UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Faculty of Engineering, Built Environment and Information Technology

EAI 320

Intelligent Systems

Practical 1 Guide

6 February 2020

Concept: Prof. Warren P. du Plessis
Guide written by: Llewellyn Strydom
Guide revised by: Michael Teles

I. General Instructions

This section contains general instructions which are applicable to all reports.

- The prescribed LaTeX format should be used, and the generated portable document format (PDF) file should be submitted.
- The commented Python source code should also be submitted.
- Reports must be submitted via the EAI 320 ClickUP page. Do not email reports to the lecturer or Assistant Lecturer (AL) as emailed reports will be considered not to have been submitted.
- The names of submitted files must have the format provided below.
  `eai320_prac_{practical number}_{student number}.{extension}`
- No late assignments will be accepted. No excuses for late submission will be accepted.
- Each student must do their own work. Academic dishonesty is unacceptable and cases will be reported to the university Legal Office for suspension.
- The report must include the standard cover-page declaration for individual assignments provided in the General Study Guide of the Department of Electrical, Electronic and Computer Engineering. The declaration is not included in the LaTeX template automatically and needs to be explicitly uncommented to indicate agreement.
- All information from other sources must be clearly identified and referenced.
- Text and code may not be included as images, and reports that do so will be considered not to have been submitted. Axis labels, legends, and other information normally included in figures may form part of a figure image.
- Computer code. should not be included in the report as it is submitted separately.
- The code provided to students should also not be included in submissions.
- Any attempt to interfere with the operation of the framework used (e.g. to modify the scores of agents) will be regarded as academic dishonestly.

A. *Submission*

Reports should only be submitted via the

- `Practicals` → `Practical 1` → `Practical 1 Code`

links on the EAI 320 ClickUP page.

Only assignments submitted in via ClickUP will be accepted. Do not email reports to the lecturer or AL as emailed reports will be ignored.

Late reports will not be accepted! Accepting late reports is extremely unfair on those students who submit their work timeously because their tardy colleagues are effectively given additional time to complete the same work. Students are advised to submit the day before the deadline to avoid inevitable problems with ClickUP, internet connections, unsynchronised clocks, load shedding, hard-drive failure, computer theft, etc.. Students who choose to submit close to the deadline accept the risk associated with their actions, and no excuses for late submissions will be accepted.

Students will be allowed to submit updated copies of their reports until the deadline, so there will be no excuse for submitting late. Rather be marked on an incomplete early version of your report than fail to submit anything.

B. *Academic Dishonesty*

Academic dishonesty is completely unacceptable. Students should thus familiarise themselves with the University of Pretoria's rules on academic dishonesty summarised in the study guide and the university's rules. Students found guilty of academic dishonesty will be reported to the Legal Office of the University of Pretoria for suspension.

Students are required to include the standard cover page for individual assignments provided in the General Study Guide of the Department of Electrical, Electronic and Computer Engineering as part of their reports. This standard cover page includes a statement that the student submitting the report is aware of the fact that academic dishonesty is unacceptable and a statement that the submitted work is the work of that student. Failure to include this cover page will mean that the report will be considered not to have been submitted.

While students are encouraged to work together to better understand the work, each student is required to independently write their own code and report. No part of any student's work may be the same as any part of another student's work.

Students should clearly indicate material from other sources and provide complete references to those sources. Examples of commonly-used sources include the textbook [1] and this document [2]. Note that this does not mean that students may reuse code and/or information found in books, on the internet, or in other sources as students are required to complete the tasks themselves.[1]

II. Scenario

Rock-paper-scissors (RPS) is a popular game played between two adversaries, where each adversary simultaneously forms one of three shapes with an outstretched hand [3], or using some other representation. Throughout this course, the framework provided by [4] will be used to teach students how different artificial intelligence (AI) principles and algorithms can be applied to the game of RPS.

The RPS terminology that will be used in this course is shown below.

Object: One of three hand shapes (*viz.* rock (R), paper (P), or scissors (S)).

Move: One game where each opponent plays an object, resulting in a win, loss or draw for each player. The RPS framework used for this practical assignment [4] refers to a move as a round.

Match: A series of moves between two players to decide the overall winner of the game.

Sequence: A particular order of objects for one opponent (e.g. rock, rock, scissors).

This assignment will task students with using two common search strategies to find a hidden sequence that will allow it to exploit an opponent's weaknesses during a match of RPS.

III. Instructions

For this practical assignment, students will investigate the effectiveness of two uninformed search techniques, namely depth-first search (DFS) and breadth-first search (BFS). The algorithms will be used to explore different possible sequences in a game of RPS.

The four major goals for this practical are thus

1) implementing a search tree structure in Python,
2) implementing a DFS and BFS algorithm,
3) implementing a RPS agent that can beat `breakable.py`, which is provided in Listing 3 in Appendix B on page 6, and

[1] The objective of all academic assignments is fundamentally that students learn by completing the assignments. Merely reusing code and/or information found elsewhere defeats this objective because a key part of the learning process is performing the tasks oneself.

Listing 1: An example code segment showing the class for a tree.

```
class node(object):
  def __init__(self,object,children = []):
    self.object = object
    self.children = children
```

Listing 2: Code showing how to create a tree using the class shown in Listing 1.

```
tree = node("Start", [
 node("R",[
     node("R"),
     node("P"),
     node("S")]),
 node("P",[
     node("R"),
     node("P"),
     node("S")]),
 node("S",[
     node("R"),
     node("P"),
     node("S")])
                    ])
```

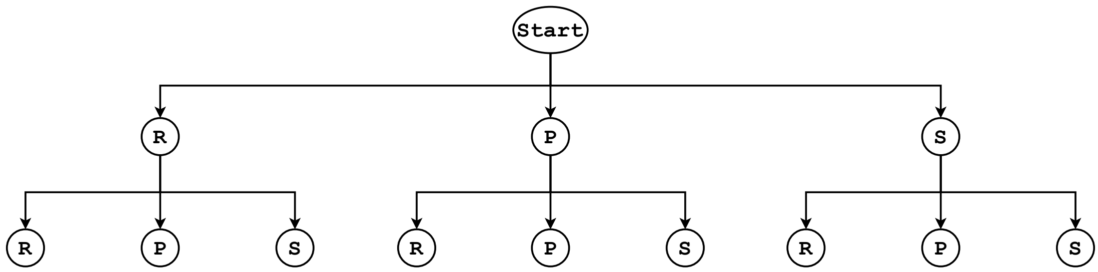

Fig. 1: The tree generated by the code in Listing 2.

A. *Task 1*

A search tree structure should be created in Python that can be used by any type of search algorithm. For this task the depth of the search tree must be dynamic and should only be limited by memory constraints.

Any process or method may be used to create the search tree. The recommended approach is to use a Python Class to represent nodes in a tree. Each of the nodes are linked and a recursive function can be used to build the tree.

An example code segment to create a tree class is provided in Listing 1, and a tree can be constructed using the code in Listing 2, resulting in the node tree shown in Figure 1.

1) Suggestion: A `depth` parameter may be passed to the constructor of the tree to allow the tree up to be constructed to the specified depth.

TABLE I: Sequences Generated by BFS Applied to the Tree in Figure 1.

Step	1	2	3	4	5	6	7	8	9	10	11	12
Sequence	R	P	S	RR	RP	RS	PR	PP	PS	SR	SP	SS

B. *Task 2*

The next task is to write BFS and DFS algorithms that can be used to incrementally search the tree built in Task 1. Each search algorithm must return a complete list of the sequences represented by each node, in the order in which they are visited by the algorithm. For example, applying the BFS algorithm to the tree in Figure 1 would give the result shown in Table I.

C. *Task 3*

For this task, the code from Tasks 1 and 2 will be used to determine the objects that will be played against an agent called `breakable.py`. This agent is programmed to play randomly without ever repeating an object, until a certain sequence of objects, called the break sequence, is played. The break sequence will be unknown and will have a length between 2 and 5, inclusive. Once the break sequence is played, the agent `breakable.py` will repeat its last move an unknown number of times, before it starts playing randomly again. While the agent is repeating itself, it is possible to exploit it and win every move until the agent starts playing randomly again. At this stage, the known break sequence is simply played again to cause the agent to repeat its objects again.

In summary, the tree should be traversed using BFS and DFS respectively, until the break sequence is found. Once the break sequence is known, it can be used to defeat the agent.

1) Suggestions: The primary goal of this assignment is that the specified search algorithms should be implemented. As a result, other approaches to solving the problem are not acceptable (e.g. playing randomly until a repeat is detected). Furthermore, the properties of the various search algorithms should be compared, so modifications to the search algorithms should not be implemented as such modifications may obscure the properties of the search algorithms (e.g. storing the previous five objects played to avoid the need for further search after the accidental breaks described in the next paragraph).

It is possible to accidentally stumble upon the break sequence before reaching the correct node in the tree. For example, steps 3 and 4 in Table I could trigger the break sequence SR, even though the current sequence being tested is RR. If this occurs, the assumed break sequence will not work consistently, and searching of the tree should be resumed.

The BFS and DFS can be implemented recursively or iteratively, but it is strongly recommended that your implementation is iterative.

Implementing the entire search process before the first match takes place is not recommended as some of the aspects of the various search algorithms will be not be clearly seen by this approach.

If one implements this task successfully, one should expect to win 100% of the games against the agent `breakable.py`, but it is possible that this may not occur. Can you explain why?

IV. Submission Requirements

In general, marks will primarily be awarded for well-formatted and well-commented code.

A. Code Instructions

Each student is required to submit code conforming to the requirements listed below. A mark of zero will be awarded if the submitted code file does not run, produces errors, or does not follow the instructions.

- All code must be commented to a point where the implementation of the underlying algorithm can be determined.
- The submission must be a single file.
- Submitted code will be evaluated using the command

  `rpsrunner.py <your_filename>.py breakable.py`

  to ensure that the agent `breakable.py` is successfully beaten.
- All print statements and any other functions that were used for unit testing etc. must be removed before submission.
- All submissions must be written using Python 3 as a result of the fact that `rpsrunner.py` is written in Python 3.
- The code submitted must be the final implementation of Task 3.
- The first line of the code must declare a variable bfs_dfs to select between the BFS (bfs_dfs = 0) and DFS algorithms (bfs_dfs = 1).
- Append the output of your program as a comment to the end of your file.
- TurnItIn will not accept code with the file extension `.py`, so the file extension should be changed from `.py` to `.txt` before submission.

References

[1] S. J. Russell and P. Norvig, *Artificial intelligence: A modern approach*, 3rd ed. Prentice Hall, 2010.

[2] L. Strydom, *EAI 320 – Practical 1 Guide*, University of Pretoria, 7 Feb. 2019.

[3] (2019, 6 Feb.) Rock-paper-scissors – wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Rock-paper-scissors

[4] B. Knoll. (2011, 6 Feb.) Rock paper scissors programming competition. [Online]. Available: http://www.rpscontest.com/

Appendix A
Abbreviations

Appendix B
Source Code for the Agent breakable.py

Listing 3: The source code for the agent breakable.py.

```
# An agent which plays randomly without repeated the objects it plays until a specific sequence is played by its opponent. The agent then repeats the last object it played a random number of times before resuming its normal sequence.
#
# Uses ideas from information in agents submitted to http://www.rpscontest.com
# Written by: W. P. du Plessis
# Last update: 2019-02-07


import random

if input == "":

    # The range of possible lengths of the break sequence.
    break_min = 2
    break_max = 5
    # The maximum number of repeats when repeating.
    repeat_max = 10

    # This line allows repeatable results.
    #random.seed(0)

    # The length of the break sequence is break_min to break_max objects.
    length = random.randint(break_min, break_max)

    # Generate the random break sequence.
    sequence = [ "X" ]*length
    for counter in range(length):
        sequence[counter] = random.choice(["R", "P", "S"])

    # Initialise the history.
    # Use "X" as the initial value because it does not match any of the objects.
    history = [ "X" ]*length

    # Initialise the variable for the number of repeats.
    repeat = 0

```

```python
    # Play randomly for the first move.
    previous = random.choice(["R", "P", "S"])

else:

    # Update the history.
    history.pop(0)
    history.append(input)

    # Initialise the repeat counter if the opponent's last objects match the break sequence.
    if history == sequence:
        repeat = random.randint(length, repeat_max)


# If the bot has entered a repeat cycle.
if repeat > 0:

    # Reduce the repeat counter because a repeat will now take place.
    repeat -= 1

    # The history needs to be reset to ensure that repeats cannot be continuously triggered.
    history = [ "X" ]*length

else:

    # Play randomly while storing the value played and without repeating a move.
    if previous == "R":
        previous = random.choice(["P", "S"])
    elif previous == "P":
        previous = random.choice(["R", "S"])
    else:
        previous = random.choice(["R", "P"])

# Play the selected object.
output = previous
```