Assignment 1 – Alan Shen

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
- 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())

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	1423253	1048576	11.549 secs
BFS	2	481706	262144	3.675 secs
BFS	3	118169	65536	0.625 secs
DFS	1	537589	41	8.420 secs
DFS	2	234250	37	2.031 secs
DFS	3	93631	33	0.735 secs
Backtracking	1	16	N/A	0.0 secs
Backtracking	2	16	N/A	0.0 secs
Backtracking	3	16	N/A	0.0 secs

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

Time complexity for BFS and DFS is $O(b^m)$, which is represented by the number of nodes generated. Space complexity is $O(b^m)$ for BFS and O(bm) for DFS. Each of the boards has b = 4 since 4 branches from each node are possible. For board 1, m = 10 since there are 10 open spaces. For board 2, m = 9 since there are 9 open spaces. For board 3, m = 8 since there are 8 open spaces.

For BFS, the expected time and space complexities for board 1, 2, and 3 are O(4^10) = O(1048576), O(4^9) = O(262144), and O(4^8) = O(65536). These values match the space complexities exactly for BFS. Regarding the time complexity, the nodes generated are on the same order of magnitude as the big-O values. In addition, the running times from board 1 to board 3 decrease by about a factor of 4, which makes sense since the big-O runtimes also decrease by four from board 1 to board 2 and board 2 to board 3. BFS aligns with the complexity analysis.

For DFS, the expected time complexities for board 1, 2, and 3 will be the same. Board 1's value of 537589 is greater than 4^9 and less than 4^10 and is therefore O(4^10). Board 2's value of 234250 is very close to O(4^9). Finally, board 3's value of 93631 is very close to O(4^8). In addition, the runtimes from board 1 to board 3 also decrease by about a factor of 4. In this way, the time complexities align. Expected space complexities would be O(40), O(36), and O(32) for boards 1, 2, and 3, respectively. These values almost exactly match the maximum nodes stored values of 41, 37, 33. This can be explained by the addition of the root node.

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

> In backtracking, the algorithm only needs to store the current assignments. In this case, because there are 16 spaces in the grid, there will be a max of 16 assignments. In backtracking, if a value to a variable doesn't work, no assignment occurs. Instead, subsequent values are tested until no conflicts appear. This process continues until all variables are filled. If at a certain point no values work on a variable, the previous assignment gets removed. In this way, backtracking isn't generating 4 nodes from each assignment at once since it first checks beforehand if the assignment is possible. In this way, no nodes are being stored.

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 constraining 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	16	0.0 secs
Backtracking-ordered	2	16	0.0 secs
Backtracking-ordered	3	16	0.0 secs
"backtracking-noOrdering"	1	22	0.004987 secs
"backtracking-noOrdering"	2	20	0.003114 secs
"backtracking-noOrdering"	3	18	0.003955 secs
"backtracking-reverse"	1	64	0.015653 secs
"backtracking-reverse"	2	41	0.015592 secs
"backtracking-reverse"	3	44	0.015619 secs

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 AC-3 since for each node, there are lots of neighbors. In a 4x4 sudoku puzzle, a single square has binary constraints with up to 7 other neighbors within the same row, column, or 2x2 grid. Therefore, when a node gets assigned a value, AC-3 propagates these constraints across the other nodes/variables, allowing a solution to be found faster.

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

The number of assignments align perfectly with what was expected. Backtracking with the most constrained variable and least constraining value yielded the lowest number for all three boards, which makes sense since these are the best heuristics. In addition, backtracking with the least constrained variable and most constraining value yielded the highest number for all three boards, which makes sense since these are the opposite of the best heuristics.

In terms of runtime, backtracking-ordered had the best runtimes for all three boards, so that aligned with expectations since that algorithm's heuristic yielded the lowest number of assignments. In addition, backtracking-reverse also had the worst runtimes which makes sense since reverse had the most assignments.

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:

Course section information, including professor and area (E.g. CS1571-1-Walker-AI,DS, CS0007-1-Cooper, CS0007-2-Bigrigg, CS0007-3-Bigrigg, CS0007-4-Bigrigg, CS0007-Ellis, CS1632-1-Ahn, CS1632-2-Ahn, etc…)

8. What are the domains of each variable:

The possible timeslots. (E.g. M/W 9:30, M/W 11, M/W 12:30, etc….)

9. What are the constraints:

For any variables X and Y such that they are two different course sections:

$X_{teacher} = Y_{teacher} \rightarrow$ X does not equal Y (Same teacher implies different time slots)

$X_{course} = Y_{course} \rightarrow$ X does not equal Y (Same course implies different time slots)

$\exists$ area $\in X_{area}$ s.t. area $\in Y_{area} \rightarrow$ X does not equal Y (Sections share ≥1 area implies different time slots)

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?

The distance between each degree of latitude is 69 miles. The distance between each degree of longitude at 40.4420 degrees latitude (latitude of Pittsburgh) is 52.468 miles. The conversion from meters to miles is 1 meter = 0.0006213712 miles.

Let 2D-Distance be:

Sqrt{ [(Latitude(n) – Latitude(goal)) * 69]^2 + [(Longitude(n) – Longitude(goal)) * 52.468]^2 }

Let h(n) = sqrt{ (2D-Distance)^2 + [(Elevation(n) – Elevation(goal)) * 0.0006213712]^2 }

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

This is a good heuristic since this represents the straight-line distance in miles from node n to node goal. This straight-line distance also incorporates elevation change as well. The straight-line distance represents the true shortest path to the goal, but the actual path cost to get to the goal from any node will take longer since the person might have to walk on particular streets to get to particular intersections. This underestimation means the heuristic is admissible.

In addition, because the heuristic is straight-line distance (and therefore represents shortest possible distance), any path taken to another neighboring node plus the straight-line distance to the goal from the neighboring node will be equivalent or greater than the heuristic at the original node. As a result, this heuristic is consistent.

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?

The algorithms return the same results. Because the heuristics are consistent, A* search will return the optimal path no matter if it's a graph or tree search. Iterative deepening A* slowly increases the threshold for the f(n) value until the goal is reached. Therefore, the path will also be optimal since the mechanism of iterative A* means that the shortest/most optimal path to a node will be explored first.