**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
* 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:

* 1. The solution to the puzzle as a String in the same format as the input string.
  2. Total number of nodes created (or in the case of backtracking, the number of assignments tried)
  3. The maximum number of nodes kept in memory at a time (ignore in the case of backtracking)
  4. 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.

Within my results, I found that DFS is far more efficient in **runtime** and **memory** required than BFS. I found the following facts in my analysis of DFS and BFS on Sudoku:

* BFS had much more nodes generated and nodes stored, in general more memory required
* DFS had significantly faster runtime

This aligns with the book, as it warns of the vast amount of memory BFS can take. When assigning a new value to a node, the CSP has 10 different values it can choose from. And since the memory complexity is O(bd), the frontier quickly gets larger as each child is expanded at every level. Whereas in DFS, since this is a finite depth that is relatively small the amount of nodes stored on the stack remains vastly smaller, and the number of nodes generated also remains smaller as the space complexity is O(bm).

For runtime, my analysis somewhat followed the books logic. Mainly, the book praises DFS memory complexity, but it does mention that if the space state is finite then DFS time complexity is bounded by m (max depth of a node) with O(bm). In Sudoku, m is low for the search tree as you can only possibly have an m value up to 81 (usually less from initial boards given). On the other hand, BFS has a time complexity of O(bd) and again since the depth is limited to up to 81, you would expect similar runtimes. The key difference which makes DFS better however, is that it digs all the way to the leaf nodes **first**, and this is where the solution to Sudoku must be located. Whereas BFS must go all the way to depth 80, and is guaranteed no solutions until it begins its depth of 81 expansion of the frontier.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm** | **Board** | #nodes generated /assignments made | max nodes stored | Running time |
| BFS | 1 | 1,423,252 | 1,048,576 | 12.46 |
| BFS | 2 | 715,954 | 262,144 | 4.15 |
| BFS | 3 | 211,798 | 65,536 | 0.74 |
| DFS | 1 | 537,588 | 31 | 5.95 |
| DFS | 2 | 234,249 | 28 | 2.26 |
| DFS | 3 | 93,630 | 25 | 0.84 |
| Backtracking | 1 | 82 | N/A | 0.00 |
| Backtracking | 2 | 40 | N/A | 0.00 |
| Backtracking | 3 | 48 | N/A | 0.00 |

1. Does the running time and space used by BFS and DFS align with the complexity analysis presented in the textbook? Explain your reasoning.
2. In the table, why does it not make sense to fill in the max nodes stored for Backtracking?

For backtracking search, there is no need for multiple states (new nodes). It simply keeps a single copy of the state and continues to alter it until a solution is found. It will pick an unassigned variable (Sudoku square) and keep filling in assignments as long as there are no conflicts. And when there are no conflict-free assignments left in the domain, it backtracks and never needs to create a new state. Rather, reuse the existing one.

1. Next, test the following three algorithms on **exampleSudokus-q1.txt:**
   1. Backtracking with minimum remaining values and least constrained values heuristic (“backtracking-ordered”).
   2. Backtracking with no value and variable ordering (“backtracking-noOrdering”).
   3. 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 | 0.00 |
| Backtracking-ordered | 2 | 16 | 0.00 |
| Backtracking-ordered | 3 | 16 | 0.00 |
| “backtracking-noOrdering” | 1 | 23 | 0.00 |
| “backtracking-noOrdering” | 2 | 22 | 0.00 |
| “backtracking-noOrdering” | 3 | 18 | 0.00 |
| “backtracking-reverse” | 1 | 32 | 0.00 |
| “backtracking-reverse” | 2 | 57 | 0.00 |
| “backtracking-reverse” | 3 | 28 | 0.00 |

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

Yes, for backtracking unordered it always would perform worse in the number of assignments than ordered. This is likely because MRV chooses variable to assign with the fewest legal values. This allows the algo to prune large parts of the tree earlier to prevent taking the wrong path. Combined with LCV which will choose a value from the domain that restricts its neighbors domains the least to assign to the variable, which will allow the algo to pick a value that is more likely to achieve the single solution we are looking for. And similarly, reversing the MRV will cause the algo to not prune branches forcing more wrong paths, and reversing LCV will look for the least likely values first also slowing the algorithms progress.

I chose forward checking because in the case of Sudoku, the rules of the game benefit from an early elimination of illegal domain assignments for arc-related boxes (same row, col, or 3x3 grid). By removing these values from those related boxes, it significantly decreases the branching factor for each assignment and therefore improves time & space complexity. I used this over AC-3 because AC-3 is only a preprocessing heuristic that doesn’t give a solution. Forward checking allows you to use inferences as the assignment increases, and also gives you a solution. In this sense, I found forward checking more practical of an approach.

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

**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.

1. What are the variables:

Courses C, Times T

Courses include the class number, teacher, section, and area.

The times will be every time slot available for the scheduling process.

1. What are the domains of each variable:

T = {1, 2, 3, 4, … n } (where n is that command line user input)

C = {“CS1632Laboon-Section1-QA”, “CS1632Laboon-Section2-QA”, …} (where each course is a line on the file input containing all the classes)

1. What are the constraints:

The neighbors of a given course (defined by sharing a course number, professor, section, **or** area) must have an AllDiff() for the allowable time slots T

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.

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

My function I created is located in both the main() function in findPath.py (line 36), and in utils.py the distance() function. For main() I did the following:

* Create a heuristic value that was [latitude, longitude, elevation] that was sent for every node in the a\_start() heuristic function
  + This adds to the original [latitude, longitude]

In distance() I did the following:

* I utilized code to find the distance in miles of the two coordinates
* I utilized the difference in height to find the elevation change in miles
* I found the diagonal (using pythagorean theroem) between the height (elevation change) and the length (distance between coords)

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

This is a good heuristic because when I use my debugPrintPath() to check the estimated heuristic cost to the goal and the real cost to the goal, I find that the **heuristic value is always lower** than the actual value. This means my h() value produced by my function is admissable.

My heuristic is also good because the heuristic estimated cost at a lower node in the path is always **higher** than its parent node’s estimate. This means that as the path gets longer, the estimated cost also increases. An inconsistent heuristic could have a lower estimate as the path gets longer, however mine continues to have an increasing path cost with every additional node.

1. 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 do **not** have different results. Both the intersections chosen and the total minutes computed do not change. However, there are key differences in the following run between O’Hara,Bouquet and Bayard,Bellefield as listed below:

* Astar
  + Nodes Generated – 20
* IDAstar
  + Nodes Generated – 189

This makes sense, as the path length is 8 so IDAstar must regenerate the nodes at least 8 times with increasing limits. Whereas Astar using the heuristic can very quickly go down the right path and only need to generate a couple of nodes before finding the solution.