MIDDLE EAST TECHNICAL UNIVERSITY

NORTHERN CYPRUS CAMPUS

CNG – 462

Assignment – 3

Time scheduler

Muhammad Somaan – 2528404

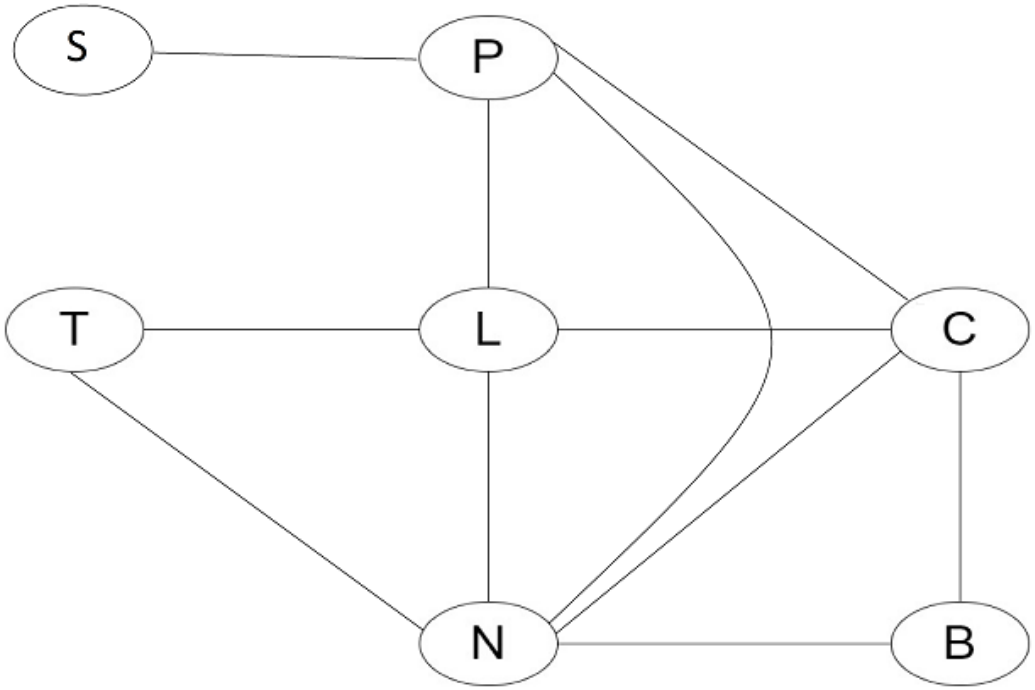
Muhammed Hashir Faraz – 2528545

Submission Date: 11th June 2023

Table of Contents

Part 1: Warming:

1. Constraint Graph



2. Backtracking Search

Step #	T	C	L	N	P	S	B
1	10						
2	10	10					
3	10	10	~~10~~				
4	10	10	11				
5	10	10	11	~~10~~			
6	10	10	11	~~11~~			
7	10	10	11	12			
8	10	10	11	12	~~10~~		
9	10	10	11	12	~~11~~		
10	10	10	11	12	~~12~~		
11	10	10	11	12	13		
12	10	10	11	12	13	10	
13	10	10	11	12	13	10	~~10~~
14	**10**	**10**	**11**	**12**	**13**	**10**	**11**

3. Backtracking with forward checking

Step #	T	C	L	N	P	S	B
1	10	10, 11, 12, 13	~~10~~, 11, 12, 13	~~10~~, 11, 12, 13	10, 11, 12, 13	10, 11, 12, 13	10, 11, 12, 13
2	10	10	11, 12, 13	11, 12, 13	~~10~~, 11, 12, 13	10,11, 12, 13	~~10~~, 11, 12, 13
3	10	10	11	~~11,~~ 12, 13	~~11,~~ 12, 13	10, 11, 12, 13	11, 12, 13
4	10	10	11	12	~~12~~, 13	10, 11, 12, 13	11, ~~12~~, 13
5	10	10	11	12	13	10, 11, 12, ~~13~~	11, 13
6	10	10	11	12	13	10	11,13
7	**10**	**10**	**11**	**12**	**13**	**10**	**11**

Part 2: Programming

Code:

```
import random


from enum import Enum



# List of all our guests. We will use this to create our variables

class Guests(Enum):

    AlanTuring = 1

    AdaLovelace = 2

    NielsBohr = 3

    MarieCurie = 4

    Socrates = 5

    Pythagoras = 6

    IsaacNewton = 7



# HERE


# A single variable in the CSP. Each variable has a domain of possible values.

# The variable's value can be assigned or unassigned.
```

```
# The edges are the variables that are connected to this variable in the constraint graph.
class Variable:
    def _init_(self, name: Guests, domain: list[int]):
        self.name = name
        self.domain = domain
        self.assignment = None
        self.edges = list()



# A constraint for scheduling guests. Each constraint consists of a list of variables,
# because multiple variables can be involved in any given constraint.
class TimeTravelConstraint:
    def _init_(self, constraint_variables: list[Variable]):
        self.constraint_variables = constraint_variables

    # Satifisfied function checks if the constraint is satisfied given the current assignment
    def is_satisfied(self, assignment: dict[Variable, int]):

        assigned_guests = list()

        # checking if all the variable are assigned
        for guest in self.constraint_variables:
            if guest in assignment:
                assigned_guests.append(guest)

        # if only 1 or none of the variable are assigned, then the constraint is satisfied
        if len(assigned_guests) <= 1:
            return True

        is_constraint_satisfied = True
```

```
        # checking between all the assigned variables if they have the same value
        # if they do, then the constraint is not satisfied
        for guest1 in assigned_guests:
            for guest2 in assigned_guests:
                if guest1 != guest2:
                    if assignment[guest1] == assignment[guest2]:
                        is_constraint_satisfied = False

        return is_constraint_satisfied


# function to print the assignments of the variables
def print_assignments(assignments_to_print: dict[Variable, int]):
    to_print = ""
    for guest in assignments_to_print:
        to_print += guest.name.name
        to_print += " is scheduled at "
        to_print += assignments_to_print[guest]._str_()
        to_print += " o'clock, "
    print(to_print)


# Our main class for the CSP
class CSP:
    # variables is a list of variables in the CSP
    # constraints is a map of variables to their constraints
    def _init_(self, variables: list[Variable]):
        self.variables = variables
        self.constraints: dict[Variable, list[TimeTravelConstraint]] = dict()
        for variable in self.variables:
            self.constraints[variable] = list()
```

```
# Add a constraint to the CSP. For each variable in the constraint,
def add_constraint(self, constraint: TimeTravelConstraint):
    for constraint_vars in constraint.constraint_variables:
        self.constraints[constraint_vars].append(constraint)
        # Add the edge to the variable, so we can do forward checking
        for edge in constraint.constraint_variables:
            if edge != constraint_vars:
                if edge not in constraint_vars.edges:
                    constraint_vars.edges.append(edge)

# checking if the constraint is satisfied given the current assignment
def are_constraints_satisfied(self, variable: Variable, assignment: dict[Variable, int]):
    for constraint in self.constraints[variable]:
        if not constraint.is_satisfied(assignment):
            return False
    return True

def backtracking_search(self, assignment: dict[Variable, int]):

    print_assignments(assignment)

    # if the assignment is complete, we're done
    if len(assignment) == len(self.variables):
        return assignment

    # get all variables in the CSP but not in the assignment
    unassigned = [variable for variable in self.variables if variable not in assignment]

    # applying backtrack on the first unassigned variable
    first = unassigned[0]
```

```
    # go through every possible domain value of the first unassigned variable
    for value in first.domain:
        temp_assignment = assignment.copy()
        temp_assignment[first] = value
        first.assignment = value

        # if no constraints are violated, we continue with the next variable
        if self.are_constraints_satisfied(first, temp_assignment):
            result = self.backtracking_search(temp_assignment)
            # if we didn't find the result, we will backtrack
            if result is not None:
                return result
    return None

def backtracking_search_with_forward_check(self, assignment: dict[Variable, int]):

    print_assignments(assignment)

    if len(assignment) == len(self.variables):
        return assignment

    unassigned = [v for v in self.variables if v not in assignment]

    first = unassigned[0]
    for value in first.domain:
        temp_assignment = assignment.copy()
        temp_assignment[first] = value
        first.domain.remove(value)
        first.assignment = value

        # Checking all the edges of the first variable
```

```
        # and removing the value from their domain
        # by doing this we are doing forward checking
        for edge in first.edges:
            if edge not in temp_assignment:
                for edge_value in edge.domain:
                    if edge_value == value:
                        edge.domain.remove(edge_value)

        if self.are_constraints_satisfied(first, temp_assignment):
            result = self.backtracking_search(temp_assignment)
            if result is not None:
                return result
    return None

def backtracking_search_with_degree_heuristic(self, assignment: dict[Variable, int]):

    print_assignments(assignment)

    if len(assignment) == len(self.variables):
        return assignment

    unassigned = [v for v in self.variables if v not in assignment]
    # sort the unassigned variables by the number of edges they have, or their degree.
    unassigned.sort(key=lambda x: len(x.edges), reverse=True)

    first = unassigned[0]
    for value in first.domain:
        temp_assignment = assignment.copy()
        temp_assignment[first] = value
        first.assignment = value
```

```
        if self.are_constraints_satisfied(first, temp_assignment):
            result = self.backtracking_search(temp_assignment)
            if result is not None:
                return result
    return None

def backtracking_search_with_mrv(self, assignment: dict[Variable, int]):

    print_assignments(assignment)

    if len(assignment) == len(self.variables):
        return assignment

    unassigned = [v for v in self.variables if v not in assignment]
    # sort the unassigned variables by the number of values in their domain, or their MRV.
    unassigned.sort(key=lambda x: len(x.domain))

    first = unassigned[0]
    for value in first.domain:
        temp_assignment = assignment.copy()
        temp_assignment[first] = value
        first.assignment = value
        first.domain.remove(value)

        if self.are_constraints_satisfied(first, temp_assignment):
            result = self.backtracking_search(temp_assignment)
            if result is not None:
                return result
    return None

# our function to get the total conflicts of a variable
```

```
def get_conflicts(self, variable: Variable):
    total_conflicts = 0
    if variable.assignment is not None:
        for edge in variable.edges:
            for edge_value in edge.domain:
                if edge_value == variable.assignment:
                    total_conflicts = total_conflicts + 1

    return total_conflicts

def backtracking_search_with_min_conflict(self, assignment: dict[Variable, int], max_steps: int):

    # randomly assign values to all variables
    # which will be our current.
    # randomly selected solution.
    for var in self.variables:
        assignment[var] = random.choice(var.domain)
        var.assignment = assignment[var]
        # print(var.name, var.assignment, self.get_conflicts(var))

    for i in range(max_steps):

        print()
        print("Assignment at iteration ", i, " is: ")
        print_assignments(assignment)

        # all the conflicted variables
        conflicted = [variable for variable in assignment if self.get_conflicts(variable) > 0]

        print("Conflicted variables are: ")
        for v in conflicted:
```

```
            print(v.name.name, v.assignment, ": conflicts=", self.get_conflicts(v))

        if len(conflicted) == 0:
            return assignment

        var = random.choice(conflicted)
        print("Randomly selected conflicted variable is: ", var.name.name)

        min_conflict = self.get_conflicts(var)
        min_conflict_value = var.assignment

        # go through every possible domain value of the conflicted variable
        for value in var.domain:
            var.assignment = value
            conflicts = self.get_conflicts(var)
            if conflicts < min_conflict:
                min_conflict = conflicts
                min_conflict_value = value

        print("Randomly selected conflicted variable's new value : ", var.assignment,
              " with conflicts: ", min_conflict)
        var.assignment = min_conflict_value
        assignment[var] = min_conflict_value

    return None

if _name_ == '_main_':

    choice = 0

    while choice != 7:
```

```
solution = None

turing = Variable(Guests.AlanTuring, [10])
lovelace = Variable(Guests.AdaLovelace, [10, 11, 12, 13])
bohr = Variable(Guests.NielsBohr, [10, 11, 12, 13])
curie = Variable(Guests.MarieCurie, [10, 11, 12, 13])
socrates = Variable(Guests.Socrates, [10, 11, 12, 13])
pythagoras = Variable(Guests.Pythagoras, [10, 11, 12, 13])
newton = Variable(Guests.IsaacNewton, [10, 11, 12, 13])

#my_guests = [turing, lovelace, bohr, curie, socrates, pythagoras, newton]
my_guests = [turing, curie, lovelace, newton, pythagoras, socrates, bohr]
my_csp = CSP(my_guests)

my_csp.add_constraint(TimeTravelConstraint([bohr, curie, newton]))
my_csp.add_constraint(TimeTravelConstraint([lovelace, pythagoras, newton]))
my_csp.add_constraint(TimeTravelConstraint([socrates, pythagoras]))
my_csp.add_constraint(TimeTravelConstraint([lovelace, curie]))
my_csp.add_constraint(TimeTravelConstraint([turing, lovelace, newton]))
my_csp.add_constraint(TimeTravelConstraint([curie, pythagoras]))

print("---------------------------------- METU Time Machine Scheduler --------------------------")
print("Select the scheduling technique:")
print("1. Pure backtracking search algorithm")
print("2. Backtracking search algorithm + forward checking")
print("3. Backtracking search algorithm + Arc Consistency")
print("4. Backtracking search algorithm + Degree Heuristic")
print("5. Backtracking search algorithm + MRV")
print("6. Backtracking search algorithm + Min-Conflict")
print("7. Exit")
```

```
choice = int(input("Enter your choice: "))

if choice == 1:
    solution = my_csp.backtracking_search(dict())
elif choice == 2:
    solution = my_csp.backtracking_search_with_forward_check(dict())
elif choice == 3:
    # backtrackingSearchArcConsistency()
    solution = my_csp.backtracking_search_with_forward_check(dict())
elif choice == 4:
    solution = my_csp.backtracking_search_with_degree_heuristic(dict())
elif choice == 5:
    solution = my_csp.backtracking_search_with_mrv(dict())
elif choice == 6:
    solution = my_csp.backtracking_search_with_min_conflict(dict(), 10)
elif choice == 7:
    print("Exiting the program.")
else:
    print("Invalid choice. Please try again.")

if solution is None:
    print("Failed to find a successful solution!")

print()
```

Part 3: Generalization

Code:

```
import random
```

```
# A single variable in the CSP. Each variable has a domain of possible values.
# The variable's value can be assigned or unassigned.
# The edges are the variables that are connected to this variable in the constraint graph.
class Variable:
    def _init_(self, name, domain):
        self.name = name
        self.domain = domain
        self.assignment = None
        self.edges = list()


# A constraint for scheduling guests. Each constraint consists of a list of variables,
# because multiple variables can be involved in any given constraint.
class Constraint:
    def _init_(self, constraint_variables: list[Variable]):
        self.constraint_variables = constraint_variables

    # Satifisfied function checks if the constraint is satisfied given the current assignment
    def is_satisfied(self, assignment):

        assigned_var = list()

        # checking if all the variable are assigned
        for var in self.constraint_variables:
            if var in assignment:
                assigned_var.append(var)

        # if only 1 or none of the variable are assigned, then the constraint is satisfied
        if len(assigned_var) <= 1:
```

```
            return True

        is_constraint_satisfied = True

        # checking between all the assigned variables if they have the same value
        # if they do, then the constraint is not satisfied
        for var1 in assigned_var:
            for var2 in assigned_var:
                if var1 != var2:
                    if assignment[var1] == assignment[var2]:
                        is_constraint_satisfied = False

        return is_constraint_satisfied



# function to print the assignments of the variables
def print_assignments(assignments_to_print):
    to_print = ""
    for var in assignments_to_print:
        to_print += var.name
        to_print += " is assigned to "
        to_print += assignments_to_print[var]._str_()
        to_print += ", "
    print(to_print)



# Our main class for the CSP
class CSP:
    # variables is a list of variables in the CSP
    # constraints is a map of variables to their constraints
    def _init_(self, variables: list[Variable]):
```

```
    self.variables = variables
    self.constraints: dict[Variable, list[Constraint]] = dict()
    for variable in self.variables:
        self.constraints[variable] = list()

# Add a constraint to the CSP. For each variable in the constraint,
def add_constraint(self, constraint: Constraint):
    for constraint_vars in constraint.constraint_variables:
        self.constraints[constraint_vars].append(constraint)
        # Add the edge to the variable, so we can do forward checking
        for edge in constraint.constraint_variables:
            if edge != constraint_vars:
                if edge not in constraint_vars.edges:
                    constraint_vars.edges.append(edge)

# checking if the constraint is satisfied given the current assignment
def are_constraints_satisfied(self, variable: Variable, assignment):
    for constraint in self.constraints[variable]:
        if not constraint.is_satisfied(assignment):
            return False
    return True

def backtracking_search(self, assignment):

    print_assignments(assignment)

    # if the assignment is complete, we're done
    if len(assignment) == len(self.variables):
        return assignment

    # get all variables in the CSP but not in the assignment
```

```
    unassigned = [variable for variable in self.variables if variable not in assignment]

    # applying backtrack on the first unassigned variable
    first = unassigned[0]
    # go through every possible domain value of the first unassigned variable
    for value in first.domain:
        temp_assignment = assignment.copy()
        temp_assignment[first] = value
        first.assignment = value

        # if no constraints are violated, we continue with the next variable
        if self.are_constraints_satisfied(first, temp_assignment):
            result = self.backtracking_search(temp_assignment)
            # if we didn't find the result, we will backtrack
            if result is not None:
                return result
    return None

def backtracking_search_with_forward_check(self, assignment):

    print_assignments(assignment)

    if len(assignment) == len(self.variables):
        return assignment

    unassigned = [v for v in self.variables if v not in assignment]

    first = unassigned[0]
    for value in first.domain:
        temp_assignment = assignment.copy()
        temp_assignment[first] = value
```

```
            first.domain.remove(value)
            first.assignment = value

            # Checking all the edges of the first variable
            # and removing the value from their domain
            # by doing this we are doing forward checking
            for edge in first.edges:
                if edge not in temp_assignment:
                    for edge_value in edge.domain:
                        if edge_value == value:
                            edge.domain.remove(edge_value)

            if self.are_constraints_satisfied(first, temp_assignment):
                result = self.backtracking_search(temp_assignment)
                if result is not None:
                    return result
        return None

    def backtracking_search_with_degree_heuristic(self, assignment):

        print_assignments(assignment)

        if len(assignment) == len(self.variables):
            return assignment

        unassigned = [v for v in self.variables if v not in assignment]
        # sort the unassigned variables by the number of edges they have, or their degree.
        unassigned.sort(key=lambda x: len(x.edges), reverse=True)

        first = unassigned[0]
        for value in first.domain:
```

```
            temp_assignment = assignment.copy()
            temp_assignment[first] = value
            first.assignment = value

            if self.are_constraints_satisfied(first, temp_assignment):
                result = self.backtracking_search(temp_assignment)
                if result is not None:
                    return result
        return None

    def backtracking_search_with_mrv(self, assignment):

        print_assignments(assignment)

        if len(assignment) == len(self.variables):
            return assignment

        unassigned = [v for v in self.variables if v not in assignment]
        # sort the unassigned variables by the number of values in their domain, or their MRV.
        unassigned.sort(key=lambda x: len(x.domain))

        first = unassigned[0]
        for value in first.domain:
            temp_assignment = assignment.copy()
            temp_assignment[first] = value
            first.assignment = value
            first.domain.remove(value)

            if self.are_constraints_satisfied(first, temp_assignment):
                result = self.backtracking_search(temp_assignment)
                if result is not None:
```

```
            return result
    return None

# our function to get the total conflicts of a variable
def get_conflicts(self, variable: Variable):
    total_conflicts = 0
    if variable.assignment is not None:
        for edge in variable.edges:
            for edge_value in edge.domain:
                if edge_value == variable.assignment:
                    total_conflicts = total_conflicts + 1

    return total_conflicts

def backtracking_search_with_min_conflict(self, assignment, max_steps: int):

    # randomly assign values to all variables
    # which will be our current.
    # randomly selected solution.
    for var in self.variables:
        assignment[var] = random.choice(var.domain)
        var.assignment = assignment[var]
        # print(var.name, var.assignment, self.get_conflicts(var))

    for i in range(max_steps):

        print()
        print("Assignment at iteration ", i, " is: ")
        print_assignments(assignment)

        # all the conflicted variables
```

```
    conflicted = [variable for variable in assignment if self.get_conflicts(variable) > 0]

    print("Conflicted variables are: ")
    for v in conflicted:
        print(v.name.name, v.assignment, ": conflicts=", self.get_conflicts(v))

    if len(conflicted) == 0:
        return assignment

    var = random.choice(conflicted)
    print("Randomly selected conflicted variable is: ", var.name.name)

    min_conflict = self.get_conflicts(var)
    min_conflict_value = var.assignment

    # go through every possible domain value of the conflicted variable
    for value in var.domain:
        var.assignment = value
        conflicts = self.get_conflicts(var)
        if conflicts < min_conflict:
            min_conflict = conflicts
            min_conflict_value = value

    print("Randomly selected conflicted variable's new value : ", var.assignment,
          " with conflicts: ", min_conflict)
    var.assignment = min_conflict_value
    assignment[var] = min_conflict_value

return None
```

```
if _name_ == '_main_':

    user_input = ""
    user_dom_input = ""
    user_cs_input = ""

    variables = list()
    var_dict = dict()

    while user_input != "exit":
        user_input = input("Enter name of variable, enter exit to finish: ")
        if user_input == "exit":
            break
        user_var = Variable(user_input, list())
        while user_dom_input != "exit":
            user_dom_input = input("Enter domain of variable, enter exit to finish: ")
            if user_dom_input == "exit":
                user_dom_input = ""
                break
            user_var.domain.append(user_dom_input)
        variables.append(user_var)
        var_dict[user_input] = user_var

    my_csp = CSP(variables)

    print("Taking input for constraints variables: enter -1 to finish adding all constraints.")

    constraint_vars = list()
    while user_cs_input != "-1":
        user_cs_input = input("Enter constraint variable name, enter exit to finish the current
constraint: ")
        if user_cs_input != "exit" and user_cs_input != "-1":
```

```
        constraint_vars.append(var_dict[user_cs_input])
    if user_cs_input == "exit":
        my_csp.add_constraint(Constraint(constraint_vars))
        constraint_vars = list()


print("Created the constraint satisfaction problem. ")

choice = 0

while choice != 7:

    solution = None

    print("Select the scheduling technique:")
    print("1. Pure backtracking search algorithm")
    print("2. Backtracking search algorithm + forward checking")
    print("3. Backtracking search algorithm + Arc Consistency")
    print("4. Backtracking search algorithm + Degree Heuristic")
    print("5. Backtracking search algorithm + MRV")
    print("6. Backtracking search algorithm + Min-Conflict")
    print("7. Exit")

    choice = int(input("Enter your choice: "))

    if choice == 1:
        solution = my_csp.backtracking_search(dict())
    elif choice == 2:
        solution = my_csp.backtracking_search_with_forward_check(dict())
    elif choice == 3:
        # backtrackingSearchArcConsistency()
```

```
    solution = my_csp.backtracking_search_with_forward_check(dict())
elif choice == 4:
    solution = my_csp.backtracking_search_with_degree_heuristic(dict())
elif choice == 5:
    solution = my_csp.backtracking_search_with_mrv(dict())
elif choice == 6:
    solution = my_csp.backtracking_search_with_min_conflict(dict(), 10)
elif choice == 7:
    print("Exiting the program.")
else:
    print("Invalid choice. Please try again.")

if solution is None:
    print("Failed to find a successful solution!")

print()
```