American University of Sharjah
School of Engineering
Computer Science & Engineering Dpt.
P. O. Box 26666
Sharjah, UAE

Instructor: Dr. Michel Pasquier
Office: ESB-2057
Phone: 971-6-515-2883
E-mail: mpasquier@aus.edu
Semester: Spring 2022

CMP 333 ARTIFICAL INTELLIGENCE

PROJECT #1

Due date/time: Tuesday 5 April 2022, 11 pm

Instructions: Submit to *iLearn,* before or on the due date, a single *Zip/Rar* archive that contains all your *code* files and output *results,* as described hereafter. Make sure to include the *name and ID* of all team members inside all your files and in the name of the archive itself, as per the following format: "CMP333-PR1-MPasquier75321-IZualkernan70594.zip". Follow any further instructions as may be posted on *iLearn.* Late submissions will be penalized as per the course policy.

Note that you are to complete this assignment *as a team of three students* exactly. Furthermore, each team must *work independently* and hand in their own original answers. You are *not* allowed to discuss or *share* any solution or to *copy* from others or from any sourced material. Plagiarism and cheating will be severely penalized, starting with a zero grade for the assignment. Recall you are bound by the AUS Academic Integrity Code which you signed when joining AUS.

Assignment: In this first AI course project, you are to program and solve several puzzles and problems using *search algorithms* and *heuristics.* Some of the selected problems were formulated earlier in class or homework/quiz assignments, and some are new.

The given Python code provides an implementation of the *General Search Algorithm* and variants, with some sample problem formulations such as for the 8-puzzle and Pacman. Note that the approach followed is exactly that in our AIMA textbook.

Why in Python? Firstly, by now all of you know Python to some extent. Secondly, this is *not* CMP 321 and the language is *just a tool,* so its usage is very basic. Thirdly, and most importantly, the code reads almost like *pseudo-code,* like in the book!

Getting ready:

Download the "Project1Search.zip" file from iLearn and unpack it where you wish. The resulting folder contains the following Python files:

"`AI_search.py`" contains a collection of search algorithms i.e., `generalSearch()`, `depthFirstSearch()`, `breadthFirstSearch()`, `iterativeDeepeningSearch()`, `uniformCostSearch()`, `greedySearch()`, and `astarSearch()`, as well as the supporting data structures i.e., `Stack`, `Queue`, and `PriorityQueue` classes.

Make sure you understand the code. You should not modify the search logic, but later you may need to uncomment and/or modify part of the code to *show the solution* and *performance details,* as elaborated hereafter. Whatever modification you make, clearly add comments as and where appropriate; each comment line should start with the two `#!` characters (for easy identification).

"`AI_heuristics.py`" contains some examples of *heuristic functions* typical of the 8-puzzle i.e., `hammingDistance()` and `manhattanDistance()`. You will need to modify these and/or add *other heuristic functions* later, in order to solve each specific problem in Part 2. Again, whatever you do, clearly add comments as stated above.

"`AI_problem.py`" contains an *abstract* `SearchProblem` class, with the following methods: `__init__()` that defines the attributes each state consists of; `getStartState()` that returns the initial state of the problem; `isGoalState()` that returns true if a state is a goal, and false otherwise; finally, `getSuccessors()` that returns all the states that can be reached from a given state. This last function will implement the available actions, with their preconditions and effects.

"`EightPuzzleProblem.py`" and "`PacmanProblem.py`" contain *concrete classes* by the same name that extend the `SearchProblem` class and override all four methods above. You should study the code to understand well the problem formulations and how they are implemented i.e., states, actions, cost/s, and heuristic functions.

"`AI_solve.py`" contains the `solve()` function that takes as arguments a *problem* (an instance of a concrete subclass of `SearchProblem`) and a *list of search algorithms* (as available from "AI_search.py"), and applies each algorithm in sequence to solve the given problem. By default, the function prints: 1. the name of the problem, 2. the name of the search algorithm employed, 3. the name of the heuristics used (if any), and 4. the details of the solution found or else "no solution".

Details include the cost of the solution (e.g., number of steps or distance), the number of nodes expanded (or number of iterations) and the number of nodes generated (in the search tree). Feel free to modify the code to print other information, change the format, etc. as you see fit. Make sure you always comment your edits, as stipulated earlier. Note that the printed solution path should be *understandable* (!) i.e., showing meaningful state information and not just abstract names or numbers.

<u>8-puzzle and Pacman</u>:

Apply the `solve()` function to solve the 8-puzzle ("`EightPuzzleProblem.py`") shown below, then to a another (non trivial) 8-puzzle of your choosing. Similarly, solve the Pacman game ("`PacmanProblem.py`") given below, then another one using a different Pacman map of your choosing.

```
puzzle = [1,8,0,
          4,3,2,
          5,7,6]
```

```
pacmap = ["P---------",
          "%%-%%-%-%%",
          "---%--%---",
          "-%%%-%%%-%",
          "---%%%-.-%",
          "-%------%%"]
```

Note that your submission must include for each problem (1) a *full trace* of your program in a plain text file, with *all required details* as explained earlier, as well as (2) a *summary table* in a Word document, such as the table hereafter. Make sure you use and report the results of *all applicable algorithms.* If you find that one algorithm is not applicable or not working, omit it but explain clearly *why* that is the case.

Search Algorithm	Nodes Expanded	Nodes Generated	Solution Cost
Depth First Search	597	1065	581
Breadth First Search	112455	134446	23
Iterative Deepening Search	2180357	2180365	23
Uniform Cost Search	112455	134446	23
Greedy Search w/ Manhattan Distance	77	136	43
Greedy Search w/ Hamming Distance	350	578	61
A* w/ Manhattan Distance	244	392	23
A* w/ Hamming Distance	12373	18942	23

Stone puzzle, Farmer problem, TSP:

Referring to the problems of Homework 2, for which you already have a formulation, create three files called "`StonePuzzleProblem.py`", "`FarmerPuzzleProblem.py`", and "`TravellingSalesmanProblem.py`", which model appropriately each problem by the same name i.e., creating for each a concrete subclass of `SearchProblem`.

Amend then apply your `solve()` function to each and show the detailed results and summary table, similarly to what you did with the 8-puzzle and Pacman problem examples in the previous section. Make sure to include all applicable algorithms and to define adequate cost and heuristic functions (with comments/explanations).

Sokoban sliding puzzle/game:

Sokoban (倉庫番) is a classic sliding puzzle in which the player pushes boxes around in a warehouse, trying to get them to designated storage locations. Your task is to devise then implement a formulation of this problem, then solve it like you did with the previous puzzles. Boards of various difficulty are provided (on iLearn). Below are three examples that are (from left to right) easy, moderate, and challenging.

```
#######          ######          ####
# .   #          #@   ##         #  ####
# # # #          # $$  #         # . . #
# # # #          # #. .#         # $$#@#
# $@  #          #     #         ##    #
#     #          #######          ######
#######
```

The game is played on a board of squares. Each square is a floor (' ') or a wall ('#'); some squares contain boxes ('$') and some are marked as storage locations ('.'). The player ('@') moves horizontally or vertically onto empty squares (never through walls or boxes). The player can move a box by walking next to it and pushing it to the square beyond. Boxes cannot be pulled, and they cannot be pushed to squares with walls or other boxes. The number of boxes equals the number of storage locations. The puzzle is solved when all boxes are placed at storage locations.

Create a file called "`SokobanPuzzleProblem.py`", which models appropriately the Sokoban puzzle as a concrete subclass of `SearchProblem`, with adequate cost and heuristic functions. The board should be provided as parameter i.e., a list of strings.

Amend then apply your `solve()` function again to solve 3 sample puzzles: 1 easy, 1 moderate, and 1 hard (pick any you wish among the 10 provided). Submit your *code,* show the *detailed results* and give a *summary table,* similarly to what you did earlier with all the class examples. *Comment* on the results. And then… have fun!