HW3 Project Details Document

1. ***Question 1*** Plain Backtracking (BT):

   - **Code Details:**

```
#do not modify the function names
#You are given L and M as input
#Each of your functions should return the minimum possible L value alongside the marker positions
#Or return -1,[] if no solution exists for the given L

#Your backtracking function implementation
def BT(L, M):
    "*** YOUR CODE HERE ***"

    # The set of Golomb Rulers

    gRuler = Golomb_Ruler(M, L)
    csp_bt = CSPBackTrack(gRuler)

    print("Implementation of CSP for the problem Golomb Ruler by Gourab Bhattacharyya - 170048888 \n")
    print ("Golomb Ruler - M: " + str(gRuler.M) + ", L: " + str(gRuler.L) + "\n")

    result, pathDict = algoTime(csp_bt)

    if result:
        resultList = []
        for key, value in pathDict.iteritems():
            resultList.append(value)
        print "Final Result", (len(pathDict), resultList)
        return (len(pathDict), resultList)
    else:
        print "Final Result", (-1,[])
        return (-1,[])

#Define Golomb Ruler class and set order M and length L for the CSP
class Golomb_Ruler:
    M = 0  # Order
    L = 0  # Length

    def __init__(self, M, L):
        self.M = M
        self.L = L



def timing(f):
    def wrap(*args):
        time1 = time.time()
        ret = f(*args)
        time2 = time.time()
        print('%s function took %0.3f ms' % (f.__name__, (time2 - time1) * 1000.0))
        print("======================end of trace======================")
        #print('\n')
        return ret

    return wrap


@timing
def algoTime(csp):
    print("======================start of trace======================")
    print("Running " + csp.__class__.__name__ + " algorithm")
    m = backtracking_search(csp)
    print("Number of consistency checks: "+str(csp.counter))
    if m is not False:
        for i in range(len(m)):
            print("M" + str(i) + ": " + str(m[i]))

        return True, m
    else:
        print("No solution")
        return False, m
```

```
def backtracking_search(csp):       #This is generic backtrack search
    return backtrack(csp.assignment, csp)


def backtrack(assignment, csp):        #This will take assignment and CSP object as input and do a backtrack through all the nodes

    if csp.complete(assignment): #if the assignment is complete and valid, return the result
        return assignment

    var = csp.selectUnassignedVariables()
    domainVals = csp.findDomainValues(var, assignment)

    for value in domainVals:
        csp.counter += 1                                # increase the counter for each consistency check

        if csp.consistent(var, value, assignment):      #if value is consistent
            assignment[var] = value
            domain = []

            for idx in range(csp.n):
                tempList = []
                for idx in range(csp.d):
                    tempList.append(1)
                domain.append(tempList)

            csp.deepcopyDomain(domain, csp.domain)

            inferences = csp.inference(var, value)

            if inferences is not False:                 #add inferences to assignment
                for inference in inferences:
                    assignment[inference[0]] = inference[1]

                #print "Count, Assignment", csp.counter, assignment
                result = backtrack(assignment, csp)

                if result:
                    return result


            assignment[var] = None                      #remove {var = value} and inferences from assignment

            csp.deepcopyDomain(csp.domain, domain)      #restore the domain values

            if inferences is not False:
                for inference in inferences:
                    assignment[inference[0]] = None
    return False
```

```
class CSPBackTrack:          #This is used for backtrack functionality

    n = 0        #number of variables
    d = 0        #domain of variables
    c=0          #number of constraints
    counter = 0
    consistentDict = dict()


    def __init__(self, ruler):
        self.n = ruler.M
        self.d = ruler.L + 1
        self.domain = []
        self.assignment = dict()

        for order in range(self.n):
            tempList = []
            for item in range(self.d):
                tempList.append(1)
            self.domain.append(tempList)


        for i in range(self.n):
            self.assignment[i] = None



    def complete(self, assignment):      #check for assignment completeness

        for i in range(self.n):
            if assignment[i] == None:
                return False

        return True



    def selectUnassignedVariables(self):        #Find the index of first any unassigned variable

        for i in range(self.n):
            if self.assignment[i] is None:
                return i
```

```
    def findDomainValues(self, item, assignment):        #find possible values in the domain

        possibleValues = []

        if item is not None:
            for i in range(len(self.domain[item])):
                if self.domain[item][i] != 0:
                    possibleValues.append(i)

        return possibleValues


    def consistent(self, item, value, sourceAssignment):        #check is the new assignment is consistent

        assignment = {}
        distance = []

        for i in range(0, len(sourceAssignment)):
            if i == item:
                assignment[i] = value
            else:
                assignment[i] = sourceAssignment[i]


        if tuple(assignment.items()) in self.consistentDict:
            return self.consistentDict[tuple(assignment.items())]


        for i in range(1, len(assignment)):
            if assignment[i] is None or assignment[i-1] is None:
                continue

            if not (assignment[i] > assignment[i-1]):
                self.consistentDict[tuple(assignment.items())] = False
                return False



        for i in range(self.n):
            for j in range(self.n - 1 - i):
                b = assignment[self.n - 1 - i]
                a = assignment[(self.n - 1 - i) - (j + 1)]
                if b is None or a is None:
                    continue

                distance.append(b-a)

        distance.sort()
        temp = None

        for i in range(len(distance)):
            if distance[i] == temp:
                self.consistentDict[tuple(assignment.items())] = False
                return False
            temp = distance[i]


        self.consistentDict[tuple(assignment.items())] = True

        return True



def inference(self, item, value):        #find out the inference for the new assignment

    newVal = set()
    return newVal


def deepcopyDomain(self, domain, values):

    for i in range(len(values)):
        for j in range(len(values[i])):
            domain[i][j] = values[i][j]
```

- The 3 main modules of this Backtracking implementation are as BT(), backtracking_search() and CSPBackTrack.
- BT() module:
  - The first module BT() is the actual thread where the call will begin and move forward.
  - Here, I am first creating a golomb ruler object with the input parameters length and marks.
  - Then calling algoTime() module which in turn calls backtracking module.
  - This method returns pathDict dictionary which holdes the path from the initial variable to the end variable is there is no failure.

- Return the output as expected format [(len(pathDict), resultList)]from this module.

➢ Backtracking_search():
- In this module I am using the CSP assignment and the CSP problem itself to identify the variables that can be assigned with legal values.
- If CSP is complete and consistent then return the assignment else continue.
- Get all the unassigned variables.
- For each value in the domain check for consistency of the assignment.
- If the new value is consistent then add the same in assignment dictionary.
- Call the same module recursively to find out all the vaqlid assignments for the variables.
- Once all the values are identified then set this new value as the domain of the values of the input variables.
- At any point if the assignment is not consistent then return a failure message

➢ CSPBackTrack:
- This is a class module which is getting instantiated with the golomb ruler object.
- For all the new assignment, this module checks for the completeness of the values
- Finds the index of the next unassigned variables
- Find all the possible values that can be available in the domain of a variable
- For a newly assigned variable value this module checks for the consistency and if it's find any issue then returns a failure
- This also check for conflicts of a variable value with its neighbor variable values
- Finally, once a value is found then add that value in the domain

2. **Question 2** BT + Forward Checking (FC):

- **Code Details:**

```
#Your backtracking+Forward checking function implementation
def FC(L, M):
    "*** YOUR CODE HERE ***"

    gRuler = Golomb_Ruler(M, L)
    csp_bt_fc = CSPBTForwardCheck(gRuler)

    print("Implementation of CSP for the problem Golomb Ruler by Gourab Bhattacharyya - 170048888 \n")
    print ("Golomb Ruler - M: " + str(gRuler.M) + ", L: " + str(gRuler.L) + "\n")

    result, pathDict = algoTime(csp_bt_fc)

    if result:
        resultList = []
        for key, value in pathDict.iteritems():
            resultList.append(value)
        print "Final Result", (len(pathDict), resultList)
        return (len(pathDict), resultList)
    else:
        print "Final Result", (-1,[])
        return (-1,[])
```

```python
#Define Golomb Ruler class and set order M and length L for the CSP
class Golomb_Ruler:
    M = 0  # Order
    L = 0  # Length

    def __init__(self, M, L):
        self.M = M
        self.L = L




def timing(f):
    def wrap(*args):
        time1 = time.time()
        ret = f(*args)
        time2 = time.time()
        print('%s function took %0.3f ms' % (f.__name__, (time2 - time1) * 1000.0))
        print("======================end of trace=======================")
        #print('\n')
        return ret

    return wrap


@timing
def algoTime(csp):
    print("====================start of trace====================")
    print("Running " + csp.__class__.__name__ + " algorithm")
    m = backtracking_search(csp)
    print("Number of consistency checks: "+str(csp.counter))
    if m is not False:
        for i in range(len(m)):
            print("M" + str(i) + ": " + str(m[i]))

        return True, m
    else:
        print("No solution")
        return False, m
```

```python
def backtracking_search(csp):       #This is generic backtrack search
    return backtrack(csp.assignment, csp)


def backtrack(assignment, csp):        #This will take assignment and CSP object as input and do a backtrack through all the nodes

    if csp.complete(assignment): #if the assignment is complete and valid, return the result
        return assignment

    var = csp.selectUnassignedVariables()
    domainVals = csp.findDomainValues(var, assignment)

    for value in domainVals:
        csp.counter += 1                                    # increase the counter for each consistency check

        if csp.consistent(var, value, assignment):      #if value is consistent
            assignment[var] = value
            domain = []

            for idx in range(csp.n):
                tempList = []
                for idx in range(csp.d):
                    tempList.append(1)
                domain.append(tempList)

            csp.deepcopyDomain(domain, csp.domain)

            inferences = csp.inference(var, value)

            if inferences is not False:                 #add inferences to assignment
                for inference in inferences:
                    assignment[inference[0]] = inference[1]

                #print "Count, Assignment", csp.counter, assignment
                result = backtrack(assignment, csp)

                if result:
                    return result


            assignment[var] = None                      #remove {var = value} and inferences from assignment

            csp.deepcopyDomain(csp.domain, domain)      #restore the domain values

            if inferences is not False:
                for inference in inferences:
                    assignment[inference[0]] = None
    return False
```

```
class CSPBTForwardCheck(CSPBackTrack):          #This is used for Backtrack with Forward Check

    def __init__(self, ruler):
        self.n = ruler.M
        self.d = ruler.L + 1
        self.domain = []
        self.assignment = dict()

        for order in range(self.n):
            tempList = []
            for item in range(self.d):
                tempList.append(1)
            self.domain.append(tempList)


        # node consistency
        for i in range(self.n):
            temp_list = []
            if i == 0:
                for x in range(self.d):
                    temp_list.append(0)
                temp_list[0] = 1
                self.domain[i] = temp_list

            elif i == self.n - 1:
                for x in range(self.d):
                    temp_list.append(0)
                temp_list[self.d - 1] = 1
                self.domain[i] = temp_list
            else:
                for x in range(self.d):
                    temp_list.append(1)

                for j in range(i + 1, self.n):
                    temp_list[self.d - self.n + j] = 0
                self.domain[i] = temp_list


        # initialize the assignment
        for i in range(self.n):
            self.assignment[i] = None

    def inference(self, item, value):

        inferenced = set()

        for i in range(len(self.domain)):

            if i == item or self.assignment[i] is not None:
                continue


            for j in range(len(self.domain[i])):
                if self.domain[i][j] == 0:
                    continue

                if not (self.consistent(i, j, self.assignment)):
                    self.domain[i][j] = 0


            if self.domain[i].count(1) == 0:
                return False
            elif self.domain[i].count(1) == 1:
                inferenced.add((i, self.domain[i].index(1)))

        return inferenced
```

- The 3 main modules of this Backtracking implementation are as FC(), backtracking_search() and CSPBTForwardCheck.
- FC() module:

- The first module FC() is the actual thread where the call will begin and move forward.
- Here, I am first creating a golomb ruler object with the input parameters length and marks.
- Then calling algoTime() module which in turn calls backtracking module.
- This method returns pathDict dictionary which holdes the path from the initial variable to the end variable is there is no failure.
- Return the output as expected format [(len(pathDict), resultList)]from this module.

➢ Backtracking_search():

- In this module I am using the CSP assignment and the CSP problem itself to identify the variables that can be assigned with legal values.
- If CSP is complete and consistent then return the assignment else continue.
- Get all the unassigned variables.
- For each value in the domain check for consistency of the assignment.
- If the new value is consistent then add the same in assignment dictionary.
- Call the same module recursively to find out all the vaqlid assignments for the variables.
- Once all the values are identified then set this new value as the domain of the values of the input variables.
- At any point if the assignment is not consistent then return a failure message

➢ CSPBTForwardCheck:

- This is a class module which is getting instantiated with the golomb ruler object.
- First, I am specifying a domain for the variables with all 1's
- Then checking for consistency and updating the domain accordingly.
- Initializing the assignment dictionary.
- Initialize a new set for inference values
- If a new value found which is consistent then add the same in the assighnment and then in the domain of values.
- This module will finally return the final domain of the valid variable values

3. **Question 3** BT + Constraint Propagation (CP):

- **Code Details:**

```
#Bonus: backtracking + constraint propagation
def CP(L, M):
    "*** YOUR CODE HERE ***"

    gRuler = Golomb_Ruler(M, L)
    csp_bt_cp = CSPBTConsProp(gRuler)

    print("Implementation of CSP for the problem Golomb Ruler by Gourab Bhattacharyya - 170048888 \n")
    print ("Golomb Ruler - M: " + str(gRuler.M) + ", L: " + str(gRuler.L) + "\n")

    result, pathDict = algoTime(csp_bt_cp)

    if result:
        resultList = []
        for key, value in pathDict.iteritems():
            resultList.append(value)
        print "Final Result", (len(pathDict), resultList)
        return (len(pathDict), resultList)
    else:
        print "Final Result", (-1,[])
        return (-1,[])
```

```python
#Define Golomb Ruler class and set order M and length L for the CSP
class Golomb_Ruler:
    M = 0  # Order
    L = 0  # Length

    def __init__(self, M, L):
        self.M = M
        self.L = L




def timing(f):
    def wrap(*args):
        time1 = time.time()
        ret = f(*args)
        time2 = time.time()
        print('%s function took %0.3f ms' % (f.__name__, (time2 - time1) * 1000.0))
        print("======================end of trace=====================")
        #print('\n')
        return ret

    return wrap


@timing
def algoTime(csp):
    print("====================start of trace====================")
    print("Running " + csp.__class__.__name__ + " algorithm")
    m = backtracking_search(csp)
    print("Number of consistency checks: "+str(csp.counter))
    if m is not False:
        for i in range(len(m)):
            print("M" + str(i) + ": " + str(m[i]))

        return True, m
    else:
        print("No solution")
        return False, m
```

```python
def backtracking_search(csp):       #This is generic backtrack search
    return backtrack(csp.assignment, csp)


def backtrack(assignment, csp):        #This will take assignment and CSP object as input and do a backtrack through all the nodes

    if csp.complete(assignment): #if the assignment is complete and valid, return the result
        return assignment

    var = csp.selectUnassignedVariables()
    domainVals = csp.findDomainValues(var, assignment)

    for value in domainVals:
        csp.counter += 1                                    # increase the counter for each consistency check

        if csp.consistent(var, value, assignment):      #if value is consistent
            assignment[var] = value
            domain = []

            for idx in range(csp.n):
                tempList = []
                for idx in range(csp.d):
                    tempList.append(1)
                domain.append(tempList)

            csp.deepcopyDomain(domain, csp.domain)

            inferences = csp.inference(var, value)

            if inferences is not False:                 #add inferences to assignment
                for inference in inferences:
                    assignment[inference[0]] = inference[1]

                #print "Count, Assignment", csp.counter, assignment
                result = backtrack(assignment, csp)

                if result:
                    return result


            assignment[var] = None                      #remove {var = value} and inferences from assignment

            csp.deepcopyDomain(csp.domain, domain)      #restore the domain values

            if inferences is not False:
                for inference in inferences:
                    assignment[inference[0]] = None
    return False
```

```
class CSPBTConsProp(CSPBTForwardCheck):          #This is used for Backtrack with Constraint Propagation

    def inference(self, item, value):

        queue = Queue.Queue()

        for i in range(len(self.domain)):

            if i != item and self.assignment[i] is None:
                queue.put((i, item))


        return ac_3(self, queue)        #call for contraint propagation
```

```
def ac_3(csp, queue):           #This is used for Constraint Propagation

    while not(queue.empty()):
        (x_i, x_j) = queue.get()
        # if the variable x_i is assigned, we can skip the tuple with certainty
        # because we dont need to change the domain of an assigned variable
        if csp.assignment[x_i] is not None:
            continue
        if _revise(csp, x_i, x_j):
            if csp.domain[x_i].count(1) == 0:
                return False
            s = _neighbors(x_i, csp)
            if x_j in s:
                s.remove(x_j)
            for x_k in s:
                queue.put((x_k, x_i))
    return set() # for consistency of the general backtrack algorithm

def _revise(csp, x_i, x_j):      #if an variable is present in a domain the returns true

    revised = False
    for x in range(len(csp.domain[x_i])):
        #ignore the impossible value
        if csp.domain[x_i][x] == 0:
            continue
        satisfy = False

        #construct the right assignment
        assignment = copy.deepcopy(csp.assignment)
        assignment[x_i] = x
        if csp.assignment[x_j] is not None:
            if csp.consistent(x_j, csp.assignment[x_j], assignment):
                satisfy = True
        else:
            for y in range(len(csp.domain[x_j])):
                # there is some y in D_x_j that satisfy the constraint
                if csp.consistent(x_j, y, assignment):
                    satisfy = True

        if not satisfy:
            #if no value in y in D_j allow (x,y) to satisfy the constraint between X_i and X_j
            csp.domain[x_i][x] = 0
            revised = True

    return revised
```

- The 3 main modules of this Backtracking implementation are as CP(), backtracking_search() and CSPBTConsProp.
- CP() module:
  - The first module CP() is the actual thread where the call will begin and move forward.
  - Here, I am first creating a golomb ruler object with the input parameters length and marks.
  - Then calling algoTime() module which in turn calls backtracking module.
  - This method returns pathDict dictionary which holdes the path from the initial variable to the end variable is there is no failure.
  - Return the output as expected format [(len(pathDict), resultList)]from this module.

- Backtracking_search():
  - In this module I am using the CSP assignment and the CSP problem itself to identify the variables that can be assigned with legal values.
  - If CSP is complete and consistent then return the assignment else continue.
  - Get all the unassigned variables.
  - For each value in the domain check for consistency of the assignment.
  - If the new value is consistent then add the same in assignment dictionary.
  - Call the same module recursively to find out all the vaqlid assignments for the variables.

- Once all the values are identified then set this new value as the domain of the values of the input variables.
- At any point if the assignment is not consistent then return a failure message

- CSPBTConsProp:
  - This is a class module which is getting instantiated with the golomb ruler object.
  - Here I am using a FIFO queue to insert all the item in a queue first and then check for inconsistency and completeness for each variable value.
  - This is calling ac_3 module which is actually performing this check.
  - ac_3() module:
    - for each element in the queue if a value is already assigned to a variable then we will skip that variable and go to the next element.
    - For each variable, we will check the constraint satisfaction with its neighbor.
    - We will ignore the impossible values
    - Construct the correct assignment of variable value
    - If there is some value that satisfy all the constraint then add that value in the domain values.
    - If the value is inconsistent then remove that value from the queue.
    - Finally, this module as a whole will return a set of domain values which are the legal and valid values for the input variables.