COSC-4368 Artificial Intelligence
Problem set 1

James Richardson
1555520

All source code will be included at the very end of this document.

Source code is also included in the provided zip file. Each problem is in its own directory named after the problem it is implementing.

Instructions on how to run the code and what is needed to run it will be provided in a readme file in each of the directories. Most of the problems have a makefile, so simply run `make` to build and run the program. For problems 1 and 4, rust and cargo will need to be installed as they were implemented in the programming language rust.

Problem 1

Breadth First Search

GOAL Reached first	G2
Expanded state	S, A, B, F, C, D, E, G2
OPEN List	G1, E
CLOSE List	S, F, B, A

Depth First Search

GOAL Reached first	G1
Expanded state	S, A, C, D, G1
OPEN List	F, B
CLOSE List	S, A, C, D

Best First Search

GOAL Reached first	G1
Expanded state	S, A, D, G1
OPEN List	S, F, B, A, C, D, G1
CLOSE List	S, A, D

A* Search

GOAL Reached first	G1
Expanded state	S, A, C, D, F, D, G1

OPEN List	B
CLOSE List	S, A, C, D, F

SMA* Search

GOAL Reached first	G1
Expanded state	S, A, F, B, A, C, D, S, D, G1
OPEN List	A, C, D
CLOSE List	S

Problem 2

State Space:

1: L 2: L Cabbage 3: L Goat 4: L Wolf 5: L Cabbage(DEAD) Goat 6: L Cabbage Wolf 7: L Goat(DEAD) Wolf 8: L Goat(DEAD) Wolf Cabbage(DEAD)	R Farmer Cabbage Goat Wolf R Farmer Goat Wolf R Farmer Cabbage Wolf R Farmer Cabbage Goat R Farmer Wolf R Farmer Cabbage R Farmer
9: L Farmer 10: L Farmer Wolf 11: L Farmer Goat 12: L Farmer Cabbage 13: L Farmer Wolf Goat 14: L Farmer Wolf Cabbage 15: L Farmer Cabbage Goat (G)16: L Farmer Cabbage Goat Wolf	R Cabbage (DEAD) Goat (DEAD) Wolf R Cabbage (DEAD) Goat R Cabbage (DEAD) Wolf R Goat (DEAD) Wolf R Cabbage R Goat R Wolf R

Operators:

The farmer moves left, or right, with one or no companions. The Farmer can only move from one side to the other, if he is already on that first side, so The farmer can only move to the left for states 1-8 and can only move right for states 9-16.

Valid for states 1-8:
Left(Farmer)

Valid For States 1, 3, 4, and 7 (when farmer and cabbage are both on left):
Left(Farmer, Cabbage)

Valid for States 1, 2, 4, 6:
Left(Farmer, Goat)

Valid for States 1, 2, 3, 5:
Left(Farmer, Wolf)

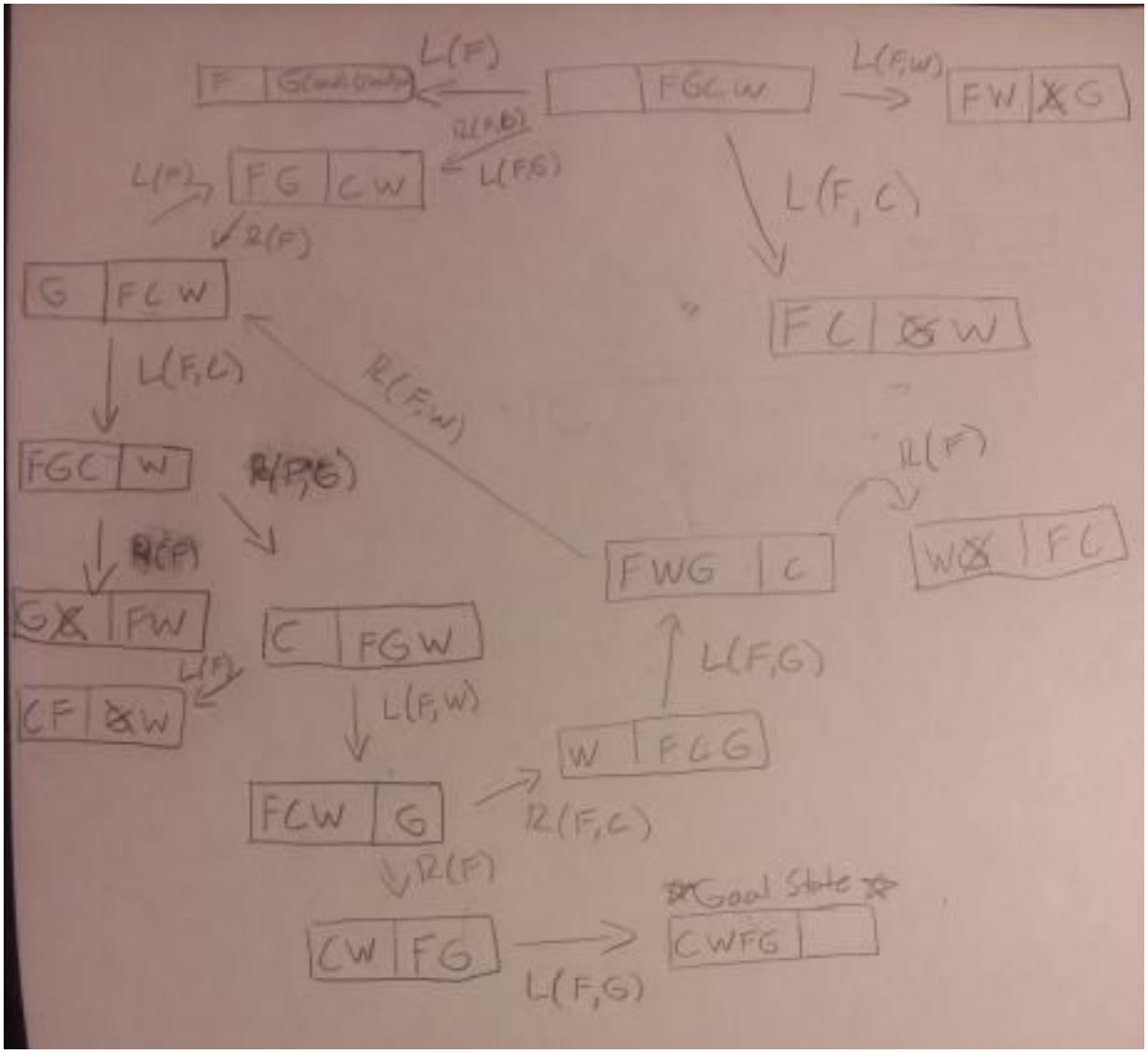
Valid for States 9-16:
Right(Farmer)

Valid for States 12, 14, 15, 16:
Right(Farmer, Cabbage)

Valid for States 11, 13, 15, 16:
Right(Farmer, Goat)

Valid for States 10, 13, 14, 16:
Right(Farmer, Wolf)

Diagram:

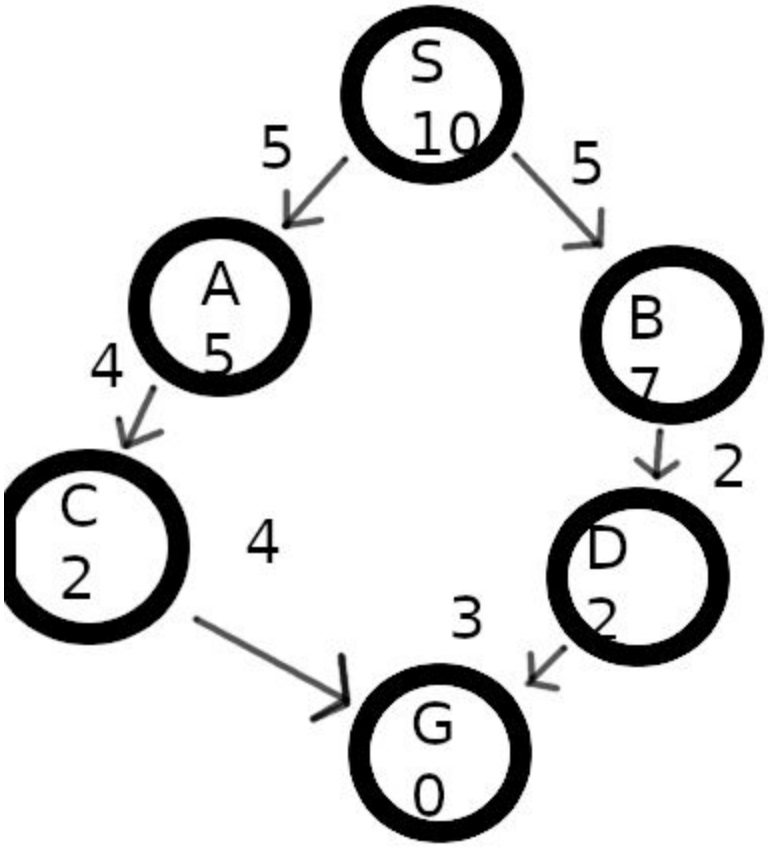

L(F)
L(F,W)
FGCW
FW X G
L(F,G)
L(F)
FG CW
R(F)
L(F,C)
G FCW
FC X W
L(F,C)
R(F,W)
FGC W
R(F,G)
L(F)
FWG C
W X FC
R(F)
G X FW
C FGW
L(F)
L(F,G)
CF X W
L(F,W)
W FGG
FCW G
R(F,C)
R(F)
Goal State
CW FG
CWFG
L(F,G)

Problem 3

a) Even though a goal node is found, there is still the possibility that there is a better path to another goal node or even that same goal node. Since the heuristic will underestimate the cost of a particular node to the goal node, some nodes may significantly underestimate the cost to the goal state causing them to be expanded sooner, even though they don't actually contain the best path.

In the attached picture, we will expand nodes in this order:

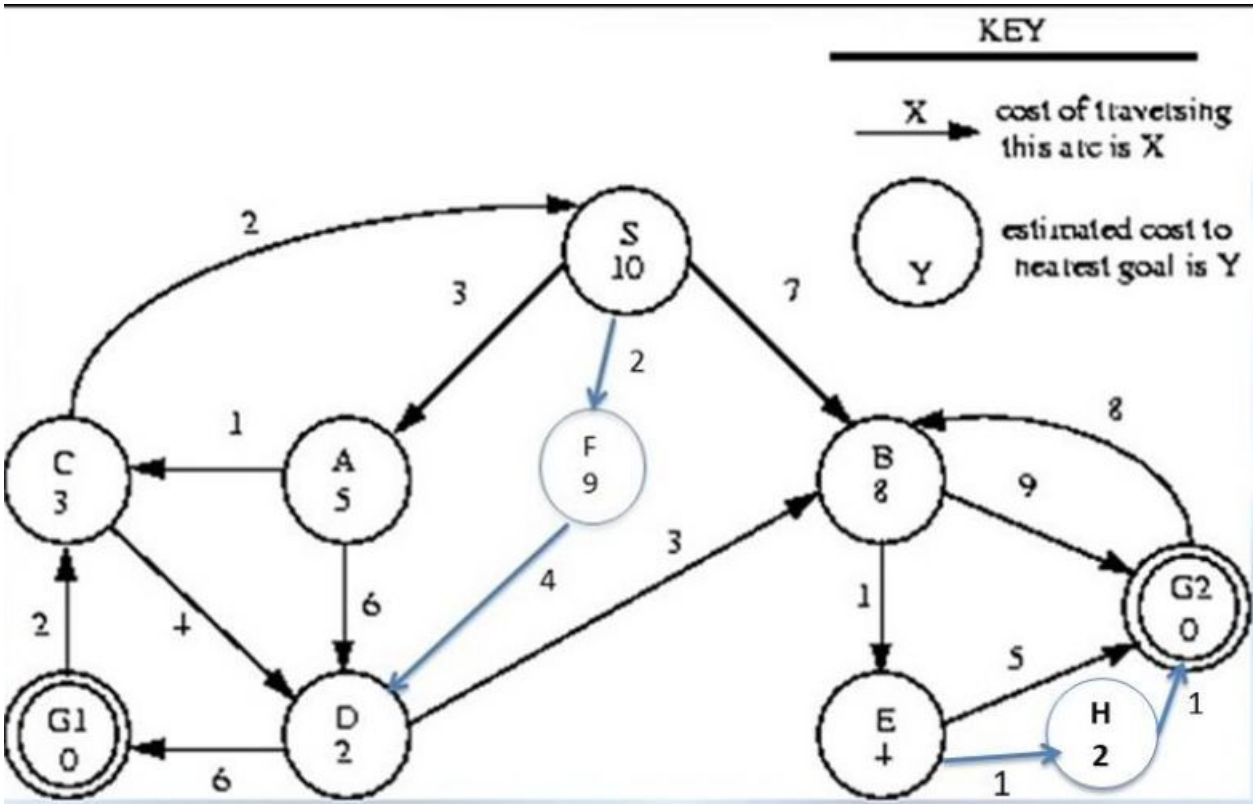
expanded node	open list
S	[(A, 10), (B, 12)]
A	[(C, 11), (B, 12)]
C	[(B, 12), (G (from A), 13)]
B	[(D, 9), (G (from A), 13)]
D	[(G (from D), 10), (G (from A), 13)]
G (from D)	



This shows how expanding the goal node as soon as we saw it would have been a bad decision because it was not the optimal path to the goal node.

This also shows another interesting feature of the A* algorithm, a feature that is actually the main concern of the next problem. If one of the nodes on the right were to overestimate their cost to the goal node, we would never have found the more optimal path. This is the beauty of the A* algorithm: It doesn't matter if the heuristic is a little low, because the path to the the current node thus far is also an equal consideration. Since the nodes on the right do not overestimate their distance to the goal, the nodes on the left can underestimate as much as they want the the algorithm will still spit out the optimal path in the end.

b) In the Depicted graph, the solution that would be founded by A* is not the optimal solution.



A* will expand nodes in this order
Expanded: S, A, C, D, F, G1

resulting in the following solution:

S -> A -> D -> G1

This solution has a cost of
3 + 6 + 6 = 15

However, the actual optimal solution for this graph is:
S -> B -> E -> H -> G2

This solution has a cost of:
7 + 1 + 1 + 1 = 10

The reason that this solution is not chosen is because B over estimates the cost to a goal state. This breaks the fundamental law of the A* heuristic, you can underestimate as much as you want but never overestimate. In reality, the minimum distance from B to the goal G2 is only 3 away, it overestimates by a significant amount and, because of the heuristic, seems to be much further away from the goal than other nodes in the open list and thus is not chosen as the solution by A* as a result!

Problem 4

For this problem I simply implemented the A* search but for the particular graph we were given. The heuristic in this case was calculated from the haversine formula. The haversine formula that I used is given below in rust:

```
fn haversine(lat1: f64, lon1: f64, lat2: f64, lon2: f64) -> f64 {
        let p: f64 = 0.017453292519943295;      // Math.PI / 180
        let a: f64 = 0.5 - f64::cos((lat2 - lat1) * p) / 2.0 + f64::cos(lat1 * p) *
                f64::cos(lat2 * p) * (1.0 - f64::cos((lon2 - lon1) * p)) / 2.0;

        return 12742.0 * f64::asin(f64::sqrt(a)); // 2 * R; R = 6371 km
}
```

This gives the result in kilometers though, I then convert the result into miles (because that's that the given distances seemed to be in).
I did not come up with this code myself, I copied it from someone on stack overflow and converted it into rust. The stack overflow link is below:

https://stackoverflow.com/questions/27928/calculate-distance-between-two-latitude-longitude-points-haversine-formula

Pseudo Code:

```
A* (start, end, graph):
        Create origin mapping
        Create distance mapping
        Create priority queue for the open list
        Create closed list

        While the priority queue is not empty:
                Current = pop from priority queue

                Add current to closed list

                For each neighbor of current
                        Store, in local variables, the distance from current to the neighbor, the gscore
                        to the neighbor, the hscore for the neighbor to the goal and the fscore of the
                        neighbor.

                        If we have not yet visited this neighbor (not in closed list):
                                Insert the neighbor in to the queue, gscore map, cost map and origin
                                map

                        If we have visited but the gscore is better than what we stored:
                                Update all of the same things we did in the previous if but do not add
                                to queue
```

Results:

```
Minneapolis -> Houston: Cost: 1145
Minneapolis
St. Louis
Houston

San Francisco -> Chicago: Cost: 1891
San Francisco
Salt Lake City
Denver
Chicago

New York -> Los Angeles: Cost: 2464
New York
Chicago
Denver
Los Angeles
```

Problem 5

Pseudocode:

```
Find Neighbors(sp, how_many, r):
        Neighbors = []
        For ii = 0..how_many:
                #additionally I ensure that sp[ii] + random is bounded by [0, 1]
                neighbors.append(<sp[0] + random(-r, r),
                                 sp[1] + random(-r, r),
                                 sp[2] + random(-r, r)>)
        Return neighbors
RHC(sp, p, r, seed) -> (int, float, (float, float, float))
        Loop:
                Find p nearest neighbors based on r and seed
                If none of these neighbors are better, terminate
                If we find a better neighbor, update our current position
```

Data

p=20 & r=0.02	#sol	Best sol	F val	# sol	Best sol	F val	# sol	Best sol	F val
(0.5, 0.5, 0.5)	45	(0.540402, 0.992467, 0.000322)	0.424368	35	(0.433036, 0.894626, 0.002120)	0.351070	42	(0.452406, 0.950782, 0.000457)	0.382342
(0, 0.5, 1)	35	(0.004136, 0.996833, 0.998523)	0.658667	38	(0.005924, 0.998698, 0.999823)	0.661441	29	(0.001392, 0.955149, 0.997664)	0.599552
(0.9, 0.6, 0.3)	22	(0.999004, 0.710428, 0.001281)	0.374722	20	(0.993339, 0.696466, 0.002888)	0.366088	19	(0.999125, 0.709329, 0.002425)	0.373266

p=20 & r=0.05	#sol	Best sol	F val	# sol	Best sol	F val	# sol	Best sol	F val
(0.5, 0.5, 0.5)	22	(0.521491, 0.993299, 0.008722)	0.412537	22	(0.591360, 0.986756, 0.003261)	0.431461	16	(0.553418, 0.999034, 0.009960)	0.422066
(0, 0.5, 1)	13	(0.018523, 0.915491, 0.998063)	0.524635	13	(0.001445, 0.995152, 0.996002)	0.655083	17	(0.005712, 0.991401, 0.996388)	0.644440
(0.9, 0.6, 0.3)	8	(0.989063, 0.686713, 0.001900)	0.362174	15	(0.997158, 0.982266, 0.006217)	0.524717	11	(0.990721, 0.700638, 0.000231)	0.368442

p=100 & r=0.02	#sol	Best sol	F val	# sol	Best sol	F val	# sol	Best sol	F val
(0.5, 0.5, 0.5)	34	(0.494250, 0.999127, 0.000631)	0.415546	44	(0.550893, 0.998098, 0.000692)	0.429607	48	(0.619004, 0.998917, 0.001303)	0.446997
(0, 0.5, 1)	31	(0.000532, 0.998656, 0.996613)	0.662750	31	(0.004132, 0.997422, 0.999028)	0.660508	34	(0.001595, 0.997605, 0.999185)	0.664594
(0.9, 0.6, 0.3)	36	(0.982332, 0.999038, 0.000082)	0.542576	34	(0.996477, 0.998927, 0.000298)	0.545829	40	(0.989556, 0.999040, 0.000839)	0.543283

p=100 & r=0.05	#sol	Best sol	F val	# sol	Best sol	F val	# sol	Best sol	F val
(0.5, 0.5, 0.5)	27	(0.852569, 0.998588, 0.001190)	0.507102	30	(0.960870, 0.987120, 0.003792)	0.522916	18	(0.697380, 0.991534, 0.001559)	0.462557
(0, 0.5, 1)	15	(0.003540, 0.995264, 0.990989)	0.643010	13	(0.000541, 0.995194, 0.987103)	0.639613	13	(0.000281, 0.989967, 0.997444)	0.651637
(0.9, 0.6, 0.3)	13	(0.985975, 0.985879, 0.002611)	0.530047	15	(0.999782, 0.999024, 0.006967)	0.536408	17	(0.978875, 0.996826, 0.000008)	0.540179

Run 37

p=10000000 & r=0.5	#sol	Best sol	F val
(0, 0.5, 1)	3	(0.000071, 0.999453, 0.999034)	0.669219

The raw data is also included in `problem5/data.txt`

Interpretation
Randomized hill climbing is pretty... random. Depending on where you start out, you will end up with a significantly different value. If we start off closer to a local maxima, then we will probably end up just at that local maxima and nowhere better. Even for the same starting point, neighborhood size and value for r some of the runs output significantly different values due to the random way in which the neighbors are generated.

The three major factors in this algorithm, are the three major parameters that we were varying. Where we start, how far away we look, and how many things we look for all have a huge impact on the ending result. If we start close to a local maxima, but are able to look far enough away and sample enough neighbors, we might just be able to look past the local maxima to a better value. However if we start very close to the actual maxima for our equation and have a small distance to look around us, we can still end up finding the absolute maximum.

If we have a large neighborhood (we are going to sample many more adjacent entities) but we are not looking far enough away, this might be worse than better. This could easily amplify the issue of getting caught in a local maximum if the better option immediately around the starting position leads to a local maximum. To get the best values, we would need to have a huge range of possible distances and a huge neighborhood. Though nothing is without flaw: increasing the neighborhood in my implementation increased the time taken for the search.

To answer the question more directly:
The closer we start to the actual maximum, the better the likelihood of finding and the easier it will be to obtain the maximum. So the effect of ‘sp’ on the results is hard to determine, as it would depend specifically on how close it is to the real value. In this particular example, it would appear that the point (0.0, 0.5, 1) was the closest to the maximum as it yielded the best results. The starting point could also be detrimental to the solution quality though if we start somewhere where we can get trapped.

The effects of p and r are much more direct. The higher they are, the better the quality of our solution. But also, the higher that they are, the longer the solution will take to find. In my 37th solution, I put this to the test, using a significantly higher p and r value (while also starting at what seemed to be the best starting point). This lead to some pretty good results, though admittedly not significantly better than those for neighborhood size of 100 (especially considering that when I made the neighborhood size so huge it took more than 100 times as long).

Problem 6

a) For this problem, I did the brute force approach. I implemented the solution recursively rather than iteratively. I iterate through the unallocated variables and try to apply each of the unallocated values until all of the constraints are satisfied. To do this, I select an unallocated value for the particular variable I am considering, if there are still more variables that aren't mapped I call the function again, with the new mapping. This will continue until all of the the variables are mapped, meaning there are no more unallocated variables. If the constraints are satisfied I return true and the function will terminate, otherwise I will return false and the next value for the variable will be attempted. If none of the available values work for this variable we return false and then the caller tries the next value. This continues until all of the constraints are satisfied.

b)

```
        Our recursive function will return True if a solution is found, False if it
        is not

CSP(unallocated_variables, unallocated_values, mapping)
        if we have no more unallocated variables, check the constraints
        return either true or false
            if we return true, print out the mapping and return

        make a copy of the unallocated values
        pick the next variable from the unallocated variables, remove it from the
        unallocated variables list.
loop:
        pick the next value from the unallocated values copy, remove it from the
        unallocated values list

        update the mapping

        call the csp function with the unallocated variables, minus the one that we
        popped off for this iteration and the unallocated values minus the value
        that we are using for this particular mapping

        if the call returns false, try the next value

        otherwise return true
```

c)

We have three input variables, unallocated_variables, unallocated_values and mapping. Mapping is a dictionary (hash map) of variable to value pairs, unallocated_variables and unallocated_values are lists of the variables and values that are yet to be used. If the unallocated_variables list is empty, we check the constraints. If the constraints are satisfied then we return true and the function will terminate, if the constraints are not satisfied we return false. Then we loop through the remaining unallocated values. For each of these values we update the mapping variable as we try and assign that value to our variable. Then we pass in these updated variables into the CSP function again. If the function returns true we return true. If it doesn't we continue with the loop. If none of the elements in the loop work, we return false.

This only obtains one solution though. To obtain all of the solutions, I return False instead of true but still print out the mappings.

d)

My program found the following solutions. The solution is of a python dictionary, the mapping variable that the program uses to be exact.

{'L': 1, 'E': 4, 'T': 2, 'C': 5, 'A': 3, 'H': 0, 'O': 6, 'M': 7}	{'L': 4, 'E': 6, 'T': 3, 'C': 1, 'A': 2, 'H': 0, 'O': 5, 'M': 8}
{'L': 1, 'E': 4, 'T': 2, 'C': 6, 'A': 5, 'H': 0, 'O': 7, 'M': 9}	{'L': 4, 'E': 6, 'T': 3, 'C': 5, 'A': 1, 'H': 0, 'O': 9, 'M': 7}
{'L': 1, 'E': 4, 'T': 2, 'C': 7, 'A': 5, 'H': 0, 'O': 8, 'M': 9}	{'L': 4, 'E': 6, 'T': 3, 'C': 5, 'A': 2, 'H': 0, 'O': 9, 'M': 8}
{'L': 1, 'E': 4, 'T': 2, 'C': 8, 'A': 3, 'H': 0, 'O': 9, 'M': 7}	{'L': 5, 'E': 2, 'T': 1, 'C': 3, 'A': 4, 'H': 0, 'O': 8, 'M': 6}
{'L': 1, 'E': 6, 'T': 3, 'C': 4, 'A': 2, 'H': 0, 'O': 5, 'M': 8}	{'L': 5, 'E': 2, 'T': 1, 'C': 3, 'A': 7, 'H': 0, 'O': 8, 'M': 9}
{'L': 2, 'E': 8, 'T': 4, 'C': 3, 'A': 1, 'H': 0, 'O': 5, 'M': 9}	{'L': 5, 'E': 2, 'T': 1, 'C': 4, 'A': 6, 'H': 0, 'O': 9, 'M': 8}
{'L': 2, 'E': 8, 'T': 4, 'C': 5, 'A': 1, 'H': 0, 'O': 7, 'M': 9}	{'L': 5, 'E': 4, 'T': 2, 'C': 1, 'A': 3, 'H': 0, 'O': 6, 'M': 7}
{'L': 3, 'E': 2, 'T': 1, 'C': 4, 'A': 6, 'H': 0, 'O': 7, 'M': 8}	{'L': 5, 'E': 6, 'T': 3, 'C': 4, 'A': 1, 'H': 0, 'O': 9, 'M': 7}
{'L': 3, 'E': 2, 'T': 1, 'C': 5, 'A': 4, 'H': 0, 'O': 8, 'M': 6}	{'L': 5, 'E': 6, 'T': 3, 'C': 4, 'A': 2, 'H': 0, 'O': 9, 'M': 8}
{'L': 3, 'E': 2, 'T': 1, 'C': 5, 'A': 7, 'H': 0, 'O': 8, 'M': 9}	{'L': 5, 'E': 8, 'T': 4, 'C': 2, 'A': 1, 'H': 0, 'O': 7, 'M': 9}
{'L': 3, 'E': 2, 'T': 1, 'C': 6, 'A': 5, 'H': 0, 'O': 9, 'M': 7}	{'L': 6, 'E': 2, 'T': 1, 'C': 3, 'A': 5, 'H': 0, 'O': 9, 'M': 7}
{'L': 3, 'E': 4, 'T': 2, 'C': 6, 'A': 1, 'H': 0, 'O': 9, 'M': 5}	{'L': 6, 'E': 4, 'T': 2, 'C': 1, 'A': 5, 'H': 0, 'O': 7, 'M': 9}
{'L': 3, 'E': 8, 'T': 4, 'C': 2, 'A': 1, 'H': 0, 'O': 5, 'M': 9}	{'L': 6, 'E': 4, 'T': 2, 'C': 3, 'A': 1, 'H': 0, 'O': 9, 'M': 5}
{'L': 4, 'E': 2, 'T': 1, 'C': 3, 'A': 6, 'H': 0, 'O': 7, 'M': 8}	{'L': 7, 'E': 4, 'T': 2, 'C': 1, 'A': 5, 'H': 0, 'O': 8, 'M': 9}
{'L': 4, 'E': 2, 'T': 1, 'C': 5, 'A': 6, 'H': 0, 'O': 9, 'M': 8}	{'L': 8, 'E': 4, 'T': 2, 'C': 1, 'A': 3, 'H': 0, 'O': 9, 'M': 7}

e)

```
variables = ["L", "E", "T", "C", "A", "H", "O", "M"]
values = list(range(0, 10))

################################################################################
#   This class is just for ease of use, so I don't have to manually calculate
#   the constraints inside of my CSP
################################################################################
class Constraint:
        def __init__(self, result, variable1=None, variable2=None, carry=None):
        assert(result)
        assert(isinstance(result, str))

        if variable1:
        assert(isinstance(variable1, str))
        if variable2:
        assert(isinstance(variable2, str))

        if not variable1 or not variable2:
        # Can't have just one variable in this particular problem
        assert(variable1 is None)
        assert(variable2 is None)
        #we MUST have a carry if we don't have any varaibles in this problem
        assert(not carry is None)

        if carry:
        assert(isinstance(carry, Constraint))

        self.v1 = variable1
        self.v2 = variable2
        self.r = result
        self.carry = carry

        def __repr__(self):
        if self.v1 and self.carry:
        return "{} + {} + {} = {}".format(self.v1, self.v2, "carry", self.r)

        if self.v1:
        return "{} + {} = {}".format(self.v1, self.v2, self.r)

        return "{} = {}".format("carry", self.r)

        def all_mapped(self, mapping):
        if not self.v1:
        return self.r in mapping and self.carry.all_mapped(mapping)

        if self.v1 not in mapping or self.v2 not in mapping or                  \
        self.r not in mapping:
        return False

        if self.carry and not self.carry.all_mapped(mapping):
        return False

        return True
```

```
        def get_carry(self, mapping):
        #this function should not be called if we don't have variables
        assert(not self.v1 is None and not self.v2 is None)
        assert(self.all_mapped(mapping))

        carry = self.carry.get_carry(mapping) if self.carry else 0

        return (mapping[self.v1] + mapping[self.v2] + carry) // 10

        def calculate_solution(self, mapping):
        c = self.carry.get_carry(mapping) if self.carry else 0

        return mapping[self.v1] + mapping[self.v2] + c if self.v1 else c

        def constraint_satisfied(self, mapping):
        # This function should never be called unless all of the variables have
        # been mapped
        assert(self.all_mapped(mapping))

        if self.carry:
        assert(self.carry.all_mapped(mapping))
        return mapping[self.r] == self.calculate_solution(mapping)


################################################################################
#   Make a list of all of the constraints.
#   In the CSP, we will check that each of these are satisfied
################################################################################
constraints = [
        Constraint("E", variable1="T", variable2="T"),
]
constraints.append(Constraint("M", variable1="E", variable2="A",
                carry=constraints[-1]))
constraints.append(Constraint("O", variable1="L", variable2="C",
                carry=constraints[-1]))
constraints.append(Constraint("H", carry=constraints[-1]))

################################################################################
#   This is the exact same thing as the pseudo code.  The Constraint class make
#   everything significantly more readable
################################################################################
def csp(unallocated_variables, unallocated_values, mapping):
        if not unallocated_variables:
        # find out if all of the constraints are satisfied and return the result
        all_satisfied = all([each.constraint_satisfied(mapping) for each in
                constraints])
        if all_satisfied:
        print(mapping)
        return all_satisfied

        variable = unallocated_variables.pop(0)
```

```
            values_to_use = unallocated_values[:]

            for value in values_to_use:
            mapping[variable] = value

            if csp(unallocated_variables[:],
            [each for each in unallocated_values[:] if each != value],
            mapping):
            return True

            return False

csp(variables[:], values[:], {})
```

f)
simply run

python3 csp.py

In any shell.

Source code and instructions on running are also provided in the `problem6` directory.