Methods of Artificial Intelligence: Task 1 – Markov Decision Process

Deadline

Upload your program together with the gridfiles in a zip or rar archive with the name

task1_group_*yourgroupnumber*

into your group's studip folder in the tutorial course until January 29th 2018, 12:00pm.

Benchmark files

The files your program will read are located in the Stud.IP folder "Programming task 1" in the Methods of Artificial Intelligence Tutorial course.

Short Description

Implement a program that reads in a grid world file and computes an optimal policy using Policy Iteration. The number of evaluation steps per iteration and the gamma value should be configurable via user input.

Program description and evaluation

Your program must be able to perform the following tasks:

Reading grid world files	Your program should be able to read the provided files and to transform them into appropriate data structures that can be used by your implementation.
Your code works	Your program does not crash for regular inputs and performs policy iteration correctly.
1step / n-step policy iteration	The user should be able to control policy iteration by running an adjustable number of policy evaluation steps and running the policy improvement phase.
automatic Iteration until convergence	The user should be able to call a function with a pathname to a gridworld file and n, the number of evaluation steps as specified above, that performs policy iteration until the values converged and the policy is stable. Then it should visualize the final policy (and the corresponding policy evaluation)

manual iteration	The user should be able to call a function with a pathname to a gridworld file. Your program should start a dialogue with the user and expect input: In each step the user can choose between 1-step or n-step iteration. The policy gets evaluated n times and updated once. The resulting policy and the policy evaluation will be visualized and the dialogue starts again. The user should also be able to terminate the program in the dialogue.

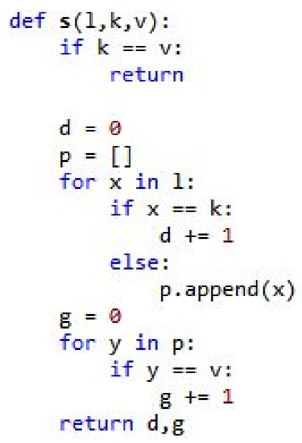
Additionally, we will evaluate the structure of your code as well as whether it is readable!

Commenting your code and self-explanatory variable names	Explain what your functions are computing and what purpose specific variables serve. Provide a short description of what your function is doing if it is not clear from the function's name.
Code structure	Perform distinct steps in your program in separate functions. Structure your code neatly and avoid excessive redundancy if possible.
Short readme file	text file that explains how to run your coude (what is the main class for java programs, any packages we need to install for python programs, etc.). We should be able to run your program following only your readme description.

You can find the underlying principles of the code in the lecture slides. In general, there are multiple ways in which you can implement a Markov Decision Process.

Writing readable code

Here is an example of what bad code looks like in python:

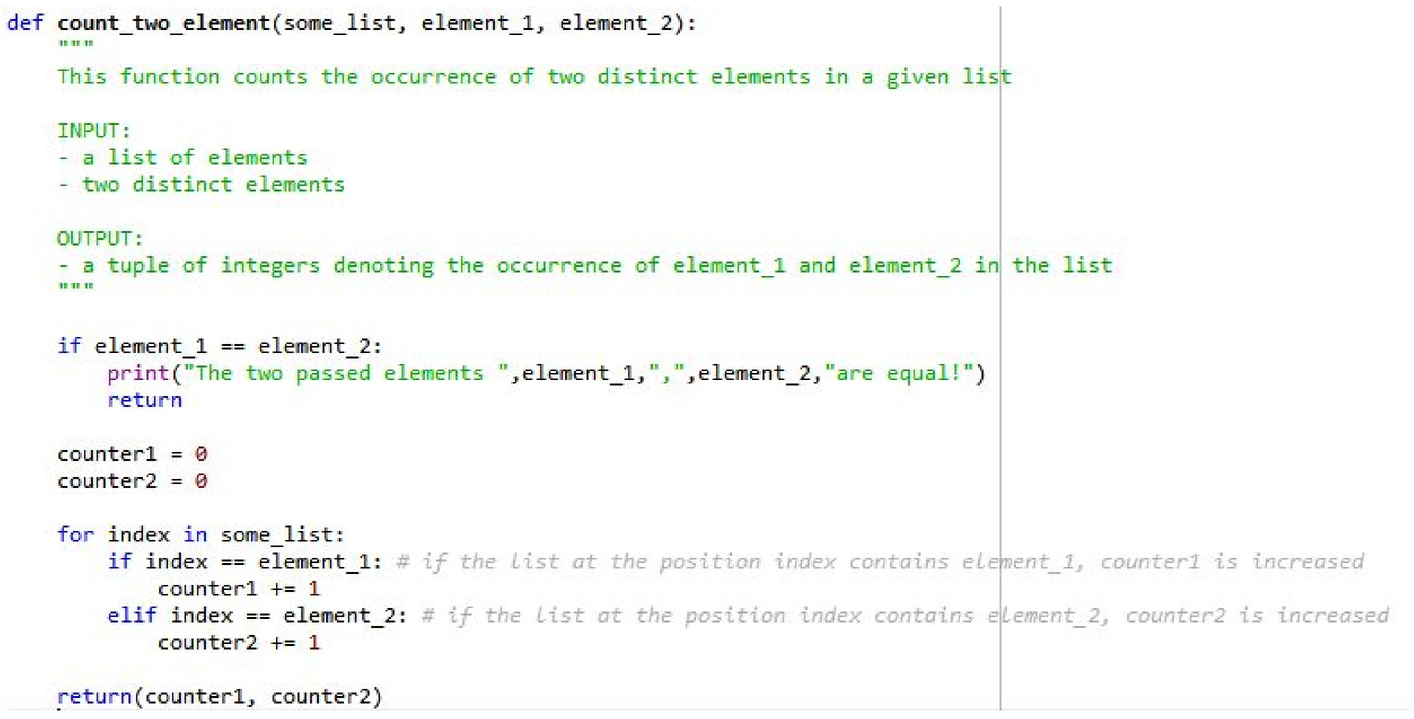


```python
def s(l,k,v):
    if k == v:
        return

    d = 0
    p = []
    for x in l:
        if x == k:
            d += 1
        else:
            p.append(x)
    g = 0
    for y in p:
        if y == v:
            g += 1
    return d,g
```

Even if you know python syntax, the code is not very readable, as variables consist of only one letter. Additionally, there are no comments describing what is being computed.

Look at the same code again but with comments and descriptive variable names:



```python
def count_two_element(some_list, element_1, element_2):
    """
    This function counts the occurrence of two distinct elements in a given list

    INPUT:
    - a list of elements
    - two distinct elements

    OUTPUT:
    - a tuple of integers denoting the occurrence of element_1 and element_2 in the list
    """

    if element_1 == element_2:
        print("The two passed elements ",element_1,",",element_2,"are equal!")
        return

    counter1 = 0
    counter2 = 0

    for index in some_list:
        if index == element_1: # if the list at the position index contains element_1, counter1 is increased
            counter1 += 1
        elif index == element_2: # if the list at the position index contains element_2, counter2 is increased
            counter2 += 1

    return(counter1, counter2)
```