



# Report

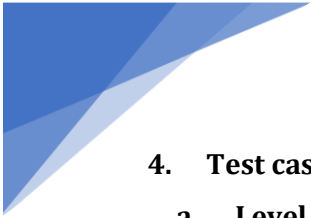
## Introduction to artificial Intelligence

### Cryptarithmic Problem



## TABLE OF CONTENTS

1. Members information: .....	4
2. Assignment plan: .....	4
3. Environment to compile and run the program: .....	4
4. General idea: .....	4
5. Algorithms and demonstration: .....	4
<b>Code general mechanism: .....</b>	<b>4</b>
a. File: .....	7
i. readfile: .....	7
ii. remove_parathesis: .....	8
iii. savefile: .....	8
b. Project1_AI_Cryptarithmic_Problem: .....	8
i. Library and modules import: .....	8
ii. timer: .....	8
iii. main: .....	9
c. Constraint: .....	12
i. Class Constraint: .....	12
d. CSP_Class: .....	14
i. Library and modules import: .....	14
ii. Class CSP: .....	14
e. Backtrack: .....	15
i. Library and modules import: .....	15
ii. Backtrack: .....	16
iii. isComplete: .....	20
iv. Assigned: .....	20
v. reAssigned: .....	20
f. Multiply: .....	21
i. isComplete: .....	21
ii. get_result: .....	21
iii. Backtrack_multiply: .....	22
g. Demonstration: .....	24



<b>4. Test cases:</b>	25
<b>a. Level 1:</b>	25
<b>b. Level 2:</b>	26
<b>c. Level 3:</b>	26
<b>d. Level 4:</b>	27
<b>5. Reflection and comment:</b>	27
<b>6. References:</b>	28

## 1. Members information:

Student ID	Full name
19127135	Phạm Bảo Hân
19127415	Huỳnh Duy Hưng
19127574	Lâm Ngọc Tiến

## 2. Assignment plan:

N.o	Task	Member	Duration
1	Generate idea	Hân	3 days
2	Code: main, Backtrack, Multiply, Constraint, CSP_Class	Hân	10 days
3	Code: main, File	Hưng	7 days
4	Reorganize code, comment	Tiến	3 days
5	Write report	Tiến, Hân	7 days
6	Testing	Tiến, Hưng	1 day

## 3. Environment to compile and run the program:

Python.

## 4. General idea: (taken directly from project's exercise question for closer and easier grasp to the idea)

Cryptarithmic Problem is a type of constraint satisfaction problem where the game is about digits and its unique replacement either with alphabets or other symbols. In cryptarithmic problem, the digits (0-9) get substituted by some possible alphabets or symbols. The task in cryptarithmic problem is to substitute each digit with an alphabet to get the result arithmetically correct.

## 5. Algorithms and demonstration:

### Code general mechanism:

Combine Backtracking and CSP methods.

Step by step:

- Step 1:
  - Initialize variables taken from input file, append carry numbers and blank character " " where necessary.

- Initialize **sign** with a plus (+) always at first to do additional operators between carry numbers and first row character candidates. For instance, **sign** will be ['+', '+'] following the example input below. In case of parenthesis, they will be removed and perform changes on the included operators where needed.
- Step 2:
  - Create a CSP with the above variable and domain ranges from 0 to 9.
  - Apply unary constraint to remove illegal values in domain through these restrictions:
    - First candidate character does not hold value 0 in the domain.
    - Domain of " and c0 only hold value 0.
- Step 3:
  - Apply Backtracking.
  - Consider each variable from the rightmost to leftmost and from above to down.

$$\begin{array}{rcccccc}
 & c4 & c3 & c2 & c1 & c0 & \\
 + & & & & & & \\
 & & S & E & N & D & \\
 + & & & & & & \\
 & & M & O & R & E & \\
 \hline
 & M & O & N & E & Y & \\
 c0 & c4 & c3 & c2 & c1 & & 
 \end{array}$$

If the variable during the execution of algorithm experiences these following situations:

- Variable exists within the formula (before equal sign):
  - No value has been assigned to variable: Assign a suitable value and change the result on that column operation result based on the recently assigned value. Then, perform Backtracking.
  - Value has been assigned to variable: Change the result on that column operation result based on the assigned value. Then, perform Backtracking.
- Variable is the result value:

- If variable has yet been assigned and the operation result from that column without another variable assigned, then we assign this variable and perform backtrack to the memorized location.
- If variable has been assigned and the result from the considering column equals to the value assigned before, simply apply backtrack until memorized location reached.
- Variable happens to be the carry number:
  - Memorized value (or carry number) does not have constraint except for c0 must be value 0.

If Backtrack succeeds, returns true. Otherwise, we must execute the value from which we receive from the resulting operation from the column back to its original state. At the end if Backtrack can no longer be performed, return false (or fail), which means the formula has no result.

To calculate an operation on a column:

- For addition (+) and subtraction (-):  
Every time a variable is traversed during the operation, we add or subtract that value of variable with the result of the considering column. After implementation, if column result larger than 10, we minus it by 10 and carry number will hold 1 after applying addition, if column result less than 0, we increase it by 10 before applying subtraction and make carry number holds -1.
- For multiplication (\*):

These algorithms are implemented based on how multiplication table is applied in real life, we would like to present our explanation on this matter using that table:

$$\begin{array}{r}
 \begin{array}{ccc}
 C3 & C2 & C1 \\
 A & B & C
 \end{array} \\
 * \\
 \begin{array}{cc}
 D & E
 \end{array} \\
 \hline
 \begin{array}{ccc}
 A * E & B * E & C * E \\
 A * D & B * D & C * D
 \end{array} \\
 \hline
 \begin{array}{ccc}
 3 * 1 & 2 * 1 & 1 * 2 \\
 3 * 2 & 2 * 2 & 1 * 2
 \end{array}
 \end{array}$$

C3, C2, C1 are column 3, 2, 1 on row 1 (A B C) consecutively. Similarly, C2, C1 as column 2, column 1 on row 2 (D E).

In reality, we multiply each candidate and the next leftwise but a column backward:

$$\begin{array}{ccc} & \text{-----} & \\ & \text{A}*E & \text{B}*E & \text{C}*E \\ \text{A}*D & \text{B}*D & \text{C}*D \end{array}$$

So, we come up with a strategy: To use indices to calculate these character candidate in order. The strategy follows like this:

$$\begin{array}{ccc} & \text{-----} & \\ & 3*1 & 2*1 & 1*2 \\ 3*2 & 2*2 & 1*2 \end{array}$$

We multiply their value at the corresponding indices (E.g. 3\*1 being A \* E), then for each column and row, we decrease and increase by 1 consecutively (E.g 3\*1 decrease 3 by 1 and increase 1 by 1 we have 2\*2 at the next line).

It should result in:

$$\begin{array}{ccc} & \text{-----} & \\ \boxed{3*2} & \boxed{\begin{array}{c} 3*1 \\ 2*2 \\ 1*3 \end{array}} & \boxed{\begin{array}{c} 2*1 \\ 1*2 \end{array}} & \boxed{1*2} \end{array}$$

However, 2\*3 and 1\*3 do not exist because there is no third candidate character at row D E.

Afterward, apply addition by column to receive the result we seek for.

The following component sections are the names (a, b, c and so on...) of files and their included functions (i, ii, iii and so on...) which is submitted along with this report.

#### a. **File:** `File.py`

##### i. **readfile:** `def readfile():`

- Simply read the formula to file, this formula has a few restrictions:
  - Formula must not have space between characters.
  - Only one line.
  - Only these following operators are allowed:
    - +
    - -
    - \* (only 2 operands!)
    - =
    - Open and close parenthesis. The next character to a close parenthesis must be a mathematic operator.
  - The input file at the end must not include another line or another empty line (or '\n').

- Input characters of the files, and store it to an array.
- Take the largest number of characters in the row, other rows whose length is less than this value (E.g. row 'SEND' and 'MORE' have 4 characters whereas 'MONEY' has 5 characters), blank character " " will be added to fill up the length (E.g. row 'SEND' now becomes ' SEND' which has 5 characters).
- The array of characters has to append the memorized values at top and bottom.  
E.g. Input: SEND+MORE=MONEY  
The character array: [['c4', 'c3', 'c2', 'c1', 'c0'], ['', 'S', 'E', 'N', 'D'], ['', 'M', 'O', 'R', 'E'], ['M', 'O', 'N', 'E', 'Y'], ['c0', 'c4', 'c3', 'c2', 'c1']]
- Now, we need to deal with parenthesis due to its priority and extract the parenthesis for the next function manners for easier executions. For this part, **remove\_parenthesis()** is implemented, which will be reported in the next section.
- In order to use the sign corresponding to the values from input file, the array stores signs is created with sign '+', this means the memorized value (carry number) is always an addition to the operands. Then it appends all the signs of the input.
- At the end, return the array of characters and the array of signs.

#### ii. remove\_parenthesis: `def remove_parenthesis(sign)`

- As mentioned from the above section, **remove\_parenthesis** function's goal is to eliminate parentheses in the corresponding manner.
- When approaching parentheses, the simple trick is to reverse the operators if before the open one is a minus (-).

#### iii. savefile: `def savefile(output)`

- The output function. It first opens the output file in **file** in write mode, then variable **tmp** used to output the sorted **output** parameter passed to **savefile**. However, before printing out to file, a condition check gate is required. For **c** within **tmp**, if **c**'s length equals to 1, write **output** at **c** index. If there is not result after execute the algorithm, write 'NO SOLUTION' to output file.

### b. Project1 AI Cryptarithmic Problem:

`Project1_AI_Cryptarithmic_Problem.py`


#### i. Library and modules import:

- Backtrack (as BTLib)
- CSP\_Class (as CSPLib)
- Constraint (as CSLib)
- Multiply (as BTMLib)
- The entire of File
- Multiprocessing

#### ii. timer:

**WAITING\_TIME:** This global variable is used to notifies the program how long maximum can the user wait. If the input happens to be very long, the user will





grow tired of waiting, this will help produce the result in a less time-consuming manner.

```
WAITING_TIME = 7 #minutes
```

### iii. main:

#### 1. main: `def main():`

- First, fetch the result of input file with **inp** to store variables and **sign** to store sorted operators from **readfile()**.
- Initialize **mulSign** as False, this variable is a flag which will be used later.

```
inp,sign=readfile()  
mulSign = False
```

- Initialize domain and assignment:
  - o **domain:** Includes alphabets and their domains.
  - o **asg\_list:** Used to aid Backtrack algorithm, this variable stores each candidate alphabets in order.

```
domain = {}  
asg_list = []
```

- A loop called **traversal** in **sign**, if a multiply operator '\*' is encountered, **mulSign** will be turned to True.

```
for traversal in sign:  
    if (traversal == '*'):  
        mulSign = True
```

- Next, we call another for loop, for easier explanation, here is the code:

```
for j in range(len(inp[0]) - 1, -1, -1):  
    for i in range(len(inp)):  
        if(inp[i][j] != ''):  
            domain[inp[i][j]] = [v for v in range(0, 10)]  
        else:  
            domain[inp[i][j]] = [0]  
        asg_list.append(inp[i][j])
```

```
asg = {}  
asg['list'] = asg_list  
asg['index'] = 0  
asg['max_len'] = [len(inp[0]), len(inp)]
```

Loop **j** in range equalizes to the number of a calculation column with step - 1 until -1 reached, for each time, loop **i** traverses throughout number of row consecutively, at which considers the following situations:

c4 c3 c2 c1 c0  
 +  
   S E N D  
 +  
   M O R E  
 -----  
 M O N E Y  
 c0 c4 c3 c2 c1

- **inp[i][j]** differentiates from an empty character "", execute the following format:

```
domain[inp[i][j]] = [v for v in range(0, 10)]
```

**domain** at character **inp[i][j]** is now a list with **v** contains value from 0 to 10.

- Otherwise, execute:

```
domain[inp[i][j]] = [0]
```

In the situation where **domain** at **inp[i][j]** equals to 0 for which in the case of character is "" results in holding value 0.

- At the end, append **inp[i][j]** to **asg\_list**.

```

asg = {}
asg['list'] = asg_list
asg['index'] = 0
asg['max_len'] = [len(inp[0]), len(inp)]

```

- The asg is the dictionary, consists of:
  - List: Variables assigned in a manner.
  - Index: Index of the assigned variables.
  - Max\_len: Row and column numbers. For which, **len(inp[0])** meaning the amount of carry numbers accordingly to the maximum character sequence candidate will be column numbers, **len(inp)** is the amount of row will be row numbers.

These are evaluated and considered during our mechanism implementation using how normal operators and operands formula works in real life except an addition to carry numbers row at the top and below with indices as **asg['index']** follows as demonstrated like this:



c4	c3	c2	c1	c0
+				
	S	E	N	D
+				
	M	O	R	E
-----				
M	O	N	E	Y
c0	c4	c3	c2	c1

Diagram illustrating the carry indices for the cryptarithm  $SEND + MORE = MONEY$ . The indices are labeled above and below the digits. A red arrow points upwards from the bottom  $c1$  to the top  $c1$ , and another red arrow points downwards from the top  $c0$  to the bottom  $c1$ .

Step by step indices traversal:  $C0 \rightarrow D \rightarrow E \rightarrow Y \rightarrow C1_{(bottom)} \rightarrow C1_{(top)} \rightarrow N \rightarrow R \rightarrow E$  and so on...

For  $\{C0, D, E, Y, C1_{(bottom)}, C1_{(top)}, N, R, E, \text{etc...}\}$  – which is also the **asg['list']**, then the **asg['index']** should be  $\{0, 1, 2, 3, 4, 5, 6, 7, 8, \text{etc...}\}$

We will explain the functions and how we use these carry numbers during the report.

- The **variable** array will be used to store value of each variable exists in the domain. Initialize the **variable** entries with character candidates from the input. This will be fixated to assign value which will be demonstrated later within this report.

```
variables = []
for i in domain:
    variables.append(i)
ExeCSP = CSPLib.CSP(variables, domain)

ExeCSP.get_constraint().unary_constraint(inp, domain)
```

- Create CSP from the **variables** and apply unary constraint on them.
- Execute and perform the corresponding Backtrack normally or exclusively for multiply operator based on **mulSign** which is executed above. Backtrack will be reported at the Backtrack section of this report.

```

if (mulSign): # If multiply operator found, do backtrack exclusively for multiply
    if BTMLib.Backtrack_multiply(asg, ExeCSP, q1=[], q2=[], result=0) == False:
        #print("NO SOLUTION")
        savefile("NO SOLUTION")
        return
else: # Otherwise, execute the normal backtrack
    if BTMLib.Backtrack(asg, ExeCSP, sign, 0, 0) == False:
        #print("NO SOLUTION")
        savefile("NO SOLUTION")
        return

# print(ExeCSP.get_all_values())
savefile(ExeCSP.get_all_values())

```

- If Backtrack returns false, no solution is found, return empty to notifies the program it has failed. Otherwise, save the result to output.

A shorter version based on our *main* source code:

- Step 1: Initialize variables supporting the source code.
- Step 2: Check if there exists multiply operator using **mulSign**.
- Step 3: Perform unary constraint.
- Step 4: Execute CSP list.
- Step 5: Depending on **mulSign**, do normal backtrack or exclusively for multiply operator backtrack.

## 2. Main of timer:

```

if __name__ == '__main__':
    t=WAITING_TIME*60
    p = multiprocessing.Process(target=main)
    p.start()
    p.join(t)
    if p.is_alive():
        p.terminate()
        print("No result")
        p.join()

```

If timer exceeds **WAITING\_TIME**, halts the program and terminate with the output "No result".

### c. Constraint: Constraint.py

#### i. Class Constraint:

```
class Constraint:
```

This class is implemented to create constraint for variables.

### 1. init: `def __init__(self):`

Constraint ranges from 0 to 10 using list **used** initialized to be False. This list will be used to check whether the number used or not. For instance, if candidate character 'A' has already taken number 7, other candidate characters such as 'B', 'C', 'D',... is not permitted to take 7 again.

```
self.used = []
for v in range(0, 10):
    self.used.append(False)
```

### 2. get used: `def get_used(self, var):`

Returns all the used constraints.

### 3. set used: `def set_used(self, var, value):`

Set a constraint to a value.

### 6. unary constraint for operand:

```
def unary_constraint_for_operand(self, inp, domain):
```

- This function implements unary constraint for each character by removing 0 in domain among the first candidates (not carry numbers). Also, blank character " will have domain of one and only number 0.

```
'''
Apply unary constraint for operand
inp  : list of input character
domain: a dictionary map the character to its domain
'''

# Remove 0 in domain among the first candidates except c
for i in inp[1:-1]:
    j = 0
    while(i[j] == ''):
        j += 1
    if(0 in domain[i[j]]):
        domain[i[j]].pop(0)
domain[''] = [0]
```

E.g.

E.g. There are 5 characters in row 'MONEY' and 4 character in row 'SEND' and 'MORE'.

Because of the length differences, we plan to add an additional blank character "", for this we need to consider 2 things:

- Blank character "" will have domain of one and only number 0.
- The first character in the character sequences 'SEND', 'MORE' and 'MONEY', which are 'S' and 'M' consecutively MUST not be 0.

These 2 restrictions are the sole purposes for unary constraint on operands.

## 7. unary constraint for c: `def unary_constraint_for_c(self, inp, domain):`

- Unary constraint for carry number. Mainly to make sure c0 holds only 0. This is due to which at the position 'c0' appears, there wouldn't be any number to be memorized (or carry number required). At the bottom row of carry number, at the first index however also has c0, since there are no other characters at the left exist, c0 can be put here with only value 0. Also the top row of carry number, at the last index, because there is no calculation before so the memorized value is 0.

```
  c4 c3 c2 c1 c0
+
      S E N D
+
      M O R E
-----
  M O N E Y
c0 c4 c3 c2 c1
```

```
...
Apply unary constraint for memorized operand
inp  : list of input character
domain: a dictionary map the character to its domain
...
for i in inp[0]:
    domain[i].clear()
domain['c0'] = [0]
```

## 8. unary constraint: `def unary_constraint(self, inp, domain):`

- Simply call the **unary\_constraint\_for\_operand** and **unary\_constraint\_for\_c** using **inp** and **domain** as passing parameters.

```
self.unary_constraint_for_operand(inp, domain)
self.unary_constraint_for_c(inp, domain)
```

### d. CSP Class: `CSP_Class.py`

Creates a Constraint Satisfaction Problem.

#### i. Library and modules import:

- Constraint (as CSLib)
- Deepcopy (from copy)

#### ii. Class CSP: `class CSP:`



A Constraint Satisfaction Problem consists of these components:

- domains: A dictionary that maps each and every variable to its domain.
- constraints: Constraints's list.
- variables: Variables' set.
- values: Value for each variable, these will be the result at the end for each character (E.g. 'A' = 7, 'B' = 1, etc...)

1. **init:** `def __init__(self, variables, domains):`

In case more constraint is needed, initialize using the above listed components.

```
"""
A CSP consists of:
variables    : a set of variables
domains      : a dictionary that maps each variable to its domain
constraints  : a list of constraints
values       : a dictionary that maps each variable to its value
"""

def __init__(self, variables, domains): # Add more constraint is required

    self.variables = variables
    self.domains = domains
    self.constraint = CSLib.Constraint()

    self.value = {}
    for i in variables:
        self.value[i] = None
    self.value[''] = 0
    self.value['c0'] = 0
```

2. **get domain:** `def get_domain(self, var):`

Which domain does the variable included in will be returned.

3. **get value:** `def get_value(self, var):`

Return variable's value.

4. **set value:** `def set_value(self, var, value):`

Set value to a variable.

5. **get constraint:** `def get_constraint(self):`

Return constraint.

6. **get all values:** `def get_all_values(self):`

Return every values of every variables.

e. **Backtrack:** `Backtrack.py`

i. **Library and modules import:**

- Deepcopy (from copy)
- Floor (from math)
- CSP\_Class (as CSPLib) – Not necessary, this is included during our test.

**ii. Backtrack:** `def Backtrack(assignment, csp, sign, result, mem):`

- This function contains 5 passing parameters:
  - **assignment:** Assignment which is initialized in **main()** as **asg**.
  - **csp:** CSP variable taken from **CSP\_Class**, initialized in **main()** as **ExeCSP**.
  - **sign:** Operators from input.
  - **result:** Result of the operations executed within the calculating column.
  - **mem:** Acts as carry number (memorized value) of the calculating column.
- Firstly, check whether backtracking is complete or not using the function **isComplete** which is demonstrated in the next section. If it is completed, return true.

```
if isComplete(assignment):
    return True
```

---

- Otherwise, start by initialize variable **var** with the value at **assignment** at **assignment['index']**

```
var = assignment['list'][assignment['index']]
location = [int(floor(assignment['index'] / assignment['max_len'][1])),
            int(assignment['index'] % assignment['max_len'][1])] # col, row
constraint = csp.get_constraint()


if location[1] == 0: # At the first character in a column
    result = 0
    mem = 0
```

---

- The corresponding row and column of the above variable in the input should be:
  - Row: `int(floor(assignment['index'] / assignment['max_len'][1]))`.
  - Column: `int(assignment['index'] % assignment['max_len'][1])`

Again, refer to our favorite demonstration table:





c4	c3	c2	c1	c0	
+					
	S	E	N	D	
+					
	M	O	R	E	
-----					
M	O	N	E	Y	
c0	c4	c3	c2	c1	

Take character 'N' at second row 'SEND', its **asgn['index']** is 6. So its row is 1 and column is 1.

- Afterward, fetch the constraint: `constraint = csp.get_constraint()`
- If the **location[1]** is 0, **result** and **mem** are also 0. Which means, we are considering a new column.

c4	c3	c2	c1	c0	<--- Row 0
+					
	S	E	N	D	<--- Row 1
+					
	M	O	R	E	<--- Row 2
-----					
M	O	N	E	Y	<--- Row 3 (result position)
c0	c4	c3	c2	c1	<--- Row 4 (memorized value position)

- Perform a stage in which we call result positioning, for this in order to minimize such a long explanation, we will try to sum up the general idea of this loop.

```


# Result positioning
if assignment['max_len'][1] - location[1] == 2:
    if result in csp.get_domain(var):
        if (csp.get_value(var) == None and constraint.get_used(result) == False):
            csp.set_value(var, result)
            constraint.set_used(result, True)
            assignment['index'] += 1
            if Backtrack(assignment, csp, sign, result, mem) == True:
                return True
            assignment['index'] -= 1
            constraint.set_used(result, False)
            csp.set_value(var, None)
        else:
            if csp.get_value(var) == result:
                assignment['index'] += 1
                if Backtrack(assignment, csp, sign, result, mem) == True:
                    return True
                assignment['index'] -= 1
    return False

```

- Check whether **assignment['max\_len'][1] - location[1]** equals to 2, meaning **assignment['max\_len'][1]** is the row number whereas **location[1]** is the calculating row number, (which is the index to the result row 'MONEY' in the example), consider result in **get\_domain** for **var** of **csp** these two cases:
  - If **var** has not been assigned and **result** is not used (E.g. before putting number 7 for 'A' to take, consider whether 'A' has taken any number and number 7 has yet taken):
    - Set value to **var** with **result** and set it to already used.
    - Increases index by 1 and perform **Backtracking** again (Backtracking the next index). If it is success, returns true.
    - Minus index by 1 again (Recursion fails).
  - Otherwise if **var** has set **value** and it shares the same **value** as **result**:
    - Also perform the above second sentence forward:
      - Increases index by 1 and perform **Backtracking** again.
      - If it is true, returns true.
      - Minus index by 1 again (Recursion fails).

By increasing and decreasing index by 1 in the above cases, we can check whether the backtracking is going on the right path and reaching the end of the formula in the consecutive manner.

- Return false if the above conditions are not succeeded.
- The next conditions are used specifically for the situation where **assignment['max\_len'][1] - location[1]** equals to 1. This is the position for the



memorized value. Similar to the conditions above except we use **remain** in place of the above **result**:

```
# Memorized value location
if assignment['max_len'][1] - location[1] == 1:
    remain = mem
    if var == 'c0' and not(mem == 0):
        return False
    csp.set_value(var, remain)
    assignment['index'] += 1
    if Backtrack(assignment, csp, sign, result, mem) == True:
        return True
    assignment['index'] -= 1
    csp.set_value(var, None)
    return False
```

- Then we determine conditions using **var** equals to 'c0' and **mem** not equals to 0.

As explained in **unary\_constraint\_for\_c** section, if the considering variable is c0, its value must be 0, but if it happens to not be 0 then it will be wrong, return false.

Otherwise, other carry numbers can be assigned a value through calculations.

- The below code concentrates on the considering value of operand variable, whether to be usable or not also shares the same principle.

```
# Operand location
if csp.get_value(var) == None:
    domain = csp.get_domain(var)
    for asg_value in domain:
        # Check whether considering value is used or not
        if constraint.get_used(asg_value) == False:
            constraint.set_used(asg_value, True)
            tmp = Assigned(location[1], sign, result, asg_value, mem)
            result = tmp[0]
            mem = tmp[1]
            csp.set_value(var, asg_value)
            assignment['index'] += 1
            if Backtrack(assignment, csp, sign, result, mem) == True:
                return True
            assignment['index'] -= 1
            csp.set_value(var, None)
            tmp = reAssigned(location[1], sign, result, asg_value, mem)
            result = tmp[0]
            mem = tmp[1]
            constraint.set_used(asg_value, False)
    else:
        tmp = Assigned(location[1], sign, result, csp.get_value(var), mem)
        result = tmp[0]
        mem = tmp[1]
        assignment['index'] += 1
        if Backtrack(assignment, csp, sign, result, mem) == True:
            return True
        assignment['index'] -= 1
        tmp = reAssigned(location[1], sign, result, csp.get_value(var), mem)
        result = tmp[0]
        mem = tmp[1]
    return False
```

- These lines of codes are implemented to check each individual character candidate. We first check whether **var** has been assigned the value.
  - If not, assign the **asg\_value** in the **domain** for **var**. After that, the **result** and **mem** in that column are changed depending on the sign, either '+' or '-', which will then be explained in **Assigned** section below. Set value **asg\_value** from **domain** to **csp**.
    - Now perform **Backtrack** just the above, we consider **Backtrack** the following indices and return true if **Backtrack** succeed.
    - Otherwise keeps narrowing the indices number back, returns **csp** to None at **var**.
    - Following that, reassign **tmp** using **reAssigned** to return the old value of **result** and **mem**, which will also be demonstrated later. Then, set the **used** of this value in the **constraint** is false.
  - If **var** has been assigned the value before, also perform **Assigned**, **Backtrack** then **reAssigned** if **Backtrack** fails without any need to consider marking **constraint** to be used or give value to **csp**.

### iii. isComplete:

- This function's sole goal is to check whether backtracking is complete by comparing whether **asg['index']** and **asg['list']**'s length is equal. A proper explanation would be to consider if the current considering index has the same value as the length of the determining list due to having passed only the amount of indices, which in turn should be the same length.

```
return asg['index'] == len(asg['list'])
```

### iv. Assigned:

```
def Assigned(k, sign, result, value, mem):
    if k == 0 or sign[k-1] == '+':
        result += value
        if result > 9:
            result -= 10
            mem += 1
    else:
        if result - value < 0:
            result = result + 10
            mem -= 1
        result -= value
    return [result, mem]
```

- Explain together with **reAssigned**.

### v. reAssigned:

```
def reAssigned(k, sign, result, value, mem):
```

```

if k == 0 or sign[k-1] == '+':
    if result - value < 0:
        result = result + 10
        mem -= 1
    result -= value
else:
    result += value
    if result > 9:
        result -= 10
        mem += 1
return [result, mem]

```

- Assign variable with a value, add the value to the sum of the column. If value cannot be assigned to variable, return to the previous result. Both has the following parameters to be passed:
  - **k**: Considering location within the calculating column. If k=0, it takes in the first row (the carry numbers row c4 c3 c2 c1 c0). Due to k=-1 does not exist, an individual condition is required, hence 'or'.
  - **sign**: Operators in order as given from the input file.
  - **result**: Assign result.
  - **value**: Variables' values.
  - **mem**: Carry number (memorized value).
- In **Assigned**:
  - Addition (+): If value added larger than 9, minus that result by 10 and remember its carry (**mem += 1**).
  - Minus (-): If value reduced lower than 0, increase that result by 10 and remember its carry (**mem -= 1**) then perform the decrease.
- When the process failed, **reAssigned** will do the opposite:
  - Addition (+): If value reduced lower than 0, increase that result by 10 and remember its carry (**mem -= 1**). Before it was increased with carry, now it requires to be decreased in order to reach the initial state, hence the reduction.
  - Minus (-): If value added larger than 9, minus that result by 10 and remember its carry (**mem += 1**). The same principle can be applied to reach its first start by increasing to its input state.
- This is in order to return the result to its initial state.

#### f. **Multiply:** [Multiply.py](#)

##### i. **isComplete:** `def isComplete(ask):`

For easier coding and less importing between modules, this function is the very same **isComplete** from the above **Backtrack** file section.

```
return ask['index'] == len(ask['list'])
```

##### ii. **get result:** `def get_result(q1, q2, result, col):`

```
def get_result(q1, q2, result, col):
    index_1 = 0
    index_2 = col
    ans = result
    while index_1 <= col and index_2 >= 0:
        ans += q1[index_1] * q2[index_2]
        index_1 += 1
        index_2 -= 1
    return ans
```

- This function includes the following passing parameters:
  - **q1**: Assigned values in row 1.
  - **q2**: Assigned values in row 2.
  - **result**: Result of assigned value.
  - **col**: Current column in check.
- The function will be used in the later component of **Backtrack\_multiply**.
- **index\_1** and **index\_2** store the first and the current column consecutively as initialization, these indices are used to traverse the current location in row 1 **q1** and row 2 **q2**.
- **ans** stores the value to be assigned.
- Perform a while loop in which **index\_1 <= col** and **index\_2 >= 0**, **ans** will add in the multiply result of **q1[index\_1] \* q2[index\_2]**. After which, increase **index\_1** and decrease **index\_2** by 1. Then, return **ans**, which is the resulting multiply we are looking for. For an easier grasp of our work, please refer for the proper explanation at **Code general mechanism** section.

### iii. Backtrack multiply:

```
def Backtrack_multiply(assignment, csp, sign, q1, q2, result):
```

- Inherit from the normal **Backtrack** above, this function is used exclusively for multiply operator with additional changes and fixes here and there in order to solve CSP formula that requires multiplying.
- The function consists of 6 parameters:
  - **assignment**: Assignment which is initialized in **main()** as **asg**.
  - **csp**: CSP variable taken from **CSP\_Class**, initialized in **main()** as **ExeCSP**.
  - **q1**: Assigned values in row 1.
  - **q2**: Assigned values in row 2.
  - **result**: Multiplication result in considering column.
- First, check whether backtrack is complete or not. If it is finished, return True.
 

```
if isComplete(assignment):
    return True
```
- Initialize **var**, **location** and **constraint** similarly to normal backtrack algorithm function above, please refer to the **Backtrack** section.

```

var = assignment['list'][assignment['index']]
location = [int(floor(assignment['index'] / assignment['max_len'])[1]),
            int(assignment['index'] % assignment['max_len'])[1]] # col, row
constraint = csp.get_constraint()

```

- The below components of this function also shares a relatively similar mechanism to the normal **Backtrack** with a few minor changes as shown below.

---

```

# Result position
if assignment['max_len'][1] - location[1] == 2:
    #print(q1, q2)
    tmp_result = int(get_result(q1, q2, result, location[0]) % 10)
    #print('result', tmp_result)
    if tmp_result in csp.get_domain(var):
        # print(csp.get_value(var))
        if (csp.get_value(var) == None and constraint.get_used(tmp_result) == False):
            csp.set_value(var, tmp_result)
            constraint.set_used(tmp_result, True)
            assignment['index'] += 1
            if Backtrack_multiply(assignment, csp, q1, q2, result) == True:
                return True
            assignment['index'] -= 1
            constraint.set_used(tmp_result, False)
            csp.set_value(var, None)
        elif csp.get_value(var) == tmp_result:
            assignment['index'] += 1
            if Backtrack_multiply(assignment, csp, q1, q2, result) == True:
                return True
            assignment['index'] -= 1
    return False

```

- When we are at the result position, using the **get\_result** function to fetch the result of considering column. Mod that value by 10 in order to get the value of the unit position. The rest of this code is the same as the **Backtrack**.

```

if assignment['max_len'][1] - location[1] == 1: # Memorized value position
    remain = int(floor(get_result(q1, q2, result, location[0]) / 10))
    if var == 'c0' and not(remain == 0):
        return False
    csp.set_value(var, remain)
    assignment['index'] += 1
    if Backtrack_multiply(assignment, csp, sign, q1, q2, remain):
        return True
    assignment['index'] -= 1
    csp.set_value(var, None)
    return False

```

- Different from the above **Backtrack**, this function does not determine **mem**, but instead it considers variable **remain** by getting the remaining number, divide the result of that column by 10 and take only values without any digits after the coma.

```

if csp.get_value(var) == None:
    domain = csp.get_domain(var)
    for i in domain:
        # Check whether considering value is appropriate
        if constraint.get_used(i) == False:
            constraint.set_used(i, True)
            if location[1] == 1:
                q1.append(i)
            elif location[1] == 2:
                q2.append(i)
            csp.set_value(var, i)
            assignment['index'] += 1
            if Backtrack_multiply(assignment, csp, sign, q1, q2, result) == True:
                return True
            assignment['index'] -= 1
            if location[1] == 1:
                q1.pop(-1)
            elif location[1] == 2:
                q2.pop(-1)
            csp.set_value(var, None)
            constraint.set_used(i, False)
    else:
        if location[1] == 1:
            q1.append(csp.get_value(var))
        elif location[1] == 2:
            q2.append(csp.get_value(var))
        assignment['index'] += 1
        if Backtrack_multiply(assignment, csp, sign, q1, q2, result) == True:
            return True
        assignment['index'] -= 1
        if location[1] == 1:
            q1.pop(-1)
        elif location[1] == 2:
            q2.pop(-1)
    return False

```

- Since for this multiply exclusive backtrack algorithm, we does not use neither **Assign** nor **reAssign** due to our restriction to use only 2 operands and not using carry number that goes with the normal plus and minus but the one with the principle of multiply. Hence, this affects **assignment['index']** when we try to adjust its index, so instead, we require **q1** and **q2** for 2 operands only to compare and with the built-in pop feature, we can get the values to multiply and get the result we seek for.

### **g. Demonstration:**

This will use test case 2 (level 1). To see all test cases and their corresponding numbers in formula we have tried so far, please refer to the next numeric section.

```

[Running] python -u "d:\Code\Project-1-Cryptarithmic-Problem-in-AI\Project1_AI_Cryptarithmic_Problem.py"
{'c0': 0, 'E': 4, 'A': 5, 'N': 9, 'c1': 0, 'M': 2, 'G': 3, 'c2': 0, 'I': 1, 'P': 0, 'c3': 1, 'c4': 1, 'J': 8, '': 0}
[Done] exited with code=0 in 0.478 seconds

```



```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.19043.1110]
(c) Microsoft Corporation. All rights reserved.

D:\Code\Project-1-Cryptarithmic-Problem-in-AI>python Project1_AI_Cryptarithmic_Problem.py
{'c0': 0, 'E': 4, 'A': 5, 'N': 9, 'c1': 0, 'M': 2, 'G': 3, 'c2': 0, 'I': 1, 'P': 0, 'c3': 1, 'c4': 1, 'J': 8, '': 0}

D:\Code\Project-1-Cryptarithmic-Problem-in-AI>
```

```
input.txt - Notepad
File Edit Format View Help
ANIME+MANGA=JAPAN
```

```
output.txt - Notepad
File Edit Format View Help
54318290
```

#### 4. Test cases:

This algorithm has the complexity of  $O(9^n)$  whereas  $n$  is the amount of variable from input. More variables means more time-consuming the program will take. The most variable number is 9, hence the most complexity should be  $O(9^9)$

##### a. Level 1:

1) AUDIENCE-AUDACITY=AYAYT

32074154-32035768=38386

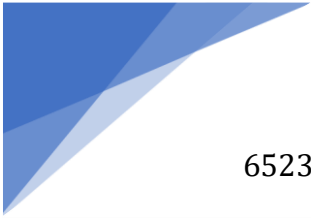
2) ANIME+MANGA=JAPAN

59124+25935=85059

3) COVID+CROWD=VIRUS

38751+36891=75642

4) LOCK+DOWN=WORKS


$$6523+8514=15037$$

$$5) \text{ STUDY}+\text{STUDY}=\text{TIRE D}$$

$$37821+37821=75642$$

**b. Level 2:**

$$1) \text{ BOOM}+\text{SHAKA}+\text{LAKA}=\text{HSBSM}$$

$$8660+13575+9575=31810$$

$$2) \text{ YO}+\text{HOO}+\text{YOLO}+\text{ZALO}+\text{ALO}+\text{OLA}+\text{AHOY}=\text{YOLEE}$$

$$12+322+1202+4502+502+205+5321=12066$$

$$3) \text{ TOO}+\text{MANY}+\text{CHAR}+\text{ROAM}+\text{ON}+\text{CHAT}+\text{MOON}+\text{THAT}+\text{NOON}+\text{OH}+\text{NO}=\text{MNAOT}$$

$$122+3045+6708+8203+24+6701+3224+1701+4224+27+42=34021$$

$$4) \text{ MON}+\text{TUE}+\text{WED}+\text{THU}+\text{FRI}+\text{SAT}+\text{SUN}=\text{WEEK}$$

NO SOLUTION

$$5) \text{ TEAM}+\text{MAKE}+\text{MORE}+\text{CASE}+\text{TO}+\text{TEST}=\text{TASTE}$$

$$1472+2704+2954+8734+19+1431=17314$$

**c. Level 3:**

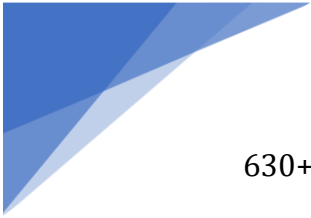
$$1) \text{ ARRAY-BOY}+\text{CANNOT-RUN}=\text{CRURBA}$$

$$12210-340+516647-286=528231$$

$$2) (\text{YOU-CAN}+\text{RUN})+(\text{YOU+CANT-CRY})=\text{CNEO}$$

$$(390-851+601)+(390+8512-863)=8179$$

$$3) \text{ TWO}+\text{DAYS}+\text{TOO}+\text{MANY-SO}-(\text{DAWN-WAY})-\text{YOLO}+\text{LOAD}+\text{MAY-DO}=\text{TWY}$$


$$630+9872+600+1857-20-(9835-387)-7040+4089+187-90=637$$

4) THIS+SIT-IS+SO+TOO-(STILL+STING-LOL+LIST-LOST+LOOT-LIL)-HILL-HIS+SOLISH=HOSTLE

$$4759+954-59+98+488-(94522+94561-282+2594-2894+2884-252)-7522-759+982597=789423$$

5) HAND-(LEG-LAND-LAN-NEGGA+LAG)+HANG-HELL+DANE+(LEND-HEN+NALE+GANG)-GEL=EAGLE

$$9021-(457-4021-402-25770+407)+9027-9544+1025+(4521-952+2045+7027)-754=50745$$

**d. Level 4:**

1) BILI\*LIBI=LIALILII

$$1020*2010=2050200$$

2) KAEYA\*AYAKA=UKYKYOKII

$$12062*26212=316169144$$

3) PARENT\*TREE=TETTUPECT

$$102345*5233=535571385$$


4) CRYPTO\*MATH=MERMYTRMR

$$102345*6748=690624060$$

5) THIS\*ANNOY=AOOTNHAT

$$1023*45567=46615041$$

**5. Reflection and comment:**

- 
- Require adjustment to create constraint more suitably in order to minimize time consumed in order to run the algorithm.
  - The algorithm can be applied to setup value min and max for the result of a column (Has been tried but does not work more efficiently hence discarded).
  - Reflection score:
    - Level 1: 10/10
    - Level 2: 10/10
    - Level 3: 10/10
    - Level 4: 10/10

## **6. References:**

- 2021-Lecture06-CSP.pdf
- 19CLC07-CSC14003-Week05-All.m

