Project 1 – Crossword Solvability

Overview: In this project, you will be creating code to solve clueless crosswords.

Goal: To gain experience with search algorithms.

Requirements: You are going to be given a set of crossword files formatted like the one below:

```
..........
.******...
...*......
...*......
...*****..
...*..*...
......*...
......*...
......*...
..........
POLAND
LHASA
SPAIN
INDIA
```

The crossword file begins with a grid showing where the words go (represented by the * character) and the spaces around the words (represented by dots). Then there is a list of words, one per line, all in caps, that are to be placed into the crossword grid. The words should be placed in the grid so that they fit the number of asterisks and so that anywhere that words intersect each other, they share a letter. For example, with the puzzle above, the solution would be:

```
..........
.POLAND...
...H......
...A......
...SPAIN..
...A..N...
......D...
......I...
......A...
..........
```

The crossword grids can range in size, and there can be more than one space for a word in a single row or column. Some crosswords may have more than one possible solution. And some will be unsolvable, e.g.

```
****
*..*
****
```

```
*..*
ABAB
BABA
ACAC
BACA
```

In Python 3.x, write a program that will determine whether the crossword can be solved with the available words. An obvious choice would be an exhaustive search algorithm, but you have many options to consider, including algorithms not yet covered in class or the textbook.

Your code should output the solution to the crossword that it finds. If there is more than one solution, you can output them all or just one of them. In addition, your code should be able to track 1) how long in milliseconds it takes to find a solution to a particular crossword, and 2) how many basic operations are performed – you will need to include a README file or a block of comments at the top of your code explaining what you are counting as the basic operation.

The files will be stored in a folder called "puzzles", which you can assume will be placed in the same location as your source code, and each puzzle file will be named puzzle1.txt, puzzle2.txt, etc. You've been provided with a number of sample crosswords to practice on, but **you should make sure to create your own for testing purposes**. The grader will be using a different set of crosswords for testing. Your code should detect which crossword files are available in the puzzles sub-directory (python's `listdir` function might help) and repeatedly offer the user a menu allowing them to choose the file to run.

Algorithm:
In addition to your code, you need to turn in a well-formatted document containing the pseudo-code for your core algorithm (written in the same format as the textbook uses for pseudo-code) along with any proofs you decide to include about your algorithm – see grading section below.

Grading:
Your algorithm will be graded according to the rubric below.

	1	**2**	**3**	**4**
Solution	Code returns yes/no to indicate whether a solution to the puzzle exists	Code returns a valid solution if one exists, otherwise indicates that one doesn't exist.	Code returns a valid solution if one exists, otherwise indicates that one doesn't exist. Proof of completeness or optimality provided.	
Time	Code always terminates in finite time.	Runtime of code at least as good as clean implementation of DFS on test problems.	Runtime at least as good as benchmark code – sample runtimes to be provided.	Best time performance of all student submissions.
Analysis	Submits accurate pseudo-code of algorithm <u>or</u> implements code to	Submits accurate pseudo-code of algorithm <u>and</u> implements code to	Has pseudo-code and timing and counting code. Defines input size and basic	

	count basic operations and show runtime in msecs.	count basic operations and show runtime in msecs.	operation for domain. Formally proves that algorithm belongs to Θ(g(n)).	

Your project will receive a final score based on the number of points from the above rubric that it accrues. The table below summarizes the minimum number of points from the rubric that would be necessary to earn various grades. Extra credit is possible.

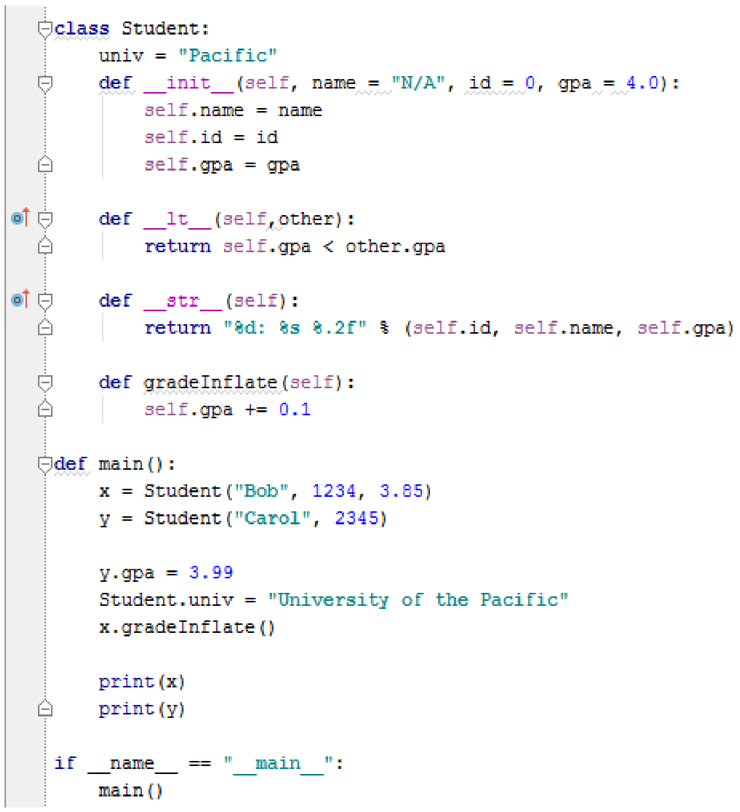
Grade:	50	47.5	42.5	40	35	30
Points on Rubric:	8	7	6	5	4	3

Runtime Guides: The following benchmarks can be used in determining whether your code is matching the performance of a clean DFS implementation and the benchmark algorithm. The benchmark times were based on running the code on one of the lab computers in CTC 114. The times are based on one run – if I get a chance, I'll provide minimum and averaged times because there is some variation in the runtime on the lab computers.

Puzzle	**Solvable?**	**DFS runtime (in secs)**	**Benchmark runtime (in secs)**
puzzle1.txt			
puzzle2.txt			
puzzle3.txt			
puzzle4.txt			
puzzle5.txt			
puzzle6.txt			
puzzle7.txt			
puzzle8.txt			

Submission Instructions: Please submit a .zip or .7z file containing the code for your project, the puzzles you tested on as well as your document containing the algorithm pseudo-code and any proofs.

Helpful Syntax: If you decide that you want to create a class for some portion of this project, I am including the following example code as a reference. Further details can be found at any of the python documentation webpages.



```python
class Student:
    univ = "Pacific"
    def __init__(self, name = "N/A", id = 0, gpa = 4.0):
        self.name = name
        self.id = id
        self.gpa = gpa

    def __lt__(self,other):
        return self.gpa < other.gpa

    def __str__(self):
        return "%d: %s %.2f" % (self.id, self.name, self.gpa)

    def gradeInflate(self):
        self.gpa += 0.1

def main():
    x = Student("Bob", 1234, 3.85)
    y = Student("Carol", 2345)

    y.gpa = 3.99
    Student.univ = "University of the Pacific"
    x.gradeInflate()

    print(x)
    print(y)

if __name__ == "__main__":
    main()
```

Member variables are created whenever you use `self.`*`varname`* in a member function, e.g. in the 3 lines of the `__init__` function. Variables that are declared outside of a member function, like `univ`, are shared between all the copies of an object (like a static variable in C++). So both of the Student objects, x and y, will share one semi-global copy of the `univ` variable and any change that one of them makes will be seen by the other one.

`__init__` is the constructor for the class. Like all member functions, its first parameter is the object itself. The following three parameters are used to pass in starting values, which init then copies into the corresponding member variables. One thing to watch out for, when using member variables, you must always use `self.`*`varname`*; if you forget the "self" then Python will create a separate local variable that happens to have the same name.

Notice that the parameters for `__init__` have been given default values (with the = symbol), which means that they are optional. The user can omit one or more of those parameter values (from right to left) as we did in main when declaring object y.

The function `gradeInflate` is an example of a regular member function. It can manipulate its own name, id and gpa member variables by modifying `self.varname`. It can modify the shared `univ` variable by using `Student.univ`.

In `main` you can see that member functions are called using the dot notation familiar from C++. Notice that member variables are not protected and can be modified from main as well.

The remaining two member functions, `__str__` and `__lt__`, are overloaded implementations of standard functions, which is why their names start and end with two underscores. `__str__` is called whenever you call "print" or "str" on an object and the job of `__str__` is to create a string representation of the data inside the object. `__lt__` is used to tell whether one object is less than the other, e.g. x < y. It is useful if you want to sort your objects because sorting algorithms often rely on less than comparisons.