Constraints Satisfaction Problems

October 26, 2022

0.0.1 Solving a Problem with Constraints

We have already discussed in class that how CSPs are formulated. Let's apply them to a real-world problem. In this problem, we have a list of names and each name can take a fixed set of values. We also have a set of constraints between these people that needs to be satisfied. Let's see how to do it.

SimpleAI provides you with a class that you will instantiate to represent your csp problems, and a few csp algorithms that you can use to find solutions for the csp problems.

Defining your problem You must simply create an instance of this class, specifying the variables, the variable domains, and the constraints as construction parameters:

```
variables will be a tuple with the variable names.
domains will be a dictionary with the variable names as keys, and the domains as values (in the
constraints will be a list of tuples with two components each: a tuple with the variables invol
```

FAQ why not merge variables and domains on one single dict?

```
Answer: because we need to preserve the order of the variables, and dicts don't have order. We
```

The constraint functions will receive two parameters to check the constraint: a variables tuple and a values tuple, both containing only the restricted variables and their values, and in the same order than the constrained variables tuple you provided. The function should return True if the values are "correct" (no constraint violation detected), or False if the constraint is violated (think this functions as answers to the question "can I use this values?").

We will illustrate with a simple example that tries to assign numbers to 3 variables (letters), but with a few restrictions.

Example:

[59]:
```python
from simpleai.search import CspProblem

variables = ('A', 'B', 'C')

domains = {
    'A': [1, 2, 3],
    'B': [1, 3],
    'C': [1, 2],
}
```

```python
# a constraint that expects different variables to have different values
def const_different(variables, values):
    return len(values) == len(set(values))  # remove repeated values and count

# a constraint that expects one variable to be bigger than other
def const_one_bigger_other(variables, values):
    return values[0] > values[1]

# a constraint thet expects two variables to be one odd and the other even,
# no matter which one is which type
def const_one_odd_one_even(variables, values):
    if values[0] % 2 == 0:
        return values[1] % 2 == 1  # first even, expect second to be odd
    else:
        return values[1] % 2 == 0  # first odd, expect second to be even

constraints = [
    (('A', 'B', 'C'), const_different),
    (('A', 'C'), const_one_bigger_other),
    (('A', 'C'), const_one_odd_one_even),
]

my_problem = CspProblem(variables, domains, constraints)
```

Searching for solutions Now, with your csp problem instantiated, you can call the csp search algorithms. They are located on the "simpleai.search" package.

For example, if you want to use backtracking search, you would do:

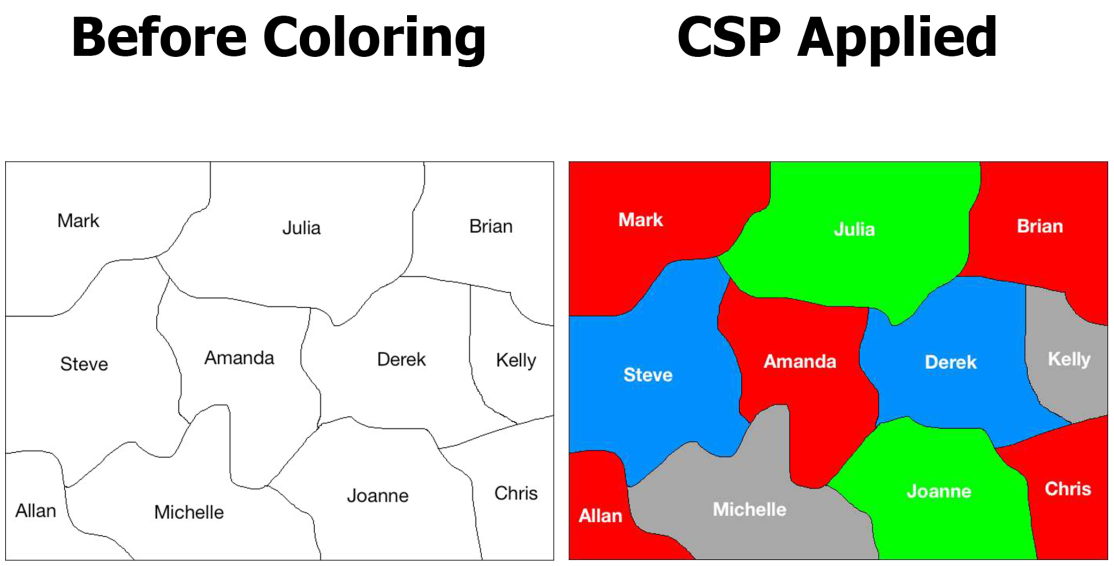
[60]:
```python
from simpleai.search import backtrack

# my_problem = ... (steps from the previous section)

result = backtrack(my_problem)
```

The result will be a dictionary with the assigned values to the variables if a solution was found, or None if couldn't find a solution.

[61]:
```python
print(result)
```

```
{'A': 2, 'B': 3, 'C': 1}
```

0.0.2 Coloring Example



[9]:
```python
from simpleai.search import CspProblem, backtrack


# Define the function that imposes the constraint
# that neighbors should be different
def constraint_func(names, values):
    return values[0] != values[1]


if __name__ == '__main__':
    # Specify the variables
    names = ('Mark', 'Julia', 'Brian', 'Steve', 'Amanda', 'Derek', 'Kelly',␣
 ↪'Allan', 'Michelle', 'Joanne', 'Chris')

    # Define the possible colors
    colors = dict((name, ['red', 'green', 'blue', 'gray']) for name in names)

    # Define the constraints
    constraints = [
#         Mark adjacent
        (('Mark', 'Julia'), constraint_func),
        (('Mark', 'Steve'), constraint_func),
#         Julia adjacent
        (('Julia', 'Mark'), constraint_func),
        (('Julia', 'Steve'), constraint_func),
        (('Julia', 'Amanda'), constraint_func),
        (('Julia', 'Derek'), constraint_func),
        (('Julia', 'Brian'), constraint_func),
#         Brian adjacent
        (('Brian', 'Julia'), constraint_func),
        (('Brian', 'Derek'), constraint_func),
        (('Brian', 'Kelly'), constraint_func),
#         Steve adjacent
        (('Steve', 'Mark'), constraint_func),
        (('Steve', 'Julia'), constraint_func),
        (('Steve', 'Amanda'), constraint_func),
        (('Steve', 'Allan'), constraint_func),
        (('Steve', 'Michelle'), constraint_func),
#         Amanda adjacent
        (('Amanda', 'Steve'), constraint_func),
        (('Amanda', 'Julia'), constraint_func),
        (('Amanda', 'Michelle'), constraint_func),
        (('Amanda', 'Joanne'), constraint_func),
        (('Amanda', 'Derek'), constraint_func),
#         Derek adjacent
        (('Derek', 'Julia'), constraint_func),
        (('Derek', 'Joanne'), constraint_func),
        (('Derek', 'Kelly'), constraint_func),
        (('Derek', 'Chris'), constraint_func),
        (('Derek', 'Brian'), constraint_func),
#         Kelly adjacent
        (('Kelly', 'Brian'), constraint_func),
        (('Kelly', 'Derek'), constraint_func),
        (('Kelly', 'Chris'), constraint_func),
#         Allan adjacent
        (('Allan', 'Steve'), constraint_func),
        (('Allan', 'Michelle'), constraint_func),
#         Michelle adjacent
        (('Michelle', 'Steve'), constraint_func),
        (('Michelle', 'Amanda'), constraint_func),
        (('Michelle', 'Joanne'), constraint_func),
#         Joanne adjacent
        (('Joanne', 'Amanda'), constraint_func),
        (('Joanne', 'Michelle'), constraint_func),
        (('Joanne', 'Derek'), constraint_func),
        (('Joanne', 'Chris'), constraint_func),
#         Chris adjacent
        (('Chris', 'Joanne'), constraint_func),
        (('Chris', 'Derek'), constraint_func),
        (('Chris', 'Kelly'), constraint_func),

    ]

    # Solve the problem
    problem = CspProblem(names, colors, constraints)

    # Print the solution
    output = backtrack(problem)
    print('\nColor mapping:\n')
    for k, v in output.items():
        print(k, '==>', v)
```

```
Color mapping:

Mark ==> red
Julia ==> green
Brian ==> red
Steve ==> blue
Amanda ==> red
Derek ==> blue
Kelly ==> green
Allan ==> red
Michelle ==> green
Joanne ==> gray
Chris ==> red
```

0.0.3 Another Example

[11]:
```
from simpleai.search import CspProblem, backtrack, \
    min_conflicts, MOST_CONSTRAINED_VARIABLE, \
    HIGHEST_DEGREE_VARIABLE, LEAST_CONSTRAINING_VALUE


# Constraint that expects all the different variables
# to have different values
def constraint_unique(variables, values):
    # Check if all the values are unique
    return len(values) == len(set(values))


# Constraint that specifies that one variable
# should be bigger than other
def constraint_bigger(variables, values):
    return values[0] > values[1]
```

```python
# Constraint that specifies that there should be
# one odd and one even variables in the two variables
def constraint_odd_even(variables, values):
    # If first variable is even, then second should
    # be odd and vice versa
    if values[0] % 2 == 0:
        return values[1] % 2 == 1
    else:
        return values[1] % 2 == 0


if __name__ == '__main__':
    variables = ('John', 'Anna', 'Tom', 'Patricia')

    domains = {
        'John': [1, 2, 3],
        'Anna': [1, 3],
        'Tom': [2, 4],
        'Patricia': [2, 3, 4],
    }

    constraints = [
        (('John', 'Anna', 'Tom'), constraint_unique),
        (('Tom', 'Anna'), constraint_bigger),
        (('John', 'Patricia'), constraint_odd_even),
    ]

    problem = CspProblem(variables, domains, constraints)

    print('\nSolutions:\n\nNormal:', backtrack(problem))
    print('\nMost constrained variable:', backtrack(problem,␣
 ↪variable_heuristic=MOST_CONSTRAINED_VARIABLE))
    print('\nHighest degree variable:', backtrack(problem,␣
 ↪variable_heuristic=HIGHEST_DEGREE_VARIABLE))
    print('\nLeast constraining value:', backtrack(problem,␣
 ↪value_heuristic=LEAST_CONSTRAINING_VALUE))
    print('\nMost constrained variable and least constraining value:',
          backtrack(problem, variable_heuristic=MOST_CONSTRAINED_VARIABLE,␣
 ↪value_heuristic=LEAST_CONSTRAINING_VALUE))
    print('\nHighest degree and least constraining value:',
          backtrack(problem, variable_heuristic=HIGHEST_DEGREE_VARIABLE,␣
 ↪value_heuristic=LEAST_CONSTRAINING_VALUE))
    print('\nMinimum conflicts:', min_conflicts(problem))
```

```
Solutions:
```

```
Normal: {'John': 1, 'Anna': 3, 'Tom': 4, 'Patricia': 2}

Most constrained variable: {'Anna': 1, 'Tom': 2, 'John': 3, 'Patricia': 2}

Highest degree variable: {'John': 1, 'Anna': 3, 'Tom': 4, 'Patricia': 2}

Least constraining value: {'John': 1, 'Anna': 3, 'Tom': 4, 'Patricia': 2}

Most constrained variable and least constraining value: {'Anna': 1, 'Tom': 2,
'John': 3, 'Patricia': 2}

Highest degree and least constraining value: {'John': 1, 'Anna': 3, 'Tom': 4,
'Patricia': 2}

Minimum conflicts: {'John': 2, 'Anna': 3, 'Tom': 4, 'Patricia': 3}
```

The implemented algorithms are: simpleai.search.csp.backtrack(problem, variable_heuristic='', value_heuristic='', inference=True)

Backtracking search.

variable_heuristic is the heuristic for variable choosing, can be MOST_CONSTRAINED_VARIABLE, HIGHEST_DEGREE_VARIABLE, or blank for simple ordered choosing. value_heuristic is the heuristic for value choosing, can be LEAST_CONSTRAINING_VALUE or blank for simple ordered choosing.

simpleai.search.csp.convert_to_binary(variables, domains, constraints)

Returns new constraint list, all binary, using hidden variables.

You can use it as previous step when creating a problem.

simpleai.search.csp.min_conflicts(problem, initial_assignment=None, iterations_limit=0)

Min conflicts search.

initial_assignment the initial assignment, or None to generate a random one. If iterations_limit is specified, the algorithm will end after that number of iterations. Else, it will continue until if finds an assignment that doesn't generate conflicts (a solution).

Using heuristics The backtrack algorithm allows the use of generic heuristics for variable and value selections. In the help of the function are listed the available heuristics for each one, and to use them you must just import them from the same simpleai.search package.

0.0.4 Another Example

You have been hired by GIFT University to assist in the management of the timetable of the classes. Your first task is managing the timetable of the CS department's new Data Science BS program. The classes need to be held weekly on three days, that is Mondays, Wednesdays, and Fridays only.

There would be a total of 5 lectures to be held on these days and a total of 3 newly hired Ph.D. teachers would be teaching these classes. You are limited by the fact that each teacher can only teach one class at a time.

The lectures to be scheduled are:

- Lecture 1 – Data Science Fundamentals: 8:00-9:00am
- Lecture 2 – Programming for Data Science: 8:30-9:30am
- Lecture 3 – Statistics for Data Science: 9:00-10:00am
- Lecture 4 – Fundamentals of AI: 9:00-10:00am
- Lecture 5 – Problem Solving and Machine Learning: 9:30-10:30am

The teachers who would be taking these lectures and their lecture preferences would be:

- Ahmed, who would be teaching Lectures 3 and 4.
- Jawad, who would be teaching Lectures 2, 3, 4, and 5, and
- Kareem, who would be teaching Lectures 1, 2, 3, 4, 5.

[12]:
```python
from simpleai.search import CspProblem, backtrack, convert_to_binary

variable = ('L1', 'L2', 'L3', 'L4', 'L5')

def const(variable, values):
    return values[0] != values[1]

domain = {
    'L1': ['Kareem'],
    'L2': ['Jawad', 'Kareem'],
    'L3': ['Ahmed', 'Jawad', 'Kareem'],
    'L4': ['Ahmed', 'Jawad', 'Kareem'],
    'L5': ['Jawad', 'Kareem'],
}


constraint = [
    (('L1', 'L2'), const),
    (('L2', 'L3'), const),
    (('L2', 'L4'), const),
    (('L3', 'L4'), const),
    (('L3', 'L5'), const),
    (('L4', 'L5'), const),
]
```

```
variable, domain, constraint = convert_to_binary(variables=variable,␣
 ↪domains=domain, constraints=constraint)

problem = CspProblem(variables=variable, domains=domain, constraints=constraint)

result = backtrack(problem)

print(result)
```

```
{'L1': 'Kareem', 'L2': 'Jawad', 'L3': 'Ahmed', 'L4': 'Kareem', 'L5': 'Jawad'}
```

0.0.5 Another example of A* Search

[64]:
```
import math
from simpleai.search import SearchProblem, astar

# Class containing the methods to solve the maze
class MazeSolver(SearchProblem):
    # Initialize the class
    def __init__(self, board):
        self.board = board
        self.goal = (0, 0)

        for y in range(len(self.board)):
            for x in range(len(self.board[y])):
                if self.board[y][x].lower() == "s":
                    self.initial = (x, y)
                elif self.board[y][x].lower() == "g":
                    self.goal = (x, y)

        super(MazeSolver, self).__init__(initial_state=self.initial)

    # Define the method that takes actions
    # to arrive at the solution
    def actions(self, state):
        actions = []
        for action in COSTS.keys():
            newx, newy = self.result(state, action)
            if self.board[newy][newx] != "#":
                actions.append(action)

        return actions

    # Update the state based on the action
    def result(self, state, action):
        x, y = state
        if action.count("up"):
            y -= 1
        if action.count("down"):
            y += 1
        if action.count("left"):
            x -= 1
        if action.count("right"):
            x += 1

        new_state = (x, y)

        return new_state

    # Check if we have reached the goal
    def is_goal(self, state):
        return state == self.goal

    # Compute the cost of taking an action
    def cost(self, state, action, state2):
        return COSTS[action]

    # Heuristic that we use to arrive at the solution
    def heuristic(self, state):
        x, y = state
        gx, gy = self.goal

        return math.sqrt((x - gx) ** 2 + (y - gy) ** 2)

if __name__ == "__main__":
    # Define the map
    MAP = """
    ##############################
    #         #              #   #
    # ####    ########       #   #
    #  S #    #              #   #
    #    ###     #####  ######   #
    #      #   ###   #           #
    #      #     #   #  #  #   ###
    #     #####    #    #  # G   #
    #              #       #     #
    ##############################
    """

    # Convert map to a list
    print(MAP)
    MAP = [list(x) for x in MAP.split("\n") if x]
```

```python
# Define cost of moving around the map
cost_regular = 1.0
cost_diagonal = 1.7

# Create the cost dictionary
COSTS = {
    "up": cost_regular,
    "down": cost_regular,
    "left": cost_regular,
    "right": cost_regular,
    "up left": cost_diagonal,
    "up right": cost_diagonal,
    "down left": cost_diagonal,
    "down right": cost_diagonal,
}

# Create maze solver object
problem = MazeSolver(MAP)

# Run the solver
result = astar(problem, graph_search=True)

# Extract the path
path = [x[1] for x in result.path()]

# Print the result
print()
for y in range(len(MAP)):
    for x in range(len(MAP[y])):
        if (x, y) == problem.initial:
            print('S', end='')
        elif (x, y) == problem.goal:
            print('G', end='')
        elif (x, y) in path:
            print('·', end='')
        else:
            print(MAP[y][x], end='')

    print()
```

```
##############################
#         #              #   #
# ####    ########       #   #
#  S #    #              #   #
#    ###     #####  ######   #
#      #   ###   #           #
#      #     #   #  #  #   ###
#     #####    #    #  # G   #
#              #       #     #
##############################


##############################
#         #              #   #
# ####    ########       #   #
#  S #    #              #   #
#   ·###     #####  ######   #
#   ·  #   ###   #  ····     #
#   ·  #     # ··# ·#  #·  ###
#    ·#####   ·# ·· #  # G   #
#     ········ #       #     #
##############################
```