add an instance of the transition function to the spreadsheet
def add_toCsv(row):
    with open("tranFunct.csv", 'a', newline='', encoding='utf-8') as f:
        writer = csv.writer(f)
        writer.writerow(row)

# check if the is a valid transition, so it computes the output of the function
def isValid(row):

    if(row[6] == "left"):
        if(row[0]==1 or row[0]==2 and row[1]==3 or row[3]>row[0] or row[1] !=
row[3]):
            return 0
    if(row[6] == "right"):
        if(row[0]==4 or row[0]==1 and row[1]==3 or row[3]<row[0] or row[1] !=
row[3]):
            return 0
    if(row[6] == "top"):
        if(row[4]<row[1] or row[1]!=row[3] or row[1]==4 or  (row[1]==1 and row[0]
<4) or (row[1]==2 and row[0]>1 and row[0]<4) or (row[0]==1 and row[1]==3)):
            return 0
    if(row[6] == "down"):
        if(row[4]>row[1] or row[1]!=row[3] or row[1]==1 or (row[1]==2 and row[0]
<4) or (row[1]==3 and row[0]>1 and row[0]<4) or (row[1]==4 and row[0]==1)):
            return 0
    if(row[0]!=row[3] and row[1]!=row[4]):
        return 0
    if((row[3]>row[0]+1) or (row[4]>row[1]+1)):
        return 0
    if((row[3]<row[0]-1) or (row[4]<row[1]-1)):
        return 0
    if((row[0]==row[3] and row[1]==row[4]) and not (row[0]==1 and row[1]>3)):
        if(row[2]==0 and row[2]==1):
            return 1
        return 0
```

```
    return 1

row = [0 for i in range(8)]
mvs = ["left", "right", "top", "down"]
c=1

# build all the possible instances for the transition function
for x in range(1, 5):
    row[0] = x
    for y in range(1, 5):
        row[1] = y
        for tool in range(2):
            row[2] = tool
            for x1 in range(1, 5):
                row[3] = x1
                for y1 in range(1, 5):
                    row[4] = y1
                    for tool1 in range(2):
                        row[5] = tool1
                        for a in mvs:
                           row[6] = a
                           row[7] = isValid(row)
                           add_toCsv(row)
```

(e)

To define a reward function we must take in two account the two phases of the problem, the one in which the agent do not own the tool and the other in which it does.

Given $s, s' \in S \mid s := \{x, y, egg\_beater\}, s' := \{x', y', egg\_beater'\}$ and $a \in A$:

1. **In the first case (Agent don't own the tool)** we will reward the agent in the sequent way:
    1. **15 point if** he makes a move that brings him in the position of the egg beater.

$$R(s, a, s') = 15 \iff s = \{x = 1,\ y = 2,\ egg\_beater = 0\} \ \wedge\ s' = \{x = 1,\ y = 3,\ egg\_beater = 1\}$$

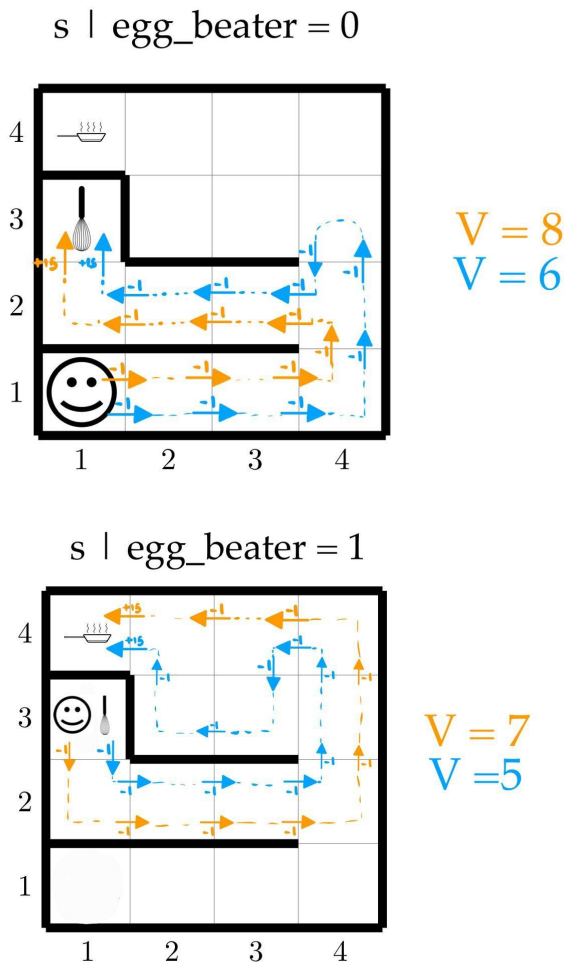
2. **In the second case (Agent own the tool)** we will reward the agent in the sequent way:
    1. **15 point if** he makes a move that brings him in the position of the frying pan

$$R(s, a, s') = 15 \iff s = \{x = 2,\ y = 4,\ egg\_beater = 1\} \ \wedge\ s' = \{x = 1,\ y = 4,\ egg\_beater = 1\}$$

Since we are interested in the **shortest path**, so we can give a **-1 reward for any other** triple $\{s,\ a,\ s'\}$ and easily set the **scaling factor** γ **>0.***

*In this way the **agent will only care for the** path that brings him to the cooking tool** in the phase where it doesn't own it and, **once he gets it**, for the **path that brings him in the final state**, because it will always give the best output for the **value function.**

> Here follow an example that show the value function for the shortest path (in orange) and for another random path (in blue), for both cases:



(f)

Any discount factor **greater than zero** $\gamma > 0$ **won't affect the optimal policy**, because a path that brings him to his objective (either the egg beater or the frying pan), with respect of any path that doesn't, would have a much bigger impact on the value function.

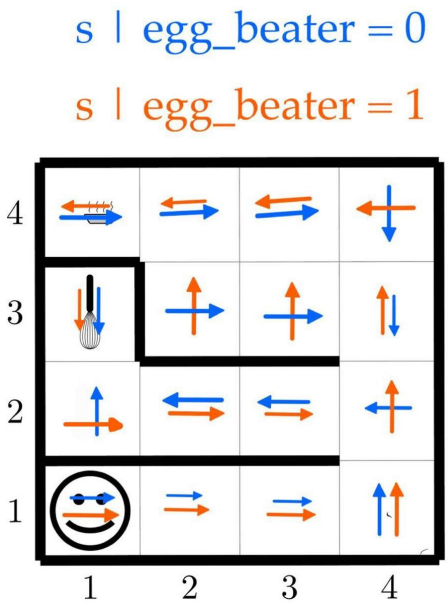
obviously for $\gamma = 0$ the most of the states would have the same value function output for any path the agent will follow that will bring him to the goal.

(g)

There are $|A|^{|S|}$ possible policies:

$$|A|^{|S|} = 4^{32} = 18446744073709551616$$

(h)



(i)

It is Deterministic

(j)

A stochastic policy wouldn't bring any advantage because:

- The **environment itself is not stochastic.**

  > if it was my policy would fail as it will always pick the exact same action at the exact same state.

- We do not have **partially observable states**.

  > A stochastic policy is more robust when part of the state is hidden from the agent, as it naturally takes the uncertainty about inferring the hidden states into account.

Part B

(a)

The transition function P for the tired version of the problem is reported in the **attached spreadsheet** named **tired_transFunct.xlxs**.

> Here is an entry in the spreadsheet, where you can see the result we obtain choosing the wrong directions:

x	y	egg_beater	x'	y'	egg_beater'	a	P	probability
1	1	0	1	1	0	left	0	0.5
1	1	0	1	1	0	down	0	0.3
1	1	0	1	2	0	top	0	0.2

Here is the **python3 code** used in order to **generate all the instances** of the function and **append them to the spreadsheet.**
It's the same code of **part A**, but it takes into account also the wrong directions that can be taken by the agent.

```
#!/usr/bin/python3
import csv

PERPENDICOLAR_LEFT = {
        "top": "left",
        "down": "right",
        "right": "top",
        "left": "down"}

PERPENDICOLAR_RIGHT = {
        "top": "right",
        "down": "left",
        "right": "down",
        "left": "top"}

# add an instance of the transition function to the spreadsheet
def add_toCsv(row):
    with open("tired_transFunct.csv", 'a', newline='', encoding='utf-8') as f:
        writer = csv.writer(f)
        writer.writerow(row)

# check if the is a valid transition
def isValid(row):

    if(row[6] == "left"):
        if(row[0]==1 or row[0]==2 and row[1]==3 or row[3]>row[0] or row[1] !=
row[3]):
            return 0
    if(row[6] == "right"):
        if(row[0]==4 or row[0]==1 and row[1]==3 or row[3]<row[0] or row[1] !=
row[3]):
            return 0
    if(row[6] == "top"):
        if(row[4]<row[1] or row[1]!=row[3] or row[1]==4 or  (row[1]==1 and row[0]
<4) or (row[1]==2 and row[0]>1 and row[0]<4) or (row[0]==1 and row[1]==3)):
            return 0
    if(row[6] == "down"):
        if(row[4]>row[1] or row[1]!=row[3] or row[1]==1 or (row[1]==2 and row[0]
<4) or (row[1]==3 and row[0]>1 and row[0]<4) or (row[1]==4 and row[0]==1)):
            return 0
    if(row[0]!=row[3] and row[1]!=row[4]):
        return 0
    if((row[3]>row[0]+1) or (row[4]>row[1]+1)):
        return 0
    if((row[3]<row[0]-1) or (row[4]<row[1]-1)):
        return 0
    if((row[0]==row[3] and row[1]==row[4]) and not (row[0]==1 and row[1]>3)):
        if(row[2]==0 and row[2]==1):
            return 1
        return 0

    return 1

row = [0 for i in range(9)]
mvs = ["left", "right", "top", "down"]
c=1
```

```
# build all the possible instances of the transition function
for x in range(1, 5):
    row[0] = x
    for y in range(1, 5):
        row[1] = y
        for tool in range(2):
            row[2] = tool
            for x1 in range(1, 5):
                row[3] = x1
                for y1 in range(1, 5):
                    row[4] = y1
                    for tool1 in range(2):
                        row[5] = tool1
                        for a in mvs:
                            prob = [0.5, 0.3, 0.2]
                            tmp_a = a
                            tmp_row = row

                            # we now take into account the wrong directions the
agent can take
                            for i in range(3):
                                tmp_row[6] = tmp_a
                                tmp_row[7] = isValid(tmp_row)
                                tmp_row[8] = prob[i]
                                add_toCsv(tmp_row)

                                if(i==0):
                                    tmp_a = PERPENDICOLAR_LEFT[a]
                                if(i==1):
                                    tmp_a = PERPENDICOLAR_RIGHT[a]

                                if(tmp_a=="left"):
                                    if(row[0]-1>0):
                                        tmp_row[4] = row[1]
                                        tmp_row[3] = row[0]-1
                                if(tmp_a=="right"):
                                    if(row[0]+1<5):
                                        tmp_row[4] = row[1]
                                        tmp_row[3] = row[0]+1
                                if(tmp_a=="top"):
                                    if(row[1]+1<5):
                                        tmp_row[3] = row[0]
                                        tmp_row[4] = row[1]+1
                                if(tmp_a=="down"):
                                    if(row[1]-1>0):
                                        tmp_row[3] = row[0]
                                        tmp_row[4] = row[1]-1
```

(b)

Of course it changes. The policy in part A doesn't take into account the wrong direction the agent can take, because the probability of following the right direction is just 50% so the policy doesn't guarantee an optimal solution in the case where the agent can change his route, because a wrong action would get a negative reward.

(c)

Of course the **value of the optimal policy will change**, because the evaluation depends from the route the agent follows, so the **value computed in this case would be less equal than the value computed for the problem of part A** . It's easy to see that the transition matrix changes with this new configuration of the problem