Elaboration of Backtracking Algorithm

1. The algorithm returns a solution or failure.
2. At the very beginning, judge if the assignment is complete. If so, return the assignment as the solution. If not, continue the algorithm.
3. Select the variable to be assigned next according to minimum remaining values algorithm and degree heuristic algorithm.
4. For each value of the variable, in an increasing order, check if the value violates the current assignment.
    - If not, add the variable with this value into the assignment of the problem. Recursively apply the backtracking algorithm. If the result is not failure, then return the result, that is, the complete assignment. If for each value, the result is failure, return failure.

Definition of the Problem

1. Introduce auxiliary variables c1, c2, c3 and c4. Specify their domains as {0, 1}. The four variables represent carry overs from previous columns.

```
   c1  c2  c3  c4
       x1  x2  x3  x4
 +     x5  x6  x7  x8
 ---------------------
   x9 x10 x11 x12 x13
```

2. Set up a set of constraints.

```
 Alldiff(x1, x2, ..., x13)
     x4 + x8 = 10*c4 + x13
c4 + x3 + x7 = 10*c3 + x12
c3 + x2 + x6 = 10*c2 + x11
c2 + x1 + x5 = 10*c1 + x10
          c1 = x9
```

Explanation of Code

Definition of Variables:

1. `char letters[17]` stores 17 variables in the order of: x1, x2, ..., x13, c1, c2, c3, c4.
2. `int assignments[17]` stores value of each variable.
3. `map<char, vector<int>> letter_to_index` The key stores the distinct letter. The value stores the indeices of the letter which may appear more than once.
4. Node represents the variable. Domain represents the domain values. Constraint represents the neighbors of the variable.

```
typedef struct node {
vector<int> domain;
set<char> constraint;
} node;
```

5. `map<char, node*> letter_to_domain_and_constraint` The key stores variables. The value stores the corresponding domain values and neighbors.
6. `bool used[10] = {false}` It denotes whether the value is been used, in order to check the constraint Alldiff.

Definition of Functions:

1. Judge if all the variables are assigned a value. If so, it returns true.

```
bool complete()
```

2. Judge if the current assignment is consistent.

```
bool consistent()
```

3. Choose the variable to be assigned next.

```
char select_unassigned_variable()
```

   - For each unassigned variable:
     - Calculate the number of legal value for each variable. Store the number into variable `domain_size`.
     - Calculate the number of the unassigned neighbors. Store the number into variable `constraint_size`.
     - Implement MRV and degree heuristic algorithm. `minimum_remaining_values` denotes the minimum domain size so far. `var` denotes the variable with the minimum domain size. If there is a variable has same domain size as the previous one, which means that there are two possible variables to be assigned next. So apply degree heuristic algorithm to these two variables. `maximum_degree` denotes the maximum number of unassigned neighbors so far.

```
// MRV
if (domain_size < minimum_remaining_values) {
        var = c;
        minimum_remaining_values = domain_size;
        maximum_degree = constraint_size;
    }
    else if (domain_size == minimum_remaining_values) {
        // Degree Heuristic
        if (constraint_size > maximum_degree) {
            var = c;
            minimum_remaining_values = domain_size;
            maximum_degree = constraint_size;
        }
        else {
```

```
                continue;
            }
        }
```

4. Implement the backtracking process.

```
bool backtrack()
```

   - Judge if the assignmnet is complete. If so, return the solution.

```
if (complete()) return true;
```

   - Select the variable to be assigned next.

```
char var = select_unassigned_variable();
```

   - For each value in `letter_to_domain_and_constraint[var]->domain`, assign the value into assignments[]. Chech if the updated assignment is consistent. If so, backtrack. If the result is true, return true. If the assignment is not consistent, remove the value from the assignment and update the `used[value]` to be false.

Main Function

1. Read the input from the file and store 13 letters into letters[] from index 0 to 12.
2. Initialize the assignments[] to be -1, which means that the variable is not assigned.
3. Store the indices of each variable into letter_to_index.
4. Initialize four auxiliary variables... Store them into letters[] from index 13 to 16. They are denoted as 'a, b, c, d'.
5. Initialize the domain values of the auxiliary variables to be {0, 1}. Store the neighbors of each auxiliary variable into new_node->constraint.
6. Initialize variables x1 to x13.
   - Initialize the domain values. For instance, in the following snippet, store integers 1 to 9 into domain of variables x1 and x5.

```
else if (std::find(indexes.begin(), indexes.end(), 0) != indexes.end() ||
std::find(indexes.begin(), indexes.end(), 4) != indexes.end()) {
            // Domain values of x1 and x5 are 1, 2, ..., 9.
            for (int i = 1; i <= 9; i++) {
                new_node->domain.push_back(i);
            }
        }
```

   - Initialize the neighbors of the variables.
     - Store all 17 variables into new_node->constraint.
     - Remove the corresponding letter of the variable itself and 4 auxiliary variables of the neighbors.
     - Add corresponding auxiliary variables of the variable according to the constraints into new_node->constraint.

7. Call `backtrack()`.
8. Save the solution `assignments[]` into the output file.

Instructions of Compiling and Running the Code

- Input the below instructions in cmd in Windows:

```
g++ -o cryptarithmetic cryptarithmetic.cpp
./cryptarithmetic.exe
```

- If we use a txt file with another name as an input file or output file, we need to modify the name of the input file in the cryptarithmetic.cpp program. For example, if we use 'Input1.txt' as input file and 'Output1.txt' as output file, we need to modify two lines(line 300 and line 482) in the program:

```
ifstream infile("Input1.txt", ios::in);
outfile.open("Output1.txt", ios::out);
```

Output Files of Three Test Input Files

- Output file for the Input1 file:

```
9567
1085
10652
```

- Output file for the Input2 file:

```
7483
7455
14938
```

Source Code

```
#include<iostream>
#include<fstream>
#include<iomanip>
#include<vector>
#include<map>
#include<queue>
#include<cmath>
#include<set>
#include<algorithm>

using namespace std;

/**
 * @brief
```

```
 * Cryptarithmetic problem
 *
 *    c1  c2  c3  c4        <- auxiliary variables
 *        x1  x2  x3  x4
 *  +     x5  x6  x7  x8    <- variables
 * ---------------------
 *    x9 x10 x11 x12 x13
 */

/**
 * @brief
 * Initial Domains:
 *
 * x9: {1}
 * x1, x5: {1, 2, ..., 9}
 * x2-4, x6-8, x10-13: {0, 1, ..., 9}
 * c1-4: {0, 1}
 */

/**
 * @brief
 * Constraints:
 *
 * Alldiff(x1, x2, ..., x13)
 *      x4 + x8 = 10*c4 + x13
 * c4 + x3 + x7 = 10*c3 + x12
 * c3 + x2 + x6 = 10*c2 + x11
 * c2 + x1 + x5 = 10*c1 + x10
 *           c1 = x9
 */

/**
 * @brief
 * "letters" stores the letter of 13 variables and 4 auxiliary variables
 * letters[0-12] -> 13 variables: x1, x2, ..., x13
 * letters[13-16] -> 4 auxiliary variables: c1, c2, c3, c4
 */
char letters[17];

/**
 * @brief
 * "assignments" stores the number of 13 variables and 4 auxiliary variables
 * Default assignment[i] = -1, which means that this variable hasn't been
assigned.
 */
int assignments[17];

/**
 * @brief
 * Store letters and their index <letter, {numbers}>
 */
map<char, vector<int>> letter_to_index;

// The structure of variable, including the domain and constraint, etc.
typedef struct node {
    vector<int> domain;
```

```
    set<char> constraint;
} node;

/**
 * @brief
 * Store letters and their corresponding domain and constraint <letter,
{numbers}>
 */
map<char, node*> letter_to_domain_and_constraint;

/**
 * @brief
 * Judge if the number is assigned
 * true  -> used
 * false -> unused
 */
bool used[10] = {false};

/**
 * @brief
 * Judge if the assignment is complete
 * @return true
 * @return false
 */
bool complete() {
    for (int i = 0; i < 17; i++) {
        if (assignments[i] == -1) {
            return false;
        }
    }
    return true;
}

/**
 * @brief
 * Judge if value is consistent with assignment
 * @return true
 * @return false
 */
bool consistent() {
    // We don't need to check Alldiff(x1, x2, ..., x13) because we have used the
array 'used[10]'

    // c1 = x9
    if (assignments[13] != -1 && assignments[8] != -1) {
        if (assignments[13] != assignments[8]) return false;
    }
    // c2 + x1 + x5 = 10*c1 + x10
    if (assignments[14] != -1 && assignments[0] != -1 && assignments[4] != -1 &&
assignments[13] != -1 && assignments[9] != -1) {
        if (assignments[14] + assignments[0] + assignments[4] !=
10*assignments[13] + assignments[9]) return false;
    }
    // c3 + x2 + x6 = 10*c2 + x11
    if (assignments[15] != -1 && assignments[1] != -1 && assignments[5] != -1 &&
assignments[14] != -1 && assignments[10] != -1) {
```

```
        if (assignments[15] + assignments[1] + assignments[5] !=
10*assignments[14] + assignments[10]) return false;
    }
    // c4 + x3 + x7 = 10*c3 + x12
    if (assignments[16] != -1 && assignments[2] != -1 && assignments[6] != -1 &&
assignments[15] != -1 && assignments[11] != -1) {
        if (assignments[16] + assignments[2] + assignments[6] !=
10*assignments[15] + assignments[11]) return false;
    }
    // x4 + x8 = 10*c4 + x13
    if (assignments[3] != -1 && assignments[7] != -1 && assignments[16] != -1 &&
assignments[12] != -1) {
        if (assignments[3] + assignments[7] != 10*assignments[16] +
assignments[12]) return false;
    }
    return true;
}

/**
 * @brief
 * Implement the function select_unassigned_variable in the algorithm
 * by using the minimum remaining values and degree heuristics.
 * @return char
 */
char select_unassigned_variable() {
    char var = 'A';
    int minimum_remaining_values = 20;
    int maximum_degree = 0;
    for (int i = 0; i < 17; i++) {
        if (assignments[i] != -1) {
            // This variable has been assigned.
            continue;
        }
        else {
            char c = letters[i];

            /**
             * @brief
             * Calculate the number of legal values of domain
             */
            // int domain_size = letter_to_domain_and_constraint[c]-
>domain.size();
            int domain_size = 0;
            if (i >= 13) {
                // Auxiliary variables aren't in Alldiff(x1, x2, ..., x13)
                domain_size = letter_to_domain_and_constraint[c]->domain.size();
            }
            else {
                // Variables
                for (int value : letter_to_domain_and_constraint[c]->domain) {
                    if (used[value] == true) {
                        // If there is some illegal value (used value), just
ignore it.
                        continue;
                    }
                    else {
```

```
                        domain_size++;
                    }
                }
            }

            /**
             * @brief
             * Calculate the number of unassigned neighbors
             */
            // int constraint_size = letter_to_domain_and_constraint[c]-
>constraint.size();
            int constraint_size = 0;
            for (auto it = letter_to_domain_and_constraint[c]-
>constraint.begin(); it != letter_to_domain_and_constraint[c]->constraint.end();
it++) {
                int constraint_index = letter_to_index[*it][0];
                if (assignments[constraint_index] != -1) {
                    // This neighbor has been assigned, just ignore it.
                    continue;
                }
                else {
                    // This neighbor hasn't been assigned yet.
                    constraint_size++;
                }
            }

            // MRV
            if (domain_size < minimum_remaining_values) {
                var = c;
                minimum_remaining_values = domain_size;
                maximum_degree = constraint_size;
            }
            else if (domain_size == minimum_remaining_values) {
                // Degree Heuristic
                if (constraint_size > maximum_degree) {
                    var = c;
                    minimum_remaining_values = domain_size;
                    maximum_degree = constraint_size;
                }
                else {
                    continue;
                }
            }
            else {
                continue;
            }

        }
    }

    return var;
}

bool backtrack() {
    // if assignment is complete then return assignment
    if (complete()) return true;
```

```
    // var <- select_unassigned_variable(csp, assignment)
    char var = select_unassigned_variable();
    // Just judge if the domain is empty. If yes, return false.
    if (letter_to_domain_and_constraint[var]->domain.size() <= 0) return false;
    /**
     * @brief
     * Instead of implementing the least constraining value heuristic
     * in the order_domain_values function, simply order the domain values
     * in increasing order (from lowest to highest).
     */
    sort(letter_to_domain_and_constraint[var]->domain.begin(),
letter_to_domain_and_constraint[var]->domain.end());
    // for each value in order_domain_values(csp, var, assignment) do
    if (var >= 'a' && var <= 'd') {
        // If we choose an auxiliary variable
        for (auto value : letter_to_domain_and_constraint[var]->domain) {
            /**
             * @brief
             * We don't need to remove used value because auxiliary variables are
not in Alldiff(x1, x2, ..., x13)
             */
            // add {var = value} to assignment
            for (int index : letter_to_index[var]) {
                assignments[index] = value;
            }
            // if value is consistent with assignment then
            if (consistent()) {
                // result <- backtrack(csp, assignment)
                bool result = backtrack();
                // if result != failure then return result
                if (result == true) return true;
            }

            // remove {var = value} to assignment
            for (int index : letter_to_index[var]) {
                assignments[index] = -1;
            }
        }
    }
    else {
        // If we choose a variable except auxiliary variables
        for (auto value : letter_to_domain_and_constraint[var]->domain) {
            /**
             * @brief
             * We need to remove used value because of Alldiff(x1, x2, ..., x13)
             */
            // Judge if this value is used
            if (used[value] == true) {
                continue;
            }

            // add {var = value} to assignment
            used[value] = true;
            for (int index : letter_to_index[var]) {
                assignments[index] = value;
            }
```

```
            // if value is consistent with assignment then
            if (consistent()) {
                // result <- backtrack(csp, assignment)
                bool result = backtrack();
                // if result != failure then return result
                if (result == true) return true;
            }
            // remove {var = value} to assignment
            for (int index : letter_to_index[var]) {
                assignments[index] = -1;
            }
            used[value] = false;
        }
    }

    return false;
}



int main() {

    // Read the file
    ifstream infile("Input1.txt", ios::in);
    if (!infile.is_open()) {
        cout << "open error!" << endl;
        return 0;
    }

    /**
     * @brief
     * Read in values from an input text file.
     * The input file contains three rows (or lines) of capital letters:
     *
     * LLLL
     * LLLL
     * LLLLL
     */
    for (int i = 0; i < 13; i++) {
        if (infile.eof()) {
            cout << "read error" << endl;
            return 0;
        }
        // Store variables into "letters"
        infile >> letters[i];
        assignments[i] = -1;
        // Store variables and their corresponding indexes into "m"
        char c = letters[i];
        if (letter_to_index.find(c) == letter_to_index.end()) {
            vector<int> tmp;
            tmp.push_back(i);
            letter_to_index.emplace(c, tmp);
        }
        else {
            letter_to_index[c].push_back(i);
        }
```

```
    }

    /**
     * @brief
     * Store 4 auxiliary variables into "letters" and "assignments"
     */
    for (int i = 13; i < 17; i++) {
        // Store auxiliary variables into "letters"
        letters[i] = 'a' + (i - 13);
        assignments[i] = -1;
        // Store auxiliary variables and their corresponding indexes into "m"
        char c = 'a' + (i - 13);
        vector<int> tmp;
        tmp.push_back(i);
        letter_to_index.emplace(c, tmp);
    }

    /**
     * @brief
     * Store the capital letter and their domain and constraints into "node"
     */
    for (auto item : letter_to_index) {
        char c = item.first;
        vector<int> indexes = item.second;

        node* new_node = new node();

        if (indexes[0] >= 13 && indexes[0] < 17) {
            // Auxiliary variables
            new_node->domain.push_back(0);
            new_node->domain.push_back(1);

            if (indexes[0] == 13) {
                // Auxiliary variables c1
                new_node->constraint.insert(letters[8]); // c1 = x9
                new_node->constraint.insert(letters[14]); // c2 + x1 + x5 = 10*c1
+ x10
                new_node->constraint.insert(letters[0]);
                new_node->constraint.insert(letters[4]);
                new_node->constraint.insert(letters[9]);
            }
            else if (indexes[0] == 14) {
                // Auxiliary variables c2
                new_node->constraint.insert(letters[0]); // c2 + x1 + x5 = 10*c1
+ x10
                new_node->constraint.insert(letters[4]);
                new_node->constraint.insert(letters[9]);
                new_node->constraint.insert(letters[13]);
                new_node->constraint.insert(letters[15]); // c3 + x2 + x6 = 10*c2
+ x11
                new_node->constraint.insert(letters[1]);
                new_node->constraint.insert(letters[5]);
                new_node->constraint.insert(letters[10]);
            }
            else if (indexes[0] == 15) {
                // Auxiliary variables c3
```

```
                new_node->constraint.insert(letters[1]); // c3 + x2 + x6 = 10*c2
+ x11
                new_node->constraint.insert(letters[5]);
                new_node->constraint.insert(letters[14]);
                new_node->constraint.insert(letters[10]);
                new_node->constraint.insert(letters[16]); // c4 + x3 + x7 = 10*c3
+ x12
                new_node->constraint.insert(letters[2]);
                new_node->constraint.insert(letters[6]);
                new_node->constraint.insert(letters[11]);
            }
            else {
                // Auxiliary variables c4
                new_node->constraint.insert(letters[2]); // c4 + x3 + x7 = 10*c3
+ x12
                new_node->constraint.insert(letters[6]);
                new_node->constraint.insert(letters[15]);
                new_node->constraint.insert(letters[11]);
                new_node->constraint.insert(letters[3]); // x4 + x8 = 10*c4 + x13
                new_node->constraint.insert(letters[7]);
                new_node->constraint.insert(letters[12]);
            }
        }
        else {
            // variable x1-13

            // Initialize the domain
            if (std::find(indexes.begin(), indexes.end(), 8) != indexes.end()) {
                // the domain of x9 = {1}
                new_node->domain.push_back(1);
            }
            else if (std::find(indexes.begin(), indexes.end(), 0) !=
indexes.end() || std::find(indexes.begin(), indexes.end(), 4) != indexes.end()) {
                // the domain of x1 and x5 = {1, 2, ..., 9}
                for (int i = 1; i <= 9; i++) {
                    new_node->domain.push_back(i);
                }
            }
            else {
                // the domain of x2-4, x6-8, x10-13 = {0, 1, ..., 9}
                for (int i = 0; i <= 9; i++) {
                    new_node->domain.push_back(i);
                }
            }

            // Initialize the constraint
            for (auto constraint_letter : letter_to_index) {
                new_node->constraint.insert(constraint_letter.first);
            }
            // Erase itself
            new_node->constraint.erase(c);
            // Erase extra auxiliary variable
            new_node->constraint.erase('a');
            new_node->constraint.erase('b');
            new_node->constraint.erase('c');
            new_node->constraint.erase('d');
```

```
            // Add corresponding auxiliary variable to constraint
            if (std::find(indexes.begin(), indexes.end(), 8) != indexes.end()) {
                // c1 = x9
                new_node->constraint.insert(letters[13]);
            }
            if (std::find(indexes.begin(), indexes.end(), 0) != indexes.end()
                  || std::find(indexes.begin(), indexes.end(), 4) !=
indexes.end()
                  || std::find(indexes.begin(), indexes.end(), 9) !=
indexes.end()) {
                // c2 + x1 + x5 = 10*c1 + x10
                new_node->constraint.insert(letters[13]);
                new_node->constraint.insert(letters[14]);
            }
            if (std::find(indexes.begin(), indexes.end(), 1) != indexes.end()
                  || std::find(indexes.begin(), indexes.end(), 5) !=
indexes.end()
                  || std::find(indexes.begin(), indexes.end(), 10) !=
indexes.end()) {
                // c3 + x2 + x6 = 10*c2 + x11
                new_node->constraint.insert(letters[14]);
                new_node->constraint.insert(letters[15]);
            }
            if (std::find(indexes.begin(), indexes.end(), 2) != indexes.end()
                  || std::find(indexes.begin(), indexes.end(), 6) !=
indexes.end()
                  || std::find(indexes.begin(), indexes.end(), 11) !=
indexes.end()) {
                // c4 + x3 + x7 = 10*c3 + x12
                new_node->constraint.insert(letters[15]);
                new_node->constraint.insert(letters[16]);
            }
            if (std::find(indexes.begin(), indexes.end(), 3) != indexes.end()
                  || std::find(indexes.begin(), indexes.end(), 7) !=
indexes.end()
                  || std::find(indexes.begin(), indexes.end(), 12) !=
indexes.end()) {
                // x4 + x8 = 10*c4 + x13
                new_node->constraint.insert(letters[16]);
            }
        }
        letter_to_domain_and_constraint.emplace(c, new_node);
    }

    bool backtracking_search = backtrack();
    if (backtracking_search == false) {
        // We can't find a solution
        cout << "failure" << endl;
        return 0;
    }

    ofstream outfile;
    outfile.open("Output1.txt", ios::out);

    for (int i = 0; i < 4; i++) {
        outfile << assignments[i];
```

```
    }
    outfile << endl;
    for (int i = 4; i < 8; i++) {
        outfile << assignments[i];
    }
    outfile << endl;
    for (int i = 8; i < 13; i++) {
        outfile << assignments[i];
    }
    outfile << endl;

    // Close the file
    outfile.close();

    /**
     * TODO: This is a slight output content in the shell,
     *       which means that the code has receive here.
     *
     *       In the end, we can just delete this line.
     */
    cout << "CSP finished successfully!" << endl;

    return 0;
}
```