Assignment 3: Automated Puzzle Solving

Due on Friday, August 13, 2021, before 11:59 pm sharp, Toronto time

You may complete this assignment individually or with a partner who can be from any section of the course. You can search for teammates on Piazza using the following post.

Learning goals

By the end of this assignment, you should be able to:

- read complex code you didn't write and understand its design and implementation, including:
  - reading the class and method docstrings carefully (including attributes, representation invariants, preconditions, etc.)
  - understanding relationships between classes, by applying your knowledge of composition and inheritance
- complete a partial implementation of a class, including:
  - reading the representation invariants to enforce important facts about implementation decisions
  - reading the preconditions to factor in assumptions that they permit
  - writing the required methods according to their docstrings
  - use inheritance to define a subclass of an abstract parent class
- implementing algorithms by translating their steps into python code by:
  - choosing an appropriate ADT to solve the problem
  - deciding if recursion is appropriate
- implement recursive methods for trees when provided with their specifications.
- perform unit testing on a program to verify correctness

And more specifically:

- implement depth and breadth first searches to solve logic puzzles
- implement a simple expression tree class
- be able to implement additional logic puzzles beyond those in this assignment
- explore the behaviour of depth and breadth first solvers on various logic puzzles

Coding Guidelines

These guidelines are designed to help you write well-designed code that will adhere to the interfaces we have defined.

You must:

- write each method in such a way that the docstrings you have been given in the starter code accurately describe the body of the method.
- avoid writing duplicate code.

You must **NOT**:

- change the parameters, parameter type annotations, or return types in any of the methods or functions you have been given in the starter code.
- add or remove any parameters in any of the methods you have been given in the starter code.
- change the type annotations of any public or private attributes you have been given in the starter code.
- create any new public attributes.
- create any new public methods or functions.
- write a method or function that mutates an object if the docstring doesn't say that it will be mutated.
- add any more import statements to your code, except for imports from the `typing` module and the modules you're implementing in this assignment.

You may find it helpful to:

- create new private helper methods or functions for the classes you have been given.
  - if you create new private methods or functions you must provide type annotations for every parameter and return value. You must also write a full docstring for each method as described in the function design recipe
- create new private attributes for the classes you have been given.
  - if you create new private attributes you must give them a type annotation and include a description of them in the class's docstring as described in the class design recipe
- import more types from the `typing` module

All code that you write should follow the function and class design recipes.

While writing your code, you can assume that all arguments passed to the methods and functions you have been given in the starter code will respect the preconditions and type annotations outlined in the provided docstrings.

Introduction: Problem Domain

Puzzle solving is a typically human activity that, in recent decades, has been explored in the context of computers. There are many potential benefits to this. For example, we can offload some puzzle-solving tasks to computers, we may better understand human puzzle-solving by programming computer puzzle solvers, or we might generate novel puzzles using a computer program.

This assignment investigates a class of puzzles that have the following features in common:

feature	description
full information	all information about the puzzle state at any given point is visible to the solver; there are no hidden or random aspects
well-defined extensions	a definition of legal extensions from a given puzzle state to new states is given
well-defined solution	a definition of what it means for the puzzle to be in a solved state is given

These features are common to a very large class of puzzles: crosswords, sudoku, peg solitaire, verbal arithmetic, and so on. This assignment generalizes the required features into an abstract superclass `Puzzle` and solving such puzzles is written in terms of this abstract class.

Once this is done, particular concrete puzzles can be modelled as subclasses of `Puzzle` and solved using the solve method of a subclass of the `Solver` abstract class, which will be described later.

Although there may be faster puzzle-specific solvers for a particular puzzle by knowing specific features of that puzzle, the general solvers are designed to work for **all** puzzles of this sort.

Starter Code

Download the starter code a3.zip. Unzip the file and place its contents in PyCharm in your `a3` folder (remember to set `a3` folder as a source roor).

Any files that start with `a3_` are files that you will need to modify and submit. Once you are done Tasks 1-5, you will be able to run the files `play_sudoku.py`. `play_expression_tree.py`, `play_word_ladder.py` and `experiment.py`. You are provided with a basic set of starter tests, `starter_tests_a3.py`, to run locally, and a moderate set of tests on MarkUs. You are expected to write your own test cases to thoroughly test your code, as we have additional hidden tests you will be evaluated on.

Task 4 of the assignment requires you to install two python packages in order for your code to run (you don't need to write any code using them, but the code we have provided makes use of them): `networkx` and `pygame-gui`.

Please follow the instructions at the bottom of the (Software Guide's Installing Python libraries section)[https://q.utoronto.ca/courses/219921/pages/software-guide] to install these two packages. Ask on Piazza or visit office hours if you need help with this.

The Puzzles

We will start by introducing how the puzzles will be represented and what specific puzzles we will consider in this assignment.

abstract `Puzzle` class

The abstract class `Puzzle` has the following methods:

method	description
`is_solved` (abstract)	returns `True` iff the puzzle is in a solved state
`extensions` (abstract)	returns a list of extensions from the current puzzle state to new states that are a single 'move' away
`fail_fast` (has a default implementation in `Puzzle`)	returns `True` if it is clear that the puzzle can never, through a sequence of extensions, move into a solved state

And that's it! Note: each subclass will need its own `__init__` method in order to represent that particular puzzle's state information.

Sudoku

This puzzle commonly appears in print media and online. You are presented with an $n \times n$ grid with some symbols, for example digits or letters, filled in. The symbols must be from a set of n symbols. The goal is to fill in the remaining symbols in such a way that each row, column, and $\sqrt{n} \times \sqrt{n}$ subsquare, contains each symbol exactly once. In order for all of that to make sense, n must be a square integer such as 4, 9, 16, or 25.

You may want to read more about sudoku to get a feel for the puzzle if you aren't already familiar with it.

Word Ladder

This puzzle involves transforming one word into a target word by changing one letter at a time. Each word must belong to a specified set of valid words. Here's an example, where we assume that the set of words is a rather large set of common English words.

Here is one potential sequence of words, where the goal is to get from the word 'cost' to the word 'save':

$$cost \rightarrow cast \rightarrow case \rightarrow cave \rightarrow save$$

Expression Tree Puzzle

This puzzle consists of an algebraic equation containing one or more variables. The puzzle is solved when the variables are assigned values that satisfy the equation.

The Solvers

Now that we know a bit about the puzzles that we will be implementing, we will turn our attention to how we will implement the solvers.

Solving a puzzle can be done by systematically searching for a solution, starting from its current state. To make this daunting task even possible, we have to be sure that we have a systematic way of exploring **all** possible puzzle states - without needlessly re-visiting the same state twice.

You will implement two standard systematic searching techniques.

The implicit tree underlying our search

To understand our searching techniques, it helps to think of the tree that is defined by all the possible states of a puzzle: each node is one puzzle state, the root is the initial state of the puzzle, and the children of a node are its extensions (puzzle states that are one move away).

When we actually implement the algorithms, we don't need to explicitly form this tree, but it is useful to remember that it is there. The A3: Search Algorithms Visual Guide examples for the Word Ladder do a good job helping us visualize this tree.)

The search algorithms

With **depth-first search**, we search deeply before we search broadly. We exhaustively search the first subtree before considering any other subtree. And we use the same strategy when we search that subtree: exhaustively searching its first subtree before considering any other subtree. And so on - think recursively!

With **breadth-first search**, we search broadly before we search deeply. We consider all puzzle states at depth 1, then all puzzle states at depth 2, and so on until we have searched all puzzle states in our tree. (You will find a queue is helpful for keeping track of puzzles states to be checked when their turn comes.) Because we consider states that are "closer" to the starting state before those that are "farther", we are guaranteed to find the shortest path to a puzzle's solution.

For both solvers, we run the risk of encountering a puzzle state that we've already seen, and which already failed to produce a solution. To avoid exploring that state all over again, we will keep track of states we've seen before and just ignore them if we encounter them again.

For certain puzzle types, we might also be able to check whether we can quickly tell if a puzzle state is unsolvable. Such a check can be incorporated into our search algorithms and you will do so in your implementation.

Rather than spell out the algorithms in full detail and simply have you translate them into code, we have put together some worked examples - one of your tasks in this assignment will be to turn the above high level descriptions and the concrete examples into 2 algorithms you can implement in Python!

The Solver Class

The abstract class `Solver` has the following methods:

method	description
`solve`	returns a *path* to a solution of the puzzle associated with the solver. This method is abstract and must be implemented in a subclass.

You will create two subclasses of the `Solver` class - `DfsSolver`, which uses the depth first search strategy in its implementation of `solve`, and `BfsSolver`, which uses the breadth first search strategy in its implementation of `solve`.

Tasks

We have broken down the assignment into several tasks. For the most part you, can tackle the assignment in any order that you want, but note that Task 5 requires Task 4 and part of Task 3 requires Task 2. Task 6 is optional, but is a great way to try out your code and see it in action!

After reading the handout, docstring examples, and comments in the starter code, please ask for clarification on Piazza if you still find any of the specifications below to be unclear.

Any official clarifications will be added to the Assignment 3 FAQ** on Piazza.**

Task 1: `a3_sudoku.py`

We provide you with a mostly-implemented subclass `SudokuPuzzle` of class `Puzzle`.

1. Read and understand the provided code in `puzzle.py` for the abstract `Puzzle` class.
2. Read through the `SudokuPuzzle` class. Understand how the puzzle is represented, how `extensions` is implemented, and familiarize yourself with the provided helper methods. Note that some parts of the code use list comprehensions, as well as the functions `any` and `all`. You do not need to use list comprehensions, but you may find them helpful in making your code simpler in the next step. These are covered in a worksheet on list comprehensions for those looking for a quick reference.
3. Implement `fail_fast` for the `SudokuPuzzle` class by having it return True for any sudoku where there is at least one empty position that will be impossible to fill in because the set of symbols has been completely used up by other positions in the same row, column or subsquare. You should run the provided doctests to test your code, but you are encouraged to add more doctests to `fail_fast` (or write pytests) for further testing.

Task 2: `a3_solver.py`

1. Read the dosctrings provided in `a3_solver.py` and familiarize yourself with the interface that the `Solver` class provides.
2. Go through the A3: Search Algorithms Visual Guide (courtesy Sophia!) to make sure you understand how the depth first and breadth first search algorithms work.
3. Implement the `DfsSolver` and `BfsSolver` classes `solver.py`, based on the depth first search and breadth first search algorithms described earlier.
4. Many logic puzzles like Sudoku require their solution to be unique. Implement the `has_unique_solution` method in the `SudokuPuzzle` class, based on its docstring description.

You can test your solvers using the provided `SudokuPuzzle` class and also the other two puzzle types once you have implemented them.

Once you have completed Tasks 1 and 2, you can try running the provided `play_sudoku` module.

Task 3: `a3_word_ladder_puzzle.py`

1. Read and understand the provided code in `a3_word_ladder_puzzle.py`.
2. Override the `__str__` and `__eq__` methods of `WorldLadderPuzzle`.
3. Override `extensions`. A legal extension of a `WordLadderPuzzle` is a new puzzle state where the new `from_word` differs by a single letter from the previous `from_word`.
4. Override `is_solved`. The puzzle is solved when `from_word` is the same as `to_word`.
5. Implement method `get_difficulty` according to its docstring. (Requires Task 2)

Note: We do NOT require you to implement fail_fast for this puzzle type, as it isn't entirely obvious what it should look like! Of course, you are certainly welcome to attempt to devise a strategy to quickly check if a WordLadderPuzzle has no solution. Feel free to discuss your approach on Piazza if you come up with anything good!

Once you have completed Tasks 2 and 3, you can try running the provided `play_word_ladder` module.

Task 4: `a3_expression_tree.py`

Note: This part of the assignment requires you to install two python packages in order for your code to run (you don't need to write any code using them, but the code we have provided makes use of them): `networkx` and `pygame-gui` (you don't *need* `pygame-gui` except for in the optional Task 6, but you can install them both now).

Please follow the instructions at the bottom of the Software Guide's Installing Python libraries section to install these two packages. Ask on Piazza or visit office hours if you need help with this.

Before we can implement the `ExpressionTreePuzzle` class, we need to develop an `ExprTree` class to represent an Expression Tree.

1. Read and understand the provided code in `a3_expression_tree.py`.
2. Implement the folowing methods in the `ExprTree` class, as specified in the starter code:
   - `__str__`
   - `__eq__`
   - `populate_lookup`
   - `eval`
   - `substitute`

3. Implement the function `construct_from_list`

Expression Tree Format

In this assignment, an expression tree can contain the single digit constants 1-9, single letter variables containing only the letters a-z, and the operations: addition (`+`) and multiplication (`*`).

Numbers and variables can only occur as leaves of the tree.

Operators must have at least 2 children.

The same variable name may appear more than once in the expression tree.

Variables can be given single digit values 0-9 (As we will see, in the context of the `ExpressionTreePuzzle` class, the value 0 corresponds to an "unassigned variable")

A Convenient Way to Construct Expression Trees

It is cumbersome to create a substantial expression tree using the initializer from class `ExprTree`. Function `construct_from_list` allows client code to create one from a list of lists instead. Here is a simple example:

```
[['+'], [3, '*', 'a', '+'], ['a', 'b'], [5, 'c']]
```

The first `'+'` will be the root of the tree.

The next list contains its children, `3`, `'*'`, `'a'`, and `'+'`.

The next list contains the children of `'*'`: `'a'` and `'b'`.

And the last list contains the children of the second `'+'`: `5` and `'c'`.

Hint: A queue can help keep track of whose children the *next* list contains.

Visually, the resulting expression tree looks like this (note we use the $\times$ symbol to denote multiplication in the diagram):

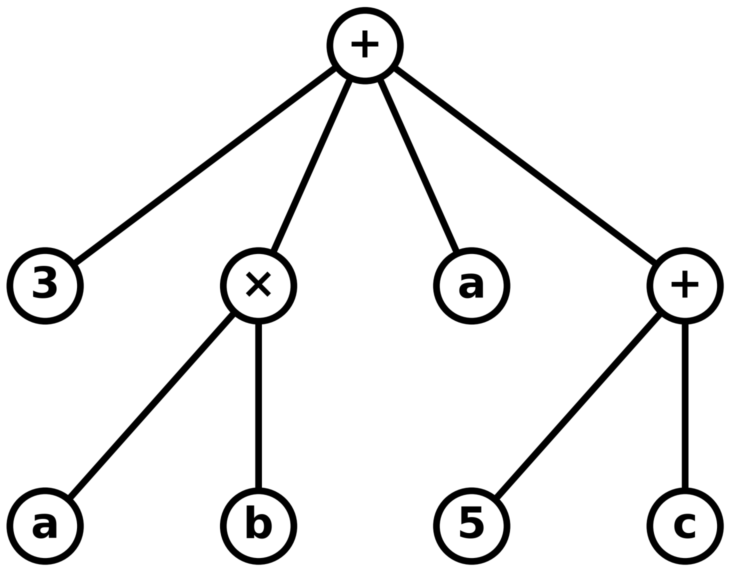
This expression tree's string representation would be: `(3 + (a * b) + a + (5 + c))`

Evaluating an `ExprTree`

In order to evaluate an expression tree, a variable lookup dictionary must be provided (see the `eval` method). All variables must appear in the lookup dictionary.

For our example above, if the lookup dictionary was `{'a': 0, 'b': 0, 'c': 0}`, then the expression tree would evaluate to `8`.

If the lookup dictionary was instead `{'a': 2, 'b': 5, 'c': 6}`, then the expression tree would evaluate to `26`.



Task 5: `a3_expression_tree_puzzle`

Once your `ExprTree` class is complete, you can implement the `ExpressionTreePuzzle` class in `a3_expression_tree_puzzle.py`.

1. Read and understand the provided code in `a3_expression_tree_puzzle.py`.
2. Override the `__str__` method for the `ExpressionTreePuzzle` subclass in `a3_expression_tree_puzzle.py`.
3. Override `extensions`. Legal extensions of a state assign each unassigned variable (with value 0) a value from 1-9. For example, if the variables dictionary was formerly `{'v': 0}`, then each extension would assign a different value from 1-9 to `'v'`. If the variables dictionary was formerly `{'v': 5}`, then the puzzle would have no extensions, as all variables have been assigned a non-zero value.
4. Override `is_solved` as specified in the code.
5. Implement `fail_fast` for the `ExpressionTreePuzzle` class. The exact implementation is up to you, but we have provided a couple of hints in the code if you aren't sure how to approach this.

Task 6: Try out your code!

Once you have completed the first 5 tasks you will be able to run:

- `play_sudoku.py` (requires Tasks 1 and 2)
- `play_word_ladder.py` (requires Tasks 2 and 3)
- `play_expression_tree.py` (requires Tasks 2, 4, and 5)
- `experiment.py` (requires Tasks 2 and 3)

There are no marks for this part, but it can be a fun way to debug your code and further explore what you can do with the code you have written.

You are free to modify or add to any of the code provided in these 3 modules.

Module `play_sudoku.py`

This module contains a simple GUI to let you try playing the sudoku puzzle you implemented. It makes use of your solvers in two ways: (1) to create a random SudokuPuzzle of a selected difficulty level and (2) to provide you with hints as you try solving the puzzle. The code also has a nice application of inheritance to let us add randomness to the puzzles we generate.

Module `play_word_ladder.py`

This module contains a simple text UI to let you try playing the word ladder puzzle you implemented. It makes use of your solvers in two ways: (1) to create a random `WordLadderPuzzle` of a selected difficulty level and (2) to provide you with hints if you get stuck solving the puzzle.

Module `play_expression_tree.py`

This module contains a simple GUI to let you try playing the expression tree puzzle you implemented. It makes use of your solvers to provide you with hints.

Module `experiment.py`

This module runs a small experiment and produces output that you can review to verify your understanding of how the solvers perform on different puzzles. Feel free to add your own experiments and share any interesting observations you make with the class!

Sample Output:

Puzzle Type	Solver	len(sol)	time
Sudoku	Bfs	13	0.01907
Sudoku	Dfs	13	0.00139
Sudoku	Bfs	54	0.43171
Sudoku	Dfs	54	0.15584
WordLadder	Bfs	5	0.03384
WordLadder	Dfs	703	0.04225

Polish!

Take some time to polish up. This step will improve your mark, but it also feels so good. Here are some things you can do:

- In each module you are submitting, run the provided `python_ta.check_all()` code to

check for errors and violations of the "PEP8" Python style guidelines. **Fix them!**
- Check your docstrings to make sure that they are precise, complete, and that they follow the conventions of the Function and Class Design Recipes.
- Read through and polish your internal comments.
- Remove any code you added just for debugging, such as print statements.
- Remove the word "TODO" wherever you have completed the task.
- Take pride in your gorgeous code!

submitting your work

Submit the following files on MarkUs

`a3_solver.py`

`a3_sudoku_puzzle.py`

`a3_word_ladder_puzzle.py`

`a3_expression_tree.py`

`a3_expression_tree_puzzle.py`

We strongly recommend that you submit early and run the provided self tests on MarkUs as you go.

We will grade the latest version you submit within the permitted submission period.

Be sure to run the tests we've provided within MarkUs one last time before the due date. This will make sure that you didn't accidentally submit the wrong version of your code, or worse yet, the starter code!

How your assignment will be marked

There will be no marks associated with defining your own test cases with pytest or hypothesis.

The marking scheme will be approximately as follows:

Category	Weight
pyTA	10 marks
MarkUs self tests	20 marks
hidden tests	70 marks