N Queens implementation

```
#include<stdio.h>
#include<math.h>

int board[20], count;
int main() {
    int n, i, j;
    void queen(int row, int n);
    printf(" - N Queens Problem Using
      Backtracking - ");
      printf("\n\nEnter number of Queens:"); scanf("%d", & n); queen(1, n);
      return 0;
    }
    //function for printing the solution
    void print(int n) {
      int i, j;
      printf("\n\nSolution %d:\n\n", ++count);
      for (i = 1; i <= n; ++i)
        printf("\t%d", i);
      for (i = 1; i <= n; ++i) {
        printf("\n\n%d", i);
        for (j = 1; j <= n; ++j) //for nxn board
        {
          if (board[i] == j)
            printf("\tQ"); //queen at i,j position
          else
            printf("\t-"); //empty slot
        }
      }
    }
    /*function to check conflicts
    If no conflict for desired position
    returns 1 otherwise returns 0*/
    int place(int row, int column) {
        int i;
        for (i = 1; i <= row - 1; ++i) {
          //checking column and diagonal conflicts
          if (board[i] == column)
            return 0;
          else
          if (abs(board[i] - column) == abs(i - row))
            return 0;
        }
        return 1; //no conflicts
      }
      //function to check for proper positioning
      of queen
    void queen(int row, int n) {
      int column;
      for (column = 1; column <= n; ++column) {
        if (place(row, column)) {
          board[row] = column; //no conflicts so
          place queen
          if (row == n) //dead end
            print(n); //printing the board
          configuration
          else //try queen with next position
            queen(row + 1, n);
        }
      }
    }
```

Graph Colouring implementation

```
class Graph:
    def __init__(self, edges, n):
        self.adjList = [[] for _ in range(n)]
        for (src, dest) in edges:
            self.adjList[src].append(dest)
            self.adjList[dest].append(src)

def colorGraph(graph, n):
    result={}
    for u in range(n):
        assigned = set([result.get(i) for i in graph.adjList[u] if i in result])
        color=1
        for c in assigned:
            if color != c:
                break
            color = color + 1

        result[u] = color

    for v in range(n):
        print(f'Color assigned to vertex {v} is {colors[result[v]]}')


if __name__ == '__main__':

    colors = ['', 'BLUE', 'GREEN', 'RED', 'YELLOW', 'ORANGE', 'PINK',
'BLACK', 'BROWN', 'WHITE', 'PURPLE', 'VOILET']
    edges = [(0, 1), (0, 4), (0, 5), (4, 5), (1, 4), (1, 3), (2, 3), (2, 4)]
    n = 6
    graph = Graph(edges, n)
    colorGraph(graph, n)
```

Constraint Satisfaction implementation

```
import re

solved = False

def solve(letters, values, visited, words):
    global solved
    if len(set) == len(values):
        map = {}
        for letter, val in zip(letters,values):
            map[letter] = val

        if map[words[0][0]] == 0 or map[words[1][0]] == 0 or map[words[2][0]] == 0:
            return

        word1, word2, res = "", "", ""
        for c in words[0]:
            word1 += str(map[c])
        for c in words[1]:
            word2 += str(map[c])
        for c in words[2]:
            res += str(map[c])

        if int(word1) + int(word2) == int(res):
            print("{} + {} = {}\t{}".format(word1, word2, res, map))
            solved = True

        return

    for i in range(10):
        if not visited[i]:
            visited[i] = True
            values.append(i)
            solve(letters, values, visited, words)
            values.pop()
            visited[i] = False


print("\nCRYPTARITHMETIC PUZZLE SOLVER")
print("WORD1 + WORD2 = RESULT")
word1 = input("Enter WORD1: ").upper()
word2 = input("Enter WORD2: ").upper()
result = input("Enter RESULT: ").upper()

if len(result) > (max(len(word1), len(word2)) + 1):
    print("\n0 Solutions!")
else:
    set = []
    for c in word1:
        if c not in set:
            set.append(c)
    for c in word2:
        if c not in set:
            set.append(c)
    for c in result:
        if c not in set:
            set.append(c)

    if len(set) > 10:
        print("\nNo solutions!")
        exit()

    print("Solutions:")
    solve(set, [], [False for _ in range(10)], [word1, word2, result])

    if not solved:
        print("\n0 solutions!")
```

Breadth First Search implementation

```
from collections import defaultdict
class Graph:

	def __init__(self):
		self.graph = defaultdict(list)

	def addEdge(self,u,v):
		self.graph[u].append(v)

	def BFS(self, s):

		visited = [False] * (max(self.graph) + 1)
		queue = []

		queue.append(s)
		visited[s] = True

		while queue:
			s = queue.pop(0)
			print (s, end = " ")
			for i in self.graph[s]:
				if visited[i] == False:
					queue.append(i)
					visited[i] = True

g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)

print ("Following is Breadth First Traversal"
                              " (starting from vertex 2)")
g.BFS(2)
```

Depth First Search implementation

```
from collections import defaultdict
class Graph:
        def __init__(self):
                self.graph = defaultdict(list)

        def addEdge(self, u, v):
                self.graph[u].append(v)

        def DFSUtil(self, v, visited):

                visited.add(v)
                print(v, end=' ')

                for neighbour in self.graph[v]:
                        if neighbour not in visited:
                                self.DFSUtil(neighbour, visited)
        def DFS(self, v):

                visited = set()

                self.DFSUtil(v, visited)

g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)

print("Following is DFS from (starting from vertex 2)")
g.DFS(2)
```

Best First Search implementation

```
from queue import PriorityQueue
v=14
graph=[[] for i in range(v)]

def bfs(source,target,n):
    visited=[0]*n

    pq=PriorityQueue()
    pq.put((0,source))
    while pq.empty()==False:
        u=pq.get()[1]
        print(u,end=" ")
        if u==target:
            break

        for v,c in graph[u]:
            if visited[v]==False:
                visited[v]=True
                pq.put((c,v))

def addedge(x,y,cost):
    graph[x].append((y,cost))
    graph[y].append((x,cost))

addedge(0,1,3)
addedge(0,2,6)
addedge(0,3,5)
addedge(1,4,9)
addedge(1,5,8)
addedge(2,6,12)
addedge(2,7,14)
addedge(3,8,7)
addedge(8,9,5)
addedge(8,10,6)
addedge(9,11,1)
addedge(9,12,10)
addedge(9,13,2)
source=0
target=9
bfs(source,target,v)
```

A * implementation

```
def aStarAlgo(start_node, stop_node):

        open_set = set(start_node)
        closed_set = set()
        g = {}
        parents = {}
        g[start_node] = 0
        parents[start_node] = start_node


        while len(open_set) > 0:
            n = None
            for v in open_set:
                if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):
                    n = v


            if n == stop_node or Graph_nodes[n] == None:
                pass
            else:
                for (m, weight) in get_neighbors(n):
                    if m not in open_set and m not in closed_set:
                        open_set.add(m)
                        parents[m] = n
                        g[m] = g[n] + weight


                    else:
                        if g[m] > g[n] + weight:
                            g[m] = g[n] + weight
                            parents[m] = n

                            if m in closed_set:
                                closed_set.remove(m)
                                open_set.add(m)

            if n == None:
                print('Path does not exist!')
                return None

            if n == stop_node:
                path = []

                while parents[n] != n:
                    path.append(n)
                    n = parents[n]

                path.append(start_node)

                path.reverse()

                print('Path found: {}'.format(path))
                return path


            open_set.remove(n)
            closed_set.add(n)

        print('Path does not exist!')
        return None

def get_neighbors(v):
    if v in Graph_nodes:
        return Graph_nodes[v]
    else:
        return None

def heuristic(n):
        H_dist = {
            'A': 10,
            'B': 6,
            'C': 9,
            'D': 1,
            'E': 7,
            'G': 0,

        }
```

```
        return H_dist[n]

Graph_nodes = {
    'A': [('B', 2), ('E', 3)],
    'B': [('C', 1),('G', 9)],
    'C': None,
    'E': [('G', 6)],
    'G': [('D', 1)],

}
aStarAlgo('A', 'G')
```

Uncertain Method implementation

```
import matplotlib.pyplot as plt

import seaborn; seaborn.set_style('whitegrid')
import numpy

from pomegranate import *

numpy.random.seed(0)
numpy.set_printoptions(suppress=True)

# The guests initial door selection is completely random
guest = DiscreteDistribution({'A': 1./3, 'B': 1./3, 'C': 1./3})

# The door the prize is behind is also completely random
prize = DiscreteDistribution({'A': 1./3, 'B': 1./3, 'C': 1./3})

# Monty is dependent on both the guest and the prize.
monty = ConditionalProbabilityTable(
[[ 'A', 'A', 'A', 0.0 ],
[ 'A', 'A', 'B', 0.5 ],
[ 'A', 'A', 'C', 0.5 ],
[ 'A', 'B', 'A', 0.0 ],
[ 'A', 'B', 'B', 0.0 ],
[ 'A', 'B', 'C', 1.0 ],

[ 'A', 'C', 'A', 0.0 ],
[ 'A', 'C', 'B', 1.0 ],
[ 'A', 'C', 'C', 0.0 ],
[ 'B', 'A', 'A', 0.0 ],
[ 'B', 'A', 'B', 0.0 ],
[ 'B', 'A', 'C', 1.0 ],
[ 'B', 'B', 'A', 0.5 ],
[ 'B', 'B', 'B', 0.0 ],
[ 'B', 'B', 'C', 0.5 ],
[ 'B', 'C', 'A', 1.0 ],
[ 'B', 'C', 'B', 0.0 ],
[ 'B', 'C', 'C', 0.0 ],
[ 'C', 'A', 'A', 0.0 ],
[ 'C', 'A', 'B', 1.0 ],
[ 'C', 'A', 'C', 0.0 ],
[ 'C', 'B', 'A', 1.0 ],
[ 'C', 'B', 'B', 0.0 ],
[ 'C', 'B', 'C', 0.0 ],
[ 'C', 'C', 'A', 0.5 ],
[ 'C', 'C', 'B', 0.5 ],
[ 'C', 'C', 'C', 0.0 ]], [guest, prize])

# State objects hold both the distribution, and a high level name.
s1 = State(guest, name="guest")
s2 = State(prize, name="prize")
s3 = State(monty, name="monty")
# Create the Bayesian network object with a useful name
model = BayesianNetwork("Monty Hall Problem")

# Add the three states to the network
model.add_states(s1, s2, s3)

# Add edges which represent conditional dependencies, where the second node is
# conditionally dependent on the first node (Monty is dependent on both guest and prize)
model.add_edge(s1, s3)
model.add_edge(s2, s3)
model.bake()
model.probability([['A', 'B', 'C']])
model.probability([['A', 'B', 'C']])
print(model.predict_proba({}))
print(model.predict_proba([[None, None, None]]))
print(model.predict_proba([['A', None, None]]))
print(model.predict_proba([{'guest': 'A', 'monty': 'B'}]))
```

Unification implementation

```
def get_index_comma(string):
    index_list = list()
    par_count = 0

    for i in range(len(string)):
        if string[i] == ',' and par_count == 0:
            index_list.append(i)
        elif string[i] == '(':
            par_count += 1
        elif string[i] == ')':
            par_count -= 1

    return index_list


def is_variable(expr):
    for i in expr:
        if i == '(' or i == ')':
            return False

    return True


def process_expression(expr):
    expr = expr.replace(' ', '')
    index = None
    for i in range(len(expr)):
        if expr[i] == '(':
            index = i
            break
    predicate_symbol = expr[:index]
    expr = expr.replace(predicate_symbol, '')
    expr = expr[1:len(expr) - 1]
    arg_list = list()
    indices = get_index_comma(expr)

    if len(indices) == 0:
        arg_list.append(expr)
    else:
        arg_list.append(expr[:indices[0]])
        for i, j in zip(indices, indices[1:]):
            arg_list.append(expr[i + 1:j])
        arg_list.append(expr[indices[len(indices) - 1] + 1:])

    return predicate_symbol, arg_list


def get_arg_list(expr):
    _, arg_list = process_expression(expr)

    flag = True
    while flag:
        flag = False

        for i in arg_list:
            if not is_variable(i):
                flag = True
                _, tmp = process_expression(i)
                for j in tmp:
                    if j not in arg_list:
                        arg_list.append(j)
                arg_list.remove(i)

    return arg_list


def check_occurs(var, expr):
    arg_list = get_arg_list(expr)
```

```
    if var in arg_list:
        return True

    return False


def unify(expr1, expr2):

    if is_variable(expr1) and is_variable(expr2):
        if expr1 == expr2:
            return 'Null'
        else:
            return False
    elif is_variable(expr1) and not is_variable(expr2):
        if check_occurs(expr1, expr2):
            return False
        else:
            tmp = str(expr2) + '/' + str(expr1)
            return tmp
    elif not is_variable(expr1) and is_variable(expr2):
        if check_occurs(expr2, expr1):
            return False
        else:
            tmp = str(expr1) + '/' + str(expr2)
            return tmp
    else:
        predicate_symbol_1, arg_list_1 = process_expression(expr1)
        predicate_symbol_2, arg_list_2 = process_expression(expr2)

        # Step 2
        if predicate_symbol_1 != predicate_symbol_2:
            return False
        # Step 3
        elif len(arg_list_1) != len(arg_list_2):
            return False
        else:
            # Step 4: Create substitution list
            sub_list = list()

            # Step 5:
            for i in range(len(arg_list_1)):
                tmp = unify(arg_list_1[i], arg_list_2[i])

                if not tmp:
                    return False
                elif tmp == 'Null':
                    pass
                else:
                    if type(tmp) == list:
                        for j in tmp:
                            sub_list.append(j)
                    else:
                        sub_list.append(tmp)

            # Step 6
            return sub_list


if _name_ == '_main_':

    f1 = 'Q(a, g(x, a), f(y))'
    f2 = 'Q(a, g(f(b), a), x)'
    # f1 = input('f1 : ')
    # f2 = input('f2 : ')

    result = unify(f1, f2)
    if not result:
        print('The process of Unification failed!')
    else:
        print('The process of Unification successful!')
        print(result)
```

Resolution implementation

```
import copy
import time


class Parameter:
    variable_count = 1

    def _init_(self, name=None):
        if name:
            self.type = "Constant"
            self.name = name
        else:
            self.type = "Variable"
            self.name = "v" + str(Parameter.variable_count)
            Parameter.variable_count += 1

    def isConstant(self):
        return self.type == "Constant"

    def unify(self, type_, name):
        self.type = type_
        self.name = name

    def _eq_(self, other):
        return self.name == other.name

    def _str_(self):
        return self.name


class Predicate:
    def _init_(self, name, params):
        self.name = name
        self.params = params

    def _eq_(self, other):
        return self.name == other.name and all(a == b for a, b in zip(self.params, other.params))

    def _str_(self):
        return self.name + "(" + ",".join(str(x) for x in self.params) + ")"

    def getNegatedPredicate(self):
        return Predicate(negatePredicate(self.name), self.params)


class Sentence:
    sentence_count = 0

    def _init_(self, string):
        self.sentence_index = Sentence.sentence_count
        Sentence.sentence_count += 1
        self.predicates = []
        self.variable_map = {}
        local = {}

        for predicate in string.split("|"):
            name = predicate[:predicate.find("(")]
            params = []

            for param in predicate[predicate.find("(") + 1: predicate.find(")")].split(","):
                if param[0].islower():
                    if param not in local:  # Variable
                        local[param] = Parameter()
                        self.variable_map[local[param].name] = local[param]
                    new_param = local[param]
                else:
                    new_param = Parameter(param)
                    self.variable_map[param] = new_param

                params.append(new_param)
```

```
            self.predicates.append(Predicate(name, params))

    def getPredicates(self):
        return [predicate.name for predicate in self.predicates]

    def findPredicates(self, name):
        return [predicate for predicate in self.predicates if predicate.name == name]

    def removePredicate(self, predicate):
        self.predicates.remove(predicate)
        for key, val in self.variable_map.items():
            if not val:
                self.variable_map.pop(key)

    def containsVariable(self):
        return any(not param.isConstant() for param in self.variable_map.values())

    def _eq_(self, other):
        if len(self.predicates) == 1 and self.predicates[0] == other:
            return True
        return False

    def _str_(self):
        return "".join([str(predicate) for predicate in self.predicates])


class KB:
    def _init_(self, inputSentences):
        self.inputSentences = [x.replace(" ", "") for x in inputSentences]
        self.sentences = []
        self.sentence_map = {}

    def prepareKB(self):
        self.convertSentencesToCNF()
        for sentence_string in self.inputSentences:
            sentence = Sentence(sentence_string)
            for predicate in sentence.getPredicates():
                self.sentence_map[predicate] = self.sentence_map.get(
                    predicate, []) + [sentence]

    def convertSentencesToCNF(self):
        for sentenceIdx in range(len(self.inputSentences)):
            # Do negation of the Premise and add them as literal
            if "=>" in self.inputSentences[sentenceIdx]:
                self.inputSentences[sentenceIdx] = negateAntecedent(
                    self.inputSentences[sentenceIdx])

    def askQueries(self, queryList):
        results = []

        for query in queryList:
            negatedQuery = Sentence(negatePredicate(query.replace(" ", "")))
            negatedPredicate = negatedQuery.predicates[0]
            prev_sentence_map = copy.deepcopy(self.sentence_map)
            self.sentence_map[negatedPredicate.name] = self.sentence_map.get(
                negatedPredicate.name, []) + [negatedQuery]
            self.timeLimit = time.time() + 40

            try:
                result = self.resolve([negatedPredicate], [
                                      False]*(len(self.inputSentences) + 1))
            except:
                result = False

            self.sentence_map = prev_sentence_map

            if result:
                results.append("TRUE")
            else:
                results.append("FALSE")
```

```
        return results

    def resolve(self, queryStack, visited, depth=0):
        if time.time() > self.timeLimit:
            raise Exception
        if queryStack:
            query = queryStack.pop(-1)
            negatedQuery = query.getNegatedPredicate()
            queryPredicateName = negatedQuery.name
            if queryPredicateName not in self.sentence_map:
                return False
            else:
                queryPredicate = negatedQuery
                for kb_sentence in self.sentence_map[queryPredicateName]:
                    if not visited[kb_sentence.sentence_index]:
                        for kbPredicate in kb_sentence.findPredicates(queryPredicateName):

                            canUnify, substitution = performUnification(
                                copy.deepcopy(queryPredicate), copy.deepcopy(kbPredicate))

                            if canUnify:
                                newSentence = copy.deepcopy(kb_sentence)
                                newSentence.removePredicate(kbPredicate)
                                newQueryStack = copy.deepcopy(queryStack)

                                if substitution:
                                    for old, new in substitution.items():
                                        if old in newSentence.variable_map:
                                            parameter = newSentence.variable_map[old]
                                            newSentence.variable_map.pop(old)
                                            parameter.unify(
                                                "Variable" if new[0].islower() else "Constant", new)
                                            newSentence.variable_map[new] = parameter

                                    for predicate in newQueryStack:
                                        for index, param in enumerate(predicate.params):
                                            if param.name in substitution:
                                                new = substitution[param.name]
                                                predicate.params[index].unify(
                                                    "Variable" if new[0].islower() else "Constant",
new)

                                for predicate in newSentence.predicates:
                                    newQueryStack.append(predicate)

                                new_visited = copy.deepcopy(visited)
                                if kb_sentence.containsVariable() and len(kb_sentence.predicates) >
1:

                                    new_visited[kb_sentence.sentence_index] = True

                                if self.resolve(newQueryStack, new_visited, depth + 1):
                                    return True
                return False
        return True


def performUnification(queryPredicate, kbPredicate):
    substitution = {}
    if queryPredicate == kbPredicate:
        return True, {}
    else:
        for query, kb in zip(queryPredicate.params, kbPredicate.params):
            if query == kb:
                continue
            if kb.isConstant():
                if not query.isConstant():
                    if query.name not in substitution:
                        substitution[query.name] = kb.name
                    elif substitution[query.name] != kb.name:
                        return False, {}
                    query.unify("Constant", kb.name)
                else:
```

```
                    return False, {}
            else:
                if not query.isConstant():
                    if kb.name not in substitution:
                        substitution[kb.name] = query.name
                    elif substitution[kb.name] != query.name:
                        return False, {}
                    kb.unify("Variable", query.name)
                else:
                    if kb.name not in substitution:
                        substitution[kb.name] = query.name
                    elif substitution[kb.name] != query.name:
                        return False, {}
    return True, substitution


def negatePredicate(predicate):
    return predicate[1:] if predicate[0] == "" else "" + predicate


def negateAntecedent(sentence):
    antecedent = sentence[:sentence.find("=>")]
    premise = []

    for predicate in antecedent.split("&"):
        premise.append(negatePredicate(predicate))

    premise.append(sentence[sentence.find("=>") + 2:])
    return "|".join(premise)


def getInput(filename):
    with open(filename, "r") as file:
        noOfQueries = int(file.readline().strip())
        inputQueries = [file.readline().strip() for _ in range(noOfQueries)]
        noOfSentences = int(file.readline().strip())
        inputSentences = [file.readline().strip()
                          for _ in range(noOfSentences)]
        return inputQueries, inputSentences


def printOutput(filename, results):
    print(results)
    with open(filename, "w") as file:
        for line in results:
            file.write(line)
            file.write("\n")
    file.close()


if _name_ == '_main_':
    inputQueries_, inputSentences_ = getInput('/home/ubuntu/environment/137/input.txt')
    knowledgeBase = KB(inputSentences_)
    knowledgeBase.prepareKB()
    results_ = knowledgeBase.askQueries(inputQueries_)
    printOutput("output.txt", results_)
```

Crypto Arithmetic implementation

```
import re

solved = False

def solve(letters, values, visited, words):
    global solved
    if len(set) == len(values):
        map = {}
        for letter, val in zip(letters,values):
            map[letter] = val

        if map[words[0][0]] == 0 or map[words[1][0]] == 0 or map[words[2][0]] == 0:
            return

        word1, word2, res = "", "", ""
        for c in words[0]:
            word1 += str(map[c])
        for c in words[1]:
            word2 += str(map[c])
        for c in words[2]:
            res += str(map[c])

        if int(word1) + int(word2) == int(res):
            print("{} + {} = {}\t{}".format(word1, word2, res, map))
            solved = True

        return

    for i in range(10):
        if not visited[i]:
            visited[i] = True
            values.append(i)
            solve(letters, values, visited, words)
            values.pop()
            visited[i] = False


print("\nCRYPTARITHMETIC PUZZLE SOLVER")
print("WORD1 + WORD2 = RESULT")
word1 = input("Enter WORD1: ").upper()
word2 = input("Enter WORD2: ").upper()
result = input("Enter RESULT: ").upper()

if len(result) > (max(len(word1), len(word2)) + 1):
    print("\n0 Solutions!")
else:
    set = []
    for c in word1:
        if c not in set:
            set.append(c)
    for c in word2:
        if c not in set:
            set.append(c)
    for c in result:
        if c not in set:
            set.append(c)

    if len(set) > 10:
        print("\nNo solutions!")
        exit()

    print("Solutions:")
    solve(set, [], [False for _ in range(10)], [word1, word2, result])

    if not solved:
        print("\n0 solutions!")
```