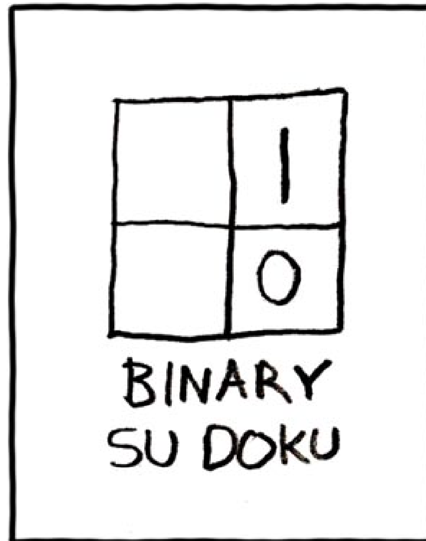
CS51A - Assignment 10

Due Sunday, April 16 at 11:59pm



https://xkcd.com/74/

For this assignment, you will be using best first search to find solutions to sudoku problems! As always (and particularly for this assignment), read through the entire handout before you start working.

For this assignment, you may (and in fact, I'd encourage you to) work with a partner. If you work with a partner, you only need to submit one file, just make sure both of your names are at the top of it. **If you choose to do work with a partner, you must both be there whenever you're working on the assignment and you should work on each of the problems collaboratively.**

Starter code

I have provided you with some code to get you started this time that includes the search method implementation, functions for generating sudoku problems (three different ones) and also a few methods in the class.

To get started, unpack the assignment 10 starter code into a new PyCharm project as usual:

`https://cs.pomona.edu/classes/cs51a/assignments/assign10-starter.zip`

This `assign10.py` file has a number of things in it that will be useful:

- At the top, I've started the class definition already for you called `SudokuState`, which you'll be filling in.
- I've written four methods for you in this class:
  - `get_subgrid_number`: The sudoku board is a 9 by 9 grid that is subdivided into nine 3 by 3 subgrids. This method takes a row and column for an entry in the 9 by 9 grid and returns a number 1 through 9 indicating which subgrid that row/column entry falls into, where subgrids are numbered as follows:
  - `get_any_available_cell`: See the experiment section for how this will be used.
  - `__str__`: Generates a "pretty" version of the state and board as a string. This looks nice and can be useful for debugging, but it hides information about what's in the cells that don't have numbers in them.
  - `get_raw_string`: Generates a more complete version of the state and board as a string. It does include the cells that haven't been filled in and will also be useful for debugging, though it prints a lot of information.
- I've included a helper class called `SudokuEntry`. Each entry on the sudoku board will need to keep track of whether or not a number has been placed or not as well as what numbers are still valid for this entry. The `SudokuEntry` class manages this. Here are some of the most important methods for the class:
  - `__init__`: Construct a new empty entry with all possible numbers available (1-9).
  - `is_fixed`: Returns `True` if a number has been placed in this entry or `False` if it's still open.
  - `values`: Returns a list of the numbers that are still available to be placed in this entry.
  - `fix(n)`: Puts the number `n` at this entry.
  - `eliminate(n)`: Remove `n` from the possible set of valid values that could be placed at this entry.

Here is a simple example of the class being used. Note that you'll be using it within your `SudokuState` methods.

```
>>> entry = SudokuEntry()
>>> entry.values()
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> entry.is_fixed()
False
>>> entry.eliminate(5)
>>> entry.values()
[1, 2, 3, 4, 6, 7, 8, 9]
>>> entry.eliminate(1)
>>> entry.values()
[2, 3, 4, 6, 7, 8, 9]
>>> entry.fix(3)
>>> entry.values()
[3]
>>> entry.is_fixed()
True
```

- I've included three functions for generating three different starting states: `problem1`, `problem2` and `heart` (the example from the class notes).
- I've also included a `dfs` implementation as well as commented code to illustrate how to call `dfs` to get a solution and then print it out.

`SudokuState` class requirements

Like any search problem, we will need our class to do three key things:

- Construct the starting state
- Determine if a state is a goal state
- From a state, determine what are the "next" states that we can get to by performing an action

Specification

To support this functionality, fill in the details for the `SudokuState` class. The class must have the following instance variables:

- `self.size`: the size of a side of the sudoku board (it will always be set to 9 for our implementation, though)

- `self.num_placed`: the number of *valid* numbers that have been put on the board. We are going to make sure that we only create states that do not violate the sudoku constraints.

- `self.board`: a list of lists (i.e. a matrix). Each entry is a SudokuEntry, which defines both a `domain` (a list of numbers 1-9) and a boolean indicating whether it is `fixed`.

Your class should have *at least* the following methods:

- `__init__`

  Should take zero parameters (besides `self`), and construct an empty sudoku board. An empty sudoku board should be a 9 by 9 matrix (list of lists) where each entry is an un-fixed `SudokuEntry` with the numbers 1 through 9. For example, the following lines would generate an empty board and print out the "pretty version" (using the `__str__` method, which doesn't print the board entries if there isn't a number there) as well as the "raw version" (using the `get_raw_string` method) showing the actual lists being stored in the matrix:

  ```
  empty_state = SudokuState()
  print(empty_state) # print the pretty version
  print(empty_state.get_raw_string()) # print the actual data
  ```

Pretty version:

```
num placed: 0
 -----------------------
| _ _ _ | _ _ _ | _ _ _ |
| _ _ _ | _ _ _ | _ _ _ |
| _ _ _ | _ _ _ | _ _ _ |
 -----------------------
| _ _ _ | _ _ _ | _ _ _ |
| _ _ _ | _ _ _ | _ _ _ |
| _ _ _ | _ _ _ | _ _ _ |
 -----------------------
| _ _ _ | _ _ _ | _ _ _ |
| _ _ _ | _ _ _ | _ _ _ |
| _ _ _ | _ _ _ | _ _ _ |
 -----------------------
```

Raw version showing the actual data:

```
num placed: 0
[[1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6,
[[1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6,
[[1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6,
[[1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6,
[[1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6,
[[1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6,
[[1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6,
[[1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6,
[[1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6,
```

Because of the size, even with a very small font, it doesn't fit on this paper, however, it is a 9 by 9 matrix (list of lists) where each entry is a list 1 through 9, since there are initially no numbers placed so every space could have every number.

Warning:
Be very careful about aliasing, that is, make sure that each entry in the board is it's own, *unique* entry. A good example to build off is the `zero_matrix` example we looked at in class which used nested `for` loops. By using nested for loops and generating each list of one through nine one at a time ensures that they are unique (you can easily generate a new list from one to nine using the `range` function).

In particular do **NOT** do something like:

```
one_to_nine_list = SudokuEntry()

for ...
   for ...
      ...append(one_to_nine_list)
```

instead, do something like:

```
for ...
   for ...
      ...append(SudokuEntry())
```

- `remove_conflict`

  Takes three parameters (agian, besides `self`, which we won't be counting from here on): a `row`, a `column` and a `number`. *If* the entry at that row and column hasn't been filled in yet, remove `number` from the list of possible values at this entry. This method will be called when a number has been placed in either the same row, column or subgrid as this entry and it will remove that number as a possibility.

  Hint: This one is pretty short! Look closely at the methods available on SudokuEntry.

- `remove_all_conflicts`

  Takes three parameters, a `row`, a `column`, and a `number`. This method is called after `number` has been placed at entry `row, column`. It should update all of the other unfilled entries that are in the same row, column or subgrid as this entry so that they no longer have `number` as an option.

  For example, if we put a 7 at (1, 2), then all unfilled entries in row 1 and column 2 and in the first subgrid would need to have 7 removed from their list of possible valid values.

  Hints:

  * Make sure that you understand what this method is doing. It's making sure that all entries that are affected when we put a number on the board are updated appropriately.
  * The `get_subgrid_number` and `remove_conflict` methods may be useful here.
  * Don't work too hard to be efficient for updating the subgrid. You can iterate through the entire board and for each position check if the subgrid number of that position is the same as the one you're updating. If it is, then you can remove the conflict.

- `add_number`

  Takes three parameters, a `row`, a `column` and a `number` and modifies the receiver by adding `number` at `row, column`. You will need to make sure that when you add that number the new state stays consistent by removing any conflicts.

  After you implement this method, you should be able to use any of the functions included in the starter code that generate starting boards (e.g., `heart()`).

  Hints:

  * Remember that any state that is an instance of a class (whether it be `self` or any other variable that references an object of a class) can call *any* of the methods of that class. This is an important method so make sure you test it thoroughly before moving on (the two "string" methods should be useful for testing this).
  * This shouldn't be a lot of code. `remove_all_conflicts` should be useful :)

- `get_most_constrained_cell`

  Takes zero parameters and returns a tuple containing the row and column of the most constrained entry in the board, i.e. the entry that is not filled in yet that has the fewest possible options remaining. Think of this function as our heuristic function that will help guide our search.

- `solution_is_possible`

  Takes zero parameters and returns `True` if all entries on the board still have at least one possible value that they can take, `False` otherwise (i.e. there is at least one entry without any possible values that it can take).

- `next_states`

  Takes zero parameters and returns a list of the next states that can be reached from this current state by trying to put a number *in the most constrained cell*. You should only return viable states in this list, i.e. states where a solution is still possible.

  Hint: To be able to tell if a state is viable, you'll need to first generate it and then check if a solution is still possible before adding it to the list of states. Note that this is a bit different from the n-queens problem where we mainly tested *from* the current state. This is another possible way of doing it.

- `is_goal`

  Takes zero parameters and returns whether or not this state is a goal state.

  Hint: Like the n-queens problem, this can be answered very easily based on information available in the state, in particular, you can do it without having to traverse all the entries in the board.

Where's best first search?

If you do everything correctly, when you've implemented these methods you should be able to uncomment the code at the bottom of your file and solve the sample problems. You may notice, however, that we're using the exact same `dfs` function that we used before. Why isn't this depth first search? The key is that the `next_states` method greedily chooses which entry to fill in based on the heuristic to fill in the most constrained entry first.

Experiments

Once you have things working, I want you to do a couple of experiments. I have included a method called `get_any_available_cell` that serves a similar functionality to `get_most_constrained_cell`, but it just chooses the first empty one it comes across. In your `next_states` method, you can change the code to use either of these methods for selecting the next entry and dfs should still find a solution. If you use the former, it performs a traditional, uninformed search, and if you use the latter, it performs a best first, informed search.

Experiment with these two variants on the three different example problems. Do you see a difference in performance (I've included the time it takes to solve the problem in the information printed out)?

As a second experiment, I've provided a method called `propagate` which works like the constraint propagation from our n-queens example in class. Insert a call to `propagate` within `next_states` just before checking if a solution is possible from each successor state (i.e., after calling `add_number`

on it). How does this influence the time to finding a solution for the heart puzzle, and why do you think that might be?

Include a short paragraph on your experiments in comments at the top of your file that talk about the results that you found and an explanation.

Hints/advice

- Make sure you understand the role that each of these methods serves. Although we only need the `is_goal` and `next_states` methods for `dfs` to work, all of the other methods I'm asking you to implement will be useful in implementing these methods or others.
- I strongly encourage you to implement your class a method at a time and then test each method individually. Since many of the methods will use previous methods, it's important that you're confident that the previous methods are correct. Here are a few examples of how you might test individual methods *as you go*, but I encourage you to also come up with your own tests:
  - `__init__`

    ```
    >>> s = SudokuState()
    >>> print(s)
    >>> print(s.get_raw_string())
    >>> s.board[0][0].fix(1)
    >>> print(s)
    ```

    The last statement is a good check to make sure you don't have aliasing.
  - `remove_all_conflicts`

    ```
    s = SudokuState()
    s.board[0][0].fix(1) # set a one in the upper left hand corner
    s.remove_all_conflicts(0, 0, 1)
    print(s)
    print(s.get_raw_string())
    ```
  - `add_number`

    ```
    s = SudokuState()
    s.add_number(0, 0, 1)
    print(s)
    print(s.get_raw_string()) # make sure s is the proper state
    ```
  - `next_states`

    ```
    s = SudokuState()
    print(s)
    next = s.next_states()
    ```

```
for sn in next:
   print(sn)
   print(sn.get_raw_string())
   # make sure every sn is different
print(s) # make sure s is the same as before
```

- I have tried to order the required methods in the order that I would suggest implementing them.
- If your code doesn't work, you need to figure out *why*. One of the best ways to do this is to put in print statements to print out how your code is working. Try to isolate particular functionalities and then make sure they're working like you expect. Make sure, however, to remove these print statements once you fix it!
- Start early! If you hit a problem, come talk to myself or the mentors (after, of course, you've tried to debug it on your own for a little bit).

When you're done

You should have a single .py file with

1) the `SudokuState` class,
2) the starter code at the bottom uncommented and
3) your experiment paragraph in comments at the top

Additionally, make sure that the file is properly commented:

- You should have comments at the very beginning of the file stating your name, course, assignment number and the date.
- Each function should have an appropriate docstring.
- Your class should have an appropriate docstring.
- Include other miscellaneous comments to make things clear.

Submit your .py file online using the courses submission mechanism. If you worked with a partner, make sure to tag them in the submission system.

Grading

	points
`init`	3
`remove_conflict`	2
`remove_all_conflicts`	5
`add_number`	2
`get_most_constrained_cell`	3
`next_states`	3
`is_goal`	2
`experiments`	3
comments/style	3
total	26