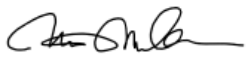
Pair Programming Equitable Participation & Honesty Affidavit

We the undersigned promise that we have in good faith attempted to follow the principles of pair programming. Although we were free to discuss ideas with others, the implementation is our own. We have shared a common workspace and taken turns at the keyboard for the majority of the work that we are submitting. Furthermore, any non programming portions of the assignment were done independently. We recognize that should this not be the case, we will be subject to penalties as outlined in the course syllabus.

Patrick Perrine, 4/15/2020

Pair Programmer 1 (print & sign your name, then date it)



Nicole Miller, 4/15/2020

Pair Programmer 2 (print & sign your name, then date it)

Answers to Part I

Patrick Perrine
Nicole Miller
CS 550
April 15th, 2020

1.

	CA	NV	AZ	UT	CO	NM	TX
Initial	R B Y O	R B Y O	R B Y O	R B Y O	R B Y O	R B Y O	R B Y O
a. After CA=B	B	R Y O	R Y O	R B Y O	R B Y O	R B Y O	R B Y O
b. After AZ =O	B	R Y	O	R B Y	R B Y*	R B Y	R B Y O

*Assuming diagonal states count as adjacent

2.

a. CA = Blue
   Queue: (CA, NV), (CA, AZ)
   Process (CA, NV), remove B from NV's domain
   Process (CA, AZ), remove B from AZ's domain

b. AZ = Orange
   Queue: (AZ, CA), (AZ, NV), (AZ, UT), (AZ, CO), (AZ, NM)
   Process (AZ, CA), remove O from CA's domain
   Process (AZ, NV), remove O from NV's domain
   Process (AZ, UT), remove O from UT's domain
   Process (AZ, CO), remove O from CO's domain*
   Process (AZ, NM), remove O from NM's domain

3.

a. CA = Blue
   Conflict sets:
   NV = {CA = B}
   AZ = {CA = B}

b. AZ = Orange
   Conflict sets:
   CA = {AZ = O}
   NV = {CA = B, AZ = O}
   UT = {AZ = O}
   CO = {AZ = O}*
   NM = {AZ = O}

4.

Assumptions of Problem:

We assume that a specific student is attempting to enroll in a class using the school's enrollment system. The following specification is the data and logic required to allow said student to attempt to enroll in any one of the courses at a time. For a class to be enrolled in, all constraints must hold true. We assume that the scope of this problem is restricted to only the four courses provided. The only possible overlap between courses (1 and 3) was identified and specifically incorporated into the constraints.

a. Variables:

(Course Dependent)

Course Number (CN)

Course Days (CD)

Spaces Left (S)

(Student Dependent)

Courses Enrolled (CE)

Workdays (WD)

Domains:

$CN \in \{1, 2, 3, 4\}$

$CD \in \{\{M, W\}, \{T, TH\}\}$

$S \in \{0, 1, 2…., 29, 30\}$

$CE \in \{Ø, \{1\}, \{2\}, \{3\}, \{4], \{1, 2\}, \{1, 4\}, \{2, 3\}, \{2, 4\}, \{3, 4\}, \{1, 2, 4\}, \{2, 3, 4\}\}$

$WD \in \{Ø, \{M\}, \{T\}, \{W\}, \{Th\}, \{M, T\}, \{M, W\}, \{M, Th\}, \{T, W\}, \{T, Th\}, \{W, Th\}, \{M, T, W\}, \{M, T, Th\}, \{M, W, Th\}, \{T, W, Th\}, \{M, T, W, Th\}\}$

Constraints:

(Course Overlap)

If $1 \in CE$, then $CN \neq 3$

If $3 \in CE$, then $CN \neq 1$

(Workday Conflicts)

$\forall x \in CD, x \notin WD$

(Seat Availability)

$S \neq 0$

b. We assume that the question is asking for an encapsulated set (E) of valid enrollment configurations for a student. Given that the domains of variables WD and S would create an unreasonable amount of possibilities to list, we become only concerned with the constraint of Course Overlapping, specifically with courses 1 and 3. This makes the encapsulation set equal to the domain of the Courses Enrolled (CE) variable defined previously.

$$E = \{Ø, \{1\}, \{2\}, \{3\}, \{4\}, \{1, 2\}, \{1, 4\}, \{2, 3\}, \{2, 4\}, \{3, 4\}, \{1, 2, 4\}, \{2, 3, 4\}\}$$

$$|E| = 12 \text{ valid schedules}$$

Note: Any possible combination with courses 1 and 3 are absent from this set.

```
"""
Filename: constraint_prop.py

Contains the AC3 function for the Sudoku AI.

Implements the AC-3 Arc Consistency Algorithm from the
Russell & Norvig text. This function allows for a given
CSP object to obtain network arc consistency.

Also contains a helper function, revise, to prune any
pairs in which the constraint is not held.

CS 550, Spring 2020, Marie Roch
@author: mroch, nmill, pperr
"""


def AC3(csp, queue=None, removals=None):
    """AC3 constraint propagation"""

    # If there is no given queue, populate the queue using
    # the neighbors list from the CSP
    if not queue:
        queue = []
        for i in csp.neighbors:
            for j in csp.neighbors[i]:
                queue.append((i, j))

    # Initialize an empty removals list if not given
    if not removals:
        removals = []

    # Iterate through the queue to call revise on each binary constraint.
    while queue:
        (xi, xj) = queue.pop()  # get binary constraint

        # Return false if Xi is found to have no elements in curr_domains
        # after revise() is called.
        if revise(csp, xi, xj, removals):
            if not csp.curr_domains[xi]:
                return False
            # If revise() returns True and there are elements in curr_domains,
            # Remove Xj from Xi's neighbors, then add binary constraint (Xk, Xi)
            # to the queue for each Xk in Xi, add Xj back to Xi's neighbors
            else:
```

```
                csp.neighbors[xi].remove(xj)
                for xk in csp.neighbors[xi]:
                    queue.append((xk, xi))
                csp.neighbors[xi].add(xj)
    return True


def revise(csp, xi, xj, removals):
    """AC3 revise function"""

    revised = False

    # For each domain value of Xi, verify the constraint holds with
    # at least 1 domain value in Xj. If not, delete the value from Xi's domain
    for x in csp.curr_domains[xi][:]:
        conHeld = False
        for y in csp.curr_domains[xj][:]:
            conHeld = csp.constraints(xi, x, xj, y)
            if conHeld:
                break
        # Prune any domain value of Xi in which the constraint is not held.
        if not conHeld:
            csp.prune(xi, x, removals)
            revised = True
    return revised
```

```
"""
Filename: backtrack.py

Contains the backtracking search function for the Sudoku AI.

Implements the Backtracking Search Algorithm from the
Russell & Norvig text. It is modular in that it allows for differing
functions for selecting unassigned variables, ordering domain
values, and inference finding to be passed as parameters.

CS 550, Spring 2020, Marie Roch
@author: mroch, nmill, pperr
"""

from csp_lib.backtrack_util import (first_unassigned_variable,
                                    unordered_domain_values,
                                    no_inference)


def backtracking_search(csp,
                        select_unassigned_variable=first_unassigned_variable,
                        order_domain_values=unordered_domain_values,
                        inference=no_inference):
    """backtracking_search
    Given a constraint satisfaction problem (CSP),
    a function handle for selecting variables,
    a function handle for selecting elements of a domain,
    and a set of inferences, solve the CSP using backtrack search"""

    def backtrack(assignment):
        """Attempt to backtrack search with current assignment
        Returns None if there is no solution.  Otherwise, the
        csp should be in a goal state.
        """

        # Checks if the assignment is already complete
        if len(assignment) == 81:
            return assignment

        # Selects unassigned variable
        var = select_unassigned_variable(assignment, csp)
        if var is None:
            return assignment

        # For every domain value of the unassigned variable,
```

```
    # determine the conflicts, create the list of inferences,
    # and remove any that fail the constraint
    for value in order_domain_values(var, assignment, csp):
        if csp.nconflicts(var, value, assignment) == 0:
            csp.assign(var, value, assignment)

            # propagate new constraints
            removals = csp.suppose(var, value)
            inferences = inference(csp, var, value, assignment, removals)

            # If inferences are found, determine a partial assignment,
            # add it to the assignment, and call backtrack recursively
            # on the new assignment
            if inferences:
                infer = csp.infer_assignment()[var]
                csp.assign(var, infer, assignment)
                curr_result = backtrack(assignment)

                # If the assignment was successful, return it.
                # Else, continue.
                if curr_result:
                    return curr_result

            # Value inconsistent / further exploration failed.
            # Restore assignment to its state at top of loop
            # and try next value (continue loop)
            csp.unassign(var, assignment)
            csp.restore(removals)

    # If no value was consistent with the , return none
    return None

# Call with empty assignments, variables accessed
# through dynamic scoping (variables in outer
# scope can be accessed in Python)
result = backtrack({})

# Determine the validity of the generated assignment
# If a valid solution or no solution found, return result respectively
success = csp.goal_test(result)
assert result is None or csp.goal_test(result)
return result
```

```
"""
Filename: driver.py

Driver class for the Sudoku AI

Demonstrates functionality of the Sudoku class, the backtracking
search function, and the AC3 function. Contains an easy and hard
puzzle for the AI to solve and then display results.

CS 550, Spring 2020, Marie Roch
@author: mroch, nmill, pperr
"""
from csp_lib.sudoku import (Sudoku, easy1, harder1)
from constraint_prop import AC3
from csp_lib.backtrack_util import (mrv, lcv, forward_checking, mac)
from backtrack import backtracking_search

# Test both puzzles with Sudoku AI
for puzzle in [easy1, harder1]:
    print("")
    print("Initial sudoku state:")
    s = Sudoku(puzzle)  # Construct a Sudoku problem
    s.display(s.infer_assignment())  # Display initial puzzle state
    print("")

    # Utilize AC3 algorithm to solve puzzle
    AC3(s)

    print("After AC3 constraint propagation:")
    # Display solved puzzle
    s.display(s.infer_assignment())

    # If AC3 fails, run backtracking search to solve puzzle
    if not s.goal_test(s.curr_domains):
        print("")
        print("AC3 constraint propagation failed.")
        print("Running a backtracking search to solve puzzle.")
        solution = backtracking_search(s, select_unassigned_variable=mrv,
                                       inference=mac)
        if solution:
            print("Backtracking search successful:")
        else:
            print("Backtracking search failed.")

        s.display(s.infer_assignment())
```

Code Output:

```
Initial sudoku state:
. . 3 | . 2 . | 6 . .
9 . . | 3 . 5 | . . 1
. . 1 | 8 . 6 | 4 . .
------+-------+------
. . 8 | 1 . 2 | 9 . .
7 . . | . . . | . . 8
. . 6 | 7 . 8 | 2 . .
------+-------+------
. . 2 | 6 . 9 | 5 . .
8 . . | 2 . 3 | . . 9
. . 5 | . 1 . | 3 . .

After AC3 constraint propagation:
4 8 3 | 9 2 1 | 6 5 7
9 6 7 | 3 4 5 | 8 2 1
2 5 1 | 8 7 6 | 4 9 3
------+-------+------
5 4 8 | 1 3 2 | 9 7 6
7 2 9 | 5 6 4 | 1 3 8
1 3 6 | 7 9 8 | 2 4 5
------+-------+------
3 7 2 | 6 8 9 | 5 1 4
8 1 4 | 2 5 3 | 7 6 9
6 9 5 | 4 1 7 | 3 8 2

Initial sudoku state:
4 1 7 | 3 6 9 | 8 . 5
. 3 . | . . . | . . .
. . . | 7 . . | . . .
------+-------+------
. 2 . | . . . | . 6 .
. . . | . 8 . | 4 . .
. . . | . 1 . | . . .
------+-------+------
. . . | 6 . 3 | . 7 .
5 . . | 2 . . | . . .
1 . 4 | . . . | . . .
```

```
After AC3 constraint propagation:
4 1 7 | 3 6 9 | 8 2 5
. 3 . | . . . | . . .
. . . | 7 . . | . . .
------+-------+------
. 2 . | . . . | . 6 .
. . . | . 8 . | 4 . .
. . . | . 1 . | . . .
------+-------+------
. . . | 6 . 3 | . 7 .
5 . . | 2 . . | . . .
1 . 4 | . . . | . . .

AC3 constraint propagation failed.
Running a backtracking search to solve puzzle.
Backtracking search successful:
4 1 7 | 3 6 9 | 8 2 5
6 3 2 | 1 5 8 | 9 4 7
9 5 8 | 7 2 4 | 3 1 6
------+-------+------
8 2 5 | 4 3 7 | 1 6 9
7 9 1 | 5 8 6 | 4 3 2
3 4 6 | 9 1 2 | 7 5 8
------+-------+------
2 8 9 | 6 4 3 | 5 7 1
5 7 3 | 2 9 1 | 6 8 4
1 6 4 | 8 7 5 | 2 9 3
```