Assignment 1

Assigned: 9/16/19
Due: 10/9/19

This assignment is intended to give you practice in implementing some of the concepts we have been discussing in class. You will be given code that implements the pseudocode in the textbook, and be asked to adapt it in a variety of ways.

Logistics

- This is an individual assignment. You may talk about it in general terms with your classmates, but your work should represent your individual work.
- Written portions of the assignment should be filled in as part of this document, and should be submitted on Gradescope. Your code should be submitted on CourseWeb
- The program you submit should be named: **last-name*-cs1571-a1.py* (**last-name**-cs1571-a1.py)
- Assume for all parts that the input is well-formed

Part A. Sudoku & Complexity (50 points)

1. Create a function named "sudokuSolver". This function should take as inputs:
   - A string representing a sudoku grid of two possible sizes: 2x2 (containing the digits 1 through 4) and 3x3 (containing the digits 1 through 9). Each grid is represented by a string where a digit denotes a filled cell and a '.' denotes an empty cell. The string is intended to fill in the Sudoku grid from left to right and top to bottom. For example, "...1.13..32.2..." is displayed as:

```
 . . | . 1
 . 1| 3 .
----+-----
 . 3| 2 .
 2 .| . .
```

   - A string representing one of the three algorithms that you are planning to run as input: "bfs", "dfs", or "backtracking".

   Run BFS (tree search), DFS (tree search), and backtracking on the three grids found in exampleSudokus-q1.txt. You should implement naïve versions of BFS and DFS, in that they can choose the variables to fill one by one, but should not use any heuristics to determine which numbers are legal to fill in. For backtracking, use minimum-remaining-values, least-constraining-value, and forward checking.

   With each Sudoku board, your program should output a number of different factors to a file:

   a. The solution to the puzzle as a String in the same format as the input string.
   b. Total number of nodes created (or in the case of backtracking, the number of assignments tried)
   c. The maximum number of nodes kept in memory at a time (ignore in the case of backtracking)
   d. The running time of the search, using the Python *time* library (time.time())

To help you, we've provided you with the code that comes along with the textbook, which includes a representation of Sudoku in csp.py. You'll notice that Sudoku subclasses both CSP and Problem, and so it is fairly easy to call the different search methods on it right out of the box.

Here are the key elements of this task:

- Figure out how to call the methods provided by the textbook.
- **Make the modifications needed to output the correct counts.**
- Make the modifications needed to run naïve BFS and DFS on the Sudoku puzzle.
- Modify the Sudoku representation to reflect the 2x2 board size. This will also make it easier to test your code.

Use the output of your program to fill in the table below.

Algorithm	**Board**	#nodes generated /assignments made	max nodes stored	Running time
BFS	1	90	23	0.0030739307403564453
BFS	2	24	2	0.0017390251159668281
BFS	3	57	7	0.0025310516357421875
DFS	1	53	8	0.0023951530456542971
DFS	2	16	3	0.0015130043029785156
DFS	3	40	4	0.0018301010131835938
Backtracking	1	63	N/A	0.0010399818420410156
Backtracking	2	14	N/A	0.0022399425506591797
Backtracking	3	34	N/A	0.0017359256744384766

2. Does the running time and space used by BFS and DFS align with the complexity analysis presented in the textbook? Explain your reasoning.

For BFS, the time complexity as explained in the textbook should be O(b^d) where d is the number of levels generated. Regarding BFS space complexity, it is given to also be O(b^d) where that is the size of the frontier. After running all three of the example grids with the BFS algorithm, you can see that the run time and the space complexity correlate in relation to the number of nodes generated and the difficulty of the board. For this reason, I do believe that the time and space complexity are accurately represented in the book because when the levels of the tree were larger, the space complexity and time complexity both accurately represented that.

For DFS I noticed a similar situation. I was able to see that again, the time and space complexity directly correlated with the number of nodes generated and the difficulty of the board. The time complexity for DFS in the textbook is mentioned to be O(b^m) where m represents the number of levels. The space complexity, is also mentioned to be O(b^m). For this reason and the reasons stated above regarding BFS search while also taking into consideration the run times and nodes stored for each algorithm I think the text book presents an accurate representation of both algorithms.

3. In the table, why does it not make sense to fill in the max nodes stored for Backtracking?

It would make sense to not store the max nodes stored because at any given point, that value can be extremely large and not actually representative of the nodes searched for a solution because of the fact that the algorithm backtracks.

4. Next, test the following three algorithms on **exampleSudokus-q1.txt:**

a. Backtracking with minimum remaining values and least constrained values heuristic ("backtracking-ordered").
b. Backtracking with no value and variable ordering ("backtracking-noOrdering").
c. Backtracking with heuristics reversed – try the least constrained variable and most constraining value ("backtracking-reverse").

It is your choice whether to use forward checking inference or AC-3. *a* and *b* are implemented for you, but *c* will require a new implementation. You should modify your Sudoku solver implementation to take the above three Strings as possible values for your algorithms parameter.

Algorithm	**Board**	**# assignments made**	**Running time**
Backtracking-ordered	1	31	0.0010991096496582031
Backtracking-ordered	2	18	0.0009701251983642578
Backtracking-ordered	3	24	0.0009281635284423828
"backtracking-noOrdering"	1	41	0.0012989044189453125
"backtracking-noOrdering"	2	26	0.0005340576171875
"backtracking-noOrdering"	3	34	0.0005741119384765625
"backtracking-reverse"	1	36	0.0019221305847167969
"backtracking-reverse"	2	19	0.001977205276489258
"backtracking-reverse"	3	28	0.0015146732330322266

5. Did you choose to use forward checking or AC-3. Why did you make that decision? Reference principles learned in this course.

I chose to use forward checking because of the fact that it checks only the constraints between the current variable and the future variable. This meaning that when a value is assigned to the current variable, any value in the domain of the future variable that has a conflict with this assignment is then temporarily removed from the domain. Therefore, it allows for branches of the search tree that will lead to failure to be pruned earlier.

6. **Are your results what you would have expected to see? Explain with reference to the # of assignments made and running time.**

The results are not exactly what I had expected to see. It was interesting that the running time of the backtrack-noOrdering was the fastest because of the fact that it was using no variable and no value assignments. This is sort of interesting to me then because in what cases/with what constraints are these variable and value heuristics beneficial. Regarding the number of assignments made,

Part B. Class Scheduling (50 points)

We will now move to a different constraint satisfaction problem; the problem of scheduling classes. Initially, this problem is subject to the following conditions:

- The same teacher can't teach two different classes at the same time
- Two different sections of the same class shouldn't be scheduled at the same time.
- Classes in the same area shouldn't be scheduled at the same time.
- *Note: don't worry about the labs and recitations, just the main sections of the courses*

Implement a scheduleCourses function that takes two parameters:
- The name of an input file consisting of the courses to be scheduled
- A number of possible "slots" for the courses

For example, if possible class days were M/W or T/Th, and class can start at 9:30AM, 11AM, 12:30PM, 2PM, or 3:30PM, the number of possible time slots is 10.

The input file is formatted as follows:

> Course number; course name; sections; labs; recitations; (professors);(sections each professor teaches), (areas)

Items in parentheses represent lists that could have 0 or more items.

Here are two example lines of the input file. You'll notice that there are no areas listed for the second line, but multiple professors with multiple sections (for example, K. Bigrigg teaches 3 sections).

CS1571;Introduction to Artificial Intelligence;1;0;0;E. Walker;1;AI,DS
CS0007;Introduction to Computer Programming;5;0;2;J.Cooper, K.Bigrigg, S. Ellis;1,3,1;

Represent this problem as a CSP by answering the following questions.

7. What are the variables:

Variables: {Courses}

8. What are the domains of each variable:

Domains: {Slots}

9. What are the constraints:

Constraints: {Course1(slot1) != Course2(slot2), Course1 != Course2}

Run a backtracking search (using mrv, a degree heuristic, and lcv) with AC-3 inference on this problem to output to a file a viable schedule to this problem, given the file and number of timeslots. Your file should consist of a series of "course number-teacher-section" and timeslot pairs "CS1571-Walker-1, 0", separated by semicolons.

This requires two modifications to the existing codebase:

- The implementation and use of a class that extends CSP and sets up the variables, domains, and constraints for the course scheduling problem
- The implementation of the degree heuristic function, named "degree"

We will be providing a sample input file for you to test your code on named partB-courseList-shortened.txt.

Part C. Navigating Around Campus (50 points)

Finally, in the last part of this assignment, you will use A* to find the quickest path between two intersections on campus, given location and elevation data. It will be up to you to decide how to define shortest, and to make sure your heuristics work with the definition that you make.

Implement a *findPath* function that takes a two intersection names as inputs, and an algorithm to use (either Astar or idAstar). Intersection names are formatted as follows: "Forbes,Bouquet". Your goal is to find the path between two intersections that will have the quickest walking time. *findPath* should output the recommended route and expected time, given the algorithm provided.

To accomplish this, you will need data on walking routes around campus. We will give you two files. The first is called partC-intersections.txt. It contains latitutde, longitude, and elevation data for each intersection. Each line of the file is formatted as follows:

```
Forbes,Bouquet,40.4420,-79.9564,279
```

Forbes and Bouquet are the cross streets, 40.4420 is the latitude, -79.9564 is the longitude, and 279 is the elevation in meters.

The other file is named partC-distances.txt and contains distances for each route between intersections in miles. It is formatted as follows:

```
Forbes,Bouquet,Forbes,Bigelow,0.18
```

The two intersections are Forbes & Bouquet and Forbes & Bigelow, and the distance is .18 miles.

You can assume that the elevation difference between each intersection represents the pedestrian's path (e.g., moving from an intersection at 240m to an intersection at 239m means the pedestrian is traveling downhill 1m). It is your responsibility to do the conversions between intersections' latitude and longitude coordinates, distances in miles, and expected time of travel (incorporating elevation into account).

The output of your function should be formatted as follows:

```
Forbes,Bouquet,Forbes,Bigelow,Forbes,Bellefield,10
```

The pedestrian is traveling from Forbes & Bouquet to Forbes & Bigelow to Forbes & Bellefield, and it is expected to take 10 minutes.

Answer the questions on the following page.

10. Design a heuristic function for use with A* that incorporates both distance and elevation information. What is your function?

```
Def h(self, node):

        X1, y1 = node.state.get_distance()

        X2, y2 = node.state.get_elevation()

        X3, y3 = self.goal

        Return abs(x3-(x2+x1)) + abs(y2-(y2+y1))
```

11. Why do you believe this is a good heuristic? Reference whether it is admissible and consistent in your answer.

I believe this was a good heuristic because based off the examples in the textbook relating to the Manhatten problem, I was trying to incorporate it the same way in the sense of taking both the elevation and distance into consideration and then using the goal state as a reference point. I thought this would be admissible because for all n, h(n) <= h*(n) and consistent because f(n) is non decreasing along every path. I think if I understood this probably a little bit more and had the opportunity to think about this more I could have found a heuristic that might have been better to use.

12. Run both A* and iterative deepening A* with findPath (this will require you to implement iterative deepening A*). Do the two algorithms return different results? Why or why not?

IdA* returns a lower amount of memory usage compared to A*. Because A* is expanding the node with the lowest f(n) cost, it should be minimizing the total estimated solution cost. It evaluates the nodes by combining the cost to reach the node and the cost to get from the node to the goal and for this reason it is slower and takes up more memory. idA* on the other hand concentrates on exploring the most promising nodes and therefore doesn't go to the same depth everywhere in the search tree. Using the idea of the threshold makes this implementation faster and use less memory. So due to these concerns, and the idea that the heuristic plays a large role in A* that may cause it to overshadow the path cost, they both cold potentially return different results based on the way they are traversing the tree and finding solutions.