CSC148, Assignment #2
automated puzzle solving
due March 24th, 10 p.m.

deadline to declare your assignment team: March 18th, 10 p.m.

overview

Puzzle solving is a typically human activity that, in recent decades, is being explored in the context of computers. There are two benefits to this: first, we can off-load some puzzle-solving tasks to computers, and second, we may understand human puzzle-solving better by studying how to implement it on a computer.

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

full information: all information about the puzzle configuration 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 configuration to new configurations is given

well-defined solution: a definition of what it means for the puzzled 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 functions are 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 solving functions. 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.

abstract Puzzle class

The abstract class `Puzzle` has the following methods, which are not implemented, but meant to be implemented in subclasses:

is_solved: returns `True` or `False`, depending on whether the puzzle is in a solved configuration or not

extensions: returns a list of extensions from the current puzzle configuration to new configurations that are a single step ("move") away

fail_fast: returns `True` iff it is clear that the current puzzle configuration can never be completed, through a sequence of extensions, to a solved state

And that's it. Notice that there is no `__init__` method, since the representations and initializations of puzzles in this class vary widely. Each subclass will need an `__init__` method to represent a particular puzzle.

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, or $\sqrt{n} \times \sqrt{n}$ subsquare, has each symbol exactly once. In order for all of that to make sense, n must be a square integer such as 4, 9, 16, 25, ...

You may want to read more about sudoku to get a feel for the puzzle.

We provide you with a mostly-implemented SudokuPuzzle subclass of `Puzzle`. We, however, did not override the `fail_fast` method inherited from `Puzzle`, and you should be able to do better with a Sudoku-specific `fail_fast` implementation.

Implement `fail_fast` for sudoku by having it return `True` for any sudoku configuration where there is at least one empty position that will be impossible to fill because the set of symbols has been completely used up by other positions in the same row, column or subsquare. Experiment to see whether this makes any difference in the performance of solver(s) for sudoku.

peg solitaire

This puzzle has gone by various names: Hi-Q, peg solitaire, solo noble, and patience, and has been around for several hundred years.

You are presented with a grid where most positions are occupied by a peg, marble, or other movable piece. The object is to "jump" a piece over a neighbouring piece and into an empty position, allowing you to remove the piece that was jumped over. A successful game leaves only one piece on the board.

Boards come in various shapes of grid. We will assume that some portion of an $m \times n$ grid is used, that some positions may be unused (no pieces may be placed there), some positions begin with pieces in them, and that a few (often just one) position is left empty to allow jumping. You can see various versions of peg solitaire on the web.

We provide you with the class declaration and `__init__` method for subclass GridPegSolitairePuzzle of `Puzzle`.

You will override the `extensions` and `is_solved` methods for this subclass, which make it possible for the solvers (which you'll also write) to solve it. A legal extension of a `GridPegSolitairePuzzle` is a new configuration that you can get to in a single jump. The puzzle is solved when a single peg is left.

We do not require you to override the `fail_fast` method, although you are welcome to see whether you can come up with one that improves performance.

word ladder

This puzzle involves starting with `from_word`, from a set of allowed words, and deriving `to_word` from the same set of allowed words, by changing one letter at a time. Each changed letter must result in a word that is from the same set of words. Here's an example, assuming that the set of words is a rather large set of English words provided to many computer applications. The goal is to get from **cost** to **save**

$$\textbf{cost} \to \textbf{cast} \to \textbf{case} \to \textbf{cave} \to \textbf{save}$$

We provide you with the class declaration and `__init__` method for subclass WordLadderPuzzle of `Puzzle`.

You will override the `extensions` and `is_solved` methods for this subclass. A legal extension of a `WordLadderPuzzle` is a new configuration where the new `from_word` differs by a single letter from the previous `from_word`. A `WordLadderPuzzle` is solved when the `from_word` is the same as the `to_word`.

We don't require you to override the `fail_fast` method, although you are welcome to see whether you can come up with one that improves performance.

mn puzzle

Sometimes called the 15-Puzzle, Boss Puzzle, or Game of 15, this puzzle consists of a 4×4, 3×3, or (in general) an $m \times n$, grid with one empty position. A player must swap symbols with the empty position until they get the entire puzzle into the correct order.

We provide the class declaration and `__init__` method. You will override `is_solved` and `extensions` methods to allow the solvers to solve this puzzle. The puzzle is solved when the configuration is the same as the supplied target configuration. Legal extensions of a configuration swap the empty space with either the symbol to its right, the symbol to its left, the symbol above it, or the symbol below it.

We don't require you to override the `fail_fast` method, but you are welcome to see whether you can design one that improves performance.

searching

One approach to solving a puzzle is to systematically search for a solution, starting from the current configuration. To make this daunting task even possible, we have to be sure that we have a systematic way of checking out puzzle configurations, and that we don't re-visit the same configuration twice.

You will implement two standard systematic searching techniques. We are well aware that plenty of code and pseudo-code for these are "out there" on the web, so you have to be very careful you are giving credit for any ideas you recycle, in order to avoid an academic offense.

depth-first search

As the name suggests, in depth-first search you search deeply before you search broadly.

1. don't bother with any puzzle configuration if it's been seen before
2. check whether the current puzzle configuration is solved. If so, you're done; otherwise...
3. for each extension of the current puzzle configuration, make that extension the current puzzle configuration and return to step 1

Notice how this process unwraps. You check a configuration, then **one of its extensions**, then **one of the extension's extensions**, and so on. That means that if a configuration has several extensions, you will be checking every possible chain of extensions from one of them **before** you check the others.

Notice also that if there are no extensions, step 3 explores no new extensions.

You will implement depth_first_solve following this strategy. You are welcome to read material on depth-first search, but be aware of several things:

- the return type for `depth_first_solve` is a `PuzzleNode`[1] that must be the first node in a path of `PuzzleNode`s that lead from the given configuration to a solution
- give credit for ideas from the web that you recycle, so you won't be guilty of an academic offense.

[1] **PuzzleNodes** are defined in puzzle_tools.py

breadth-first search

As the name suggests, in breadth-first search you search broadly before you search deeply.

1. skip any puzzle configuration that has already been seen
2. check whether the current puzzle configuration is solved. If so, you're done; otherwise...
3. check whether any extensions of the current puzzle configuration are solved; if so, you're done; otherwise...
4. check whether any extensions of extensions of the current puzzle configuration are solved; if so, you're done; otherwise...
5. ...

Notice how this approach examines configurations that are "closer" to the starting configuration before it examines those that are "farther." This may be useful for finding a shortest solution to a puzzle.

You will implement breadth_first_solve following this strategy. You are welcome to read material on breadth-first search, but be warned:

- the return type for `breadth_first_solve` is a `PuzzleNode` that is the first node in a path of `PuzzleNodes` from the current puzzle configuration to a solution
- give credit for ideas from the web that you recycle, so you won't be guilty of an academic offense

starter code

You'll find the following files, with associated tasks, in Starter. Leave the class declarations, `__init__` methods, and the `if __name__ == "__main__":` block as you find them: we have verified that neither PyCharm[2] nor PEP8 disapprove of them.

puzzle.py Read and understand the `Puzzle` class.

sudoku_puzzle.py Read and understand the `SudokuPuzzle` class, then override the `fail_fast` method inherited from `Puzzle`.

grid_peg_solitaire_puzzle.py Read the `__init__` method, then override the `extensions` and `is_solved` methods inherited from `Puzzle`.

word_ladder_puzzle.py Read the `__init__` method, then override the `extensions` and `is_solved` methods inherited from `Puzzle`.

mn_puzzle.py Read the `__init__` method, then override the `extensions` and `is_solved` methods inherited from `Puzzle`.

puzzle_tools.py Read and understand the `PuzzleNode` class, then implement module-level functions `depth_first_solve` and `breadth_first_solve`.

[2] Well, at least, PyCharm Community Edition 5.0.4

declaring your assignment team

You may do this assignment alone or with one other student. Your partner(s) may be from any section of the course at UTM. You must declare your team (even if you are working solo) using the MarkUs online system.

Navigate to the MarkUs page for the assignment and find "Group Information". If you are working solo, click "work individually". If you are working with another student:

First: one of you needs to "invite" the other to be partners, providing MarkUs with their utorid.

Second: the invited student must accept the invitation.

Important: there must be **only** one inviter, and the other group member accepts **after** being invited.

To accept an invitation, find "Group Information" on the Assignment page, find the invitation listed there, and click on "Join".

submitting your work

Submit the following files on MarkUs by 10 p.m. March 24th:

- `sudoku_puzzle.py`
- `grid_peg_solitaire_puzzle.py`
- `word_ladder_puzzle.py`
- `mn_puzzle.py`
- `puzzle_tools.py`

Click on the "Submissions" tab near the top. Click "Add a New File" and either type a file name or use the "Browse" button to choose one. Then click "Submit". You can submit a new version of a file later (before the deadline, of course); look in the "Replace" column.

Only one team member should submit the assignment. Because you declared your team, all of you will get credit for the work.