Hacettepe University - Computer Engineering

Artificial Intelligence CMP682

Homework-2

Student Name	Ayça KULA
Student ID:	202237285

Fall 2021

1 3-SAT with Z3

Consider the propositional formula written in CNF form,

$$(x \lor z \lor t) \land (y \lor a \lor \neg b) \land (t \lor a \lor b) \land (y \lor t \lor a) \land (((z \lor a) \land (x \land b)) \lor (t \lor a)) \land (z \lor y \lor x) \tag{1}$$

The formula is satisfiable, because if we choose $a = True$ and $z = True$, for any other x,y,t, and b values the result yeilds to $True$.

$$\begin{aligned}(x \lor True \lor t) \land (y \lor True \lor \neg b) \land (t \lor True \lor b) \land (y \lor t \lor True) \\ \land(((z \lor True) \land (x \land b)) \lor (t \lor True)) \land (True \lor y \lor x)\end{aligned} \tag{2}$$

$$True \land True \land True \land True \land (((z \lor True) \land (x \land b)) \lor True) \land True \tag{3}$$

$$True \land True \land True \land True \land True \land True \tag{4}$$

Our constraints are x,z,t,y,a,b respectively. The SMT-Lib is written as:

```
(declare-const x Bool)
(declare-const z Bool)
(declare-const t Bool)
(declare-const y Bool)
(declare-const a Bool)
(declare-const b Bool)
(assert (and
   (or x z t)
   (or y a (not b))
   (or t a b)
   (or y t a)
   (or (and (or z a) (and x b)) (or t a) )
   (or z y x)
   )
)
(check-sat)
(get-model)
(exit)

```

By using [1], we get the 3-SAT solution as:

```
sat
(model
  (define-fun z () Bool
    true)
  (define-fun t () Bool
    false)
  (define-fun y () Bool
    false)
  (define-fun a () Bool
    true)
  (define-fun x () Bool
    false)
  (define-fun b () Bool
```

```
            false)
        )

```

2 Unrestricted Formulas

Given φ a SAT formula we create a 3SAT formula ψ. Consider the propositional formula obtained from [2],

$$\begin{aligned}\varphi = &(x_1 \Rightarrow \neg x_4) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \\ &\wedge ((x_2 \Rightarrow \neg x_3) \vee (\neg x_4 \Rightarrow x_1)) \wedge (x_1)\end{aligned} \tag{5}$$

We convert into CNF form as eliminating arrows, driving in negations and by distributing.

$$\begin{aligned}\varphi = &(\neg x_1 \vee \neg x_4) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \\ &\wedge (\neg x_2 \vee \neg x_3 \vee x_4 \vee x_1) \wedge (x_1)\end{aligned} \tag{6}$$

Let us show this equation in SMT-Lib format in order to show equisatisfiablity at the end.

```
        (declare-const x1 Bool)
        (declare-const x2 Bool)
        (declare-const x3 Bool)
        (declare-const x4 Bool)

        (assert (and
          (or (not x1) (not x4))
          (or x1 (not x2) (not x3))
          (or (not x2) (not x3) x4 x1)
          (or x1)
          )
        )
        (check-sat)
        (get-model)
        (exit)
```

The solution is obtained from [1] and it can be written as:

```
sat
(
  (define-fun x3 () Bool
    false)
  (define-fun x2 () Bool
    false)
  (define-fun x1 () Bool
    true)
  (define-fun x4 () Bool
    false)
)
```

Let $z, y1, u, v$ be new our variables. And finally we write 3-SAT as:

$$
\begin{aligned}
\psi = &(\neg x_1 \vee \neg x_4 \vee z) \wedge (\neg x_1 \vee \neg x_4 \vee \neg z) \\
&\wedge (x_1 \vee \neg x_2 \vee \neg x_3) \\
&\wedge (\neg x_2 \vee \neg x_3 \vee y_1) \wedge (x_4 \vee x_1 \vee \neg y_1) \\
&\wedge (x_1 \vee u \vee v) \wedge (x_1 \vee u \vee \neg v) \\
&\wedge (x_1 \vee \neg u \vee v) \wedge (x_1 \vee \neg u \vee \neg v)
\end{aligned} \tag{7}
$$

We can write the formula above in SMT-LIB2 format as:

```
(declare-const x1 Bool)
(declare-const x2 Bool)
(declare-const x3 Bool)
(declare-const x4 Bool)
(declare-const z Bool)
(declare-const y Bool)
(declare-const u Bool)
(declare-const v Bool)

(assert (and
(or (not x1) (not x4) z)
(or (not x1) (not x4) (not z))
(or x1 (not x2) (not x3))
(or (not x2) (not x3) y)
(or x4 x1 (not y))
(or x1 u v)
(or x1 u (not v))
(or x1 (not u) v)
(or x1 (not u) (not v))
)
)
(check-sat)
(get-model)
(exit)
```

The 3-SAT solution is obtained from [1] and it can be written as:

```
sat
(
  (define-fun y () Bool
    false)
  (define-fun x3 () Bool
    false)
  (define-fun z () Bool
    false)
  (define-fun x2 () Bool
    false)
  (define-fun x1 () Bool
    true)
  (define-fun x4 () Bool
    false)
  (define-fun u () Bool
    false)
  (define-fun v () Bool
    false)
)
```

Since both of the solutions obtained from [1] are satisfiable, we say that they are equisatisfiable.

3 N-Queens Problem

3.1 Specify the problem

The main idea here is that for every position (i, j) on the board a booelean variable p_{ij} shows whether there is a queen or not[3]. Let's consider a 4-queen problem and set some equations.

(a) Rows (b) Columns

Figure 1: Obtaining row and column equations.

1. **Rows:** As seen from figure 1a, p_{ij} and $p_{i'j'}$ are in the same row if $i = j$.

   - There has to be at **least** one queen on every row i:

   $$p_{i1} \vee p_{i2} \vee p_{i3} \vee p_{i4} \tag{8}$$

   And if we show this equation for all N queen problem, we can show this formula as:

   $$\bigvee_{j=1}^{n} p_{ij} \tag{9}$$

   - There has to be at **most** one queen on every row i: If we take two variable, at least one of them has to be false. It means that for ever, $j < k$ not both p_{ij} and p_{ik} are true.

   $$\text{Therefore, } \neg p_{ij} \vee \neg p_{ik} \text{ for all } j < k \tag{10}$$

   $$\bigwedge_{0<j<k\leq n} (\neg p_{ij} \vee \neg p_{ik}) \tag{11}$$

2. **Columns:** For columns, the i and j are swapped. However, the requirements are the same.

- There has to be at **least** one queen on every column:

$$\bigwedge_{j=1}^{n} \bigvee_{i=1}^{n} p_{ij} \tag{12}$$

- There has to be at **most** one queen on every column:

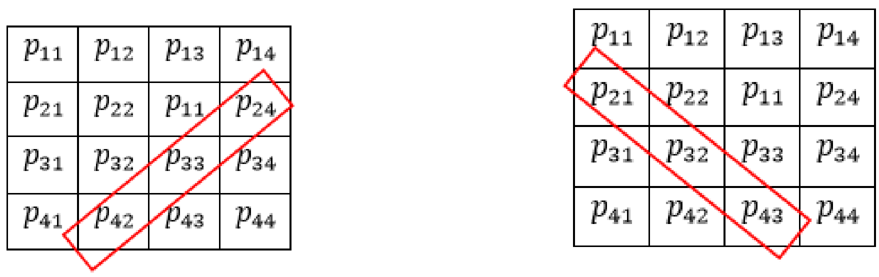
$$\bigwedge_{j=1}^{n} \bigwedge_{0<i<k\leq n} (\neg p_{ij} \vee \neg p_{kj}) \tag{13}$$

3. **Diagonal:**

   At **most** one queen should be on evry diagonal.If variables p_{ij} and $p_{i'j'}$ are on the same diagonal as in figure 2a, $i+j=i'+j'$. However, if the diagonal is reversed in other direction as in 2b, $i-j=i'-j'$. Therefore, for all i,j,i',j' with $(i,j)\neq(i'j')$ satisfying $i+j=i'+j'$ or $i-j=i'-j'$:

$$\neg p_{ij} \vee \neg p_{i'j'} \tag{14}$$

So, both of the variables should not be true at the same time.



(a) Diagonal

(b) Diagonal is reversed

Figure 2: Obtaining diagonal equations.

We define $i<i'$, and therefore the equation is given as:

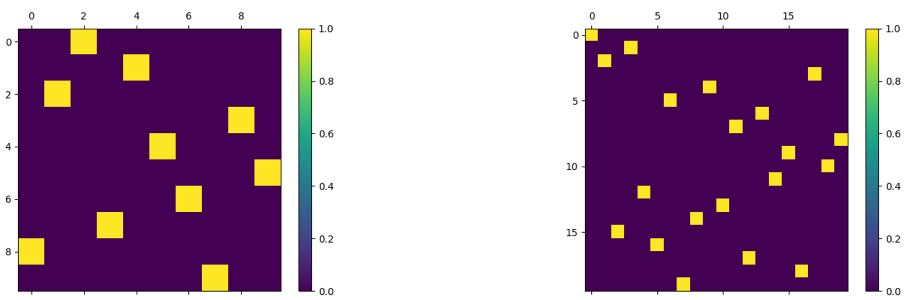
$$\bigwedge_{0<i<i'\leq n} \left(\bigwedge_{j,j':i+j=i'+j' \vee i-j=i'-j'} \neg p_{ij} \vee \neg p_{i'j'} \right) \tag{15}$$

Total formula is given as:

$$
\begin{aligned}
&\bigwedge_{i=1}^{8} \bigvee_{j=1}^{8} p_{ij} \wedge \\
&\bigwedge_{i=1}^{8} \bigwedge_{0<j<k\leq 8} (\neg p_{ij} \vee \neg p_{ik}) \wedge \\
&\bigwedge_{j=1}^{8} \bigvee_{i=1}^{8} p_{ij} \wedge \\
&\bigwedge_{j=1}^{8} \bigwedge_{0<k\leq 8} (\neg p_{ij} \vee \neg p_{kj}) \wedge \\
&\bigwedge_{0<i<i'\leq 8} \left(\bigwedge_{j,j':i+j=i'+j' \vee i-j=i'-j'} \neg p_{ij} \vee \neg p_{i'j'} \right)
\end{aligned} \tag{16}
$$

3.2 Codes/Results

The code given in 1 is obtain for the N-queens problem with Z3 based on [4]. Here, *z3_solver* has been imported. By the commented texts, you can see that the requirements considered in the first part of this question has been implemented. The results obtained from the code is given in figures 3a,3b only for the 10 and 20 queen's problem. In order to solve SAT problems with z3 such commands like Bool, s.check(), s.add(..) has been used. Moreover, these functions are used with the help of [5]. You can obtain different N-Queen problem by modfying the N parameter in the code 1.



(a) 10 Queen

(b) 20 Queen

Figure 3: N Queen Problem

```python
import numpy as np
import matplotlib.pyplot as plt
from z3 import *

N = 10 # N queens problem

# Each queen must be in a different row.
# Each queen is represented by a single integer: the column position
Q = [Int(f"Q_{row + 1}") for row in range(N)]

# Each queen is in a column {1, ... N }
val_c = [And(1 <= Q[row], Q[row] <= N) for row in range(N)]

# At most one queen per column
col_c = [Distinct(Q)]

# Diagonal constraint
diag_c = [If(i == j, True,
             And(Q[i] - Q[j] != i - j, Q[i] - Q[j] != j - i))
          for i in range(N) for j in range(i)]

solve(val_c + col_c + diag_c)
s = Solver()
s.add(val_c + col_c + diag_c)
s.check()
m = s.model()
board = np.zeros((N,N))

# Visualize the board
for i in range(N):
    board[i, m[Q[i]].as_long() - 1] = 1

# Plot figure
figure = plt.figure()
axes = figure.add_subplot(111)
caxes = axes.matshow(board)
figure.colorbar(caxes)
plt.show()
```

Algorithm 1: N-Queens problem with Z3

Moreover, you can see the SMT-LIB format for the 4-Queen's problem.

```
;; The definitions of the variables
(declare-const x0y0 Bool)
(declare-const x0y1 Bool)
(declare-const x0y2 Bool)
(declare-const x0y3 Bool)
```

```
(declare-const x1y0 Bool)
(declare-const x1y1 Bool)
(declare-const x1y2 Bool)
(declare-const x1y3 Bool)
(declare-const x2y0 Bool)
(declare-const x2y1 Bool)
(declare-const x2y2 Bool)
(declare-const x2y3 Bool)
(declare-const x3y0 Bool)
(declare-const x3y1 Bool)
(declare-const x3y2 Bool)
(declare-const x3y3 Bool)
;;"one queen by line" clauses

(assert (or x0y0  x0y1  x0y2 x0y3))
(assert (or x1y0  x1y1  x1y2 x1y3))
(assert (or x2y0  x2y1  x2y2 x2y3))
(assert (or x3y0  x3y1  x3y2 x3y3))

;;"only one queen by line" clauses

(assert (not (or(and x0y1 x0y0)(and x0y2 x0y0)(and x0y2 x0y1)(and x0y3 x0y0)(and x0y3 x0y1)(and x0y3 x0y2))))
(assert (not (or(and x1y1 x1y0)(and x1y2 x1y0)(and x1y2 x1y1)(and x1y3 x1y0)(and x1y3 x1y1)(and x1y3 x1y2))))
(assert (not (or(and x2y1 x2y0)(and x2y2 x2y0)(and x2y2 x2y1)(and x2y3 x2y0)(and x2y3 x2y1)(and x2y3 x2y2))))
(assert (not (or(and x3y1 x3y0)(and x3y2 x3y0)(and x3y2 x3y1)(and x3y3 x3y0)(and x3y3 x3y1)(and x3y3 x3y2))))

;;"only one queen by column" clauses
(assert (not (or(and x1y0 x0y0)(and x2y0 x0y0)(and x2y0 x1y0)(and x3y0 x0y0)(and x3y0 x1y0)(and x3y0 x2y0))))
(assert (not (or(and x1y1 x0y1)(and x2y1 x0y1)(and x2y1 x1y1)(and x3y1 x0y1)(and x3y1 x1y1)(and x3y1 x2y1))))
(assert (not (or(and x1y2 x0y2)(and x2y2 x0y2)(and x2y2 x1y2)(and x3y2 x0y2)(and x3y2 x1y2)(and x3y2 x2y2))))
(assert (not (or(and x1y3 x0y3)(and x2y3 x0y3)(and x2y3 x1y3)(and x3y3 x0y3)(and x3y3 x1y3)(and x3y3 x2y3))))

;;"only one queen by diagonal" clauses
(assert (not (or (and x0y0 x1y1) (and x0y0 x2y2) (and x0y0 x3y3) (and x1y1 x2y2) (and x1y1 x3y3) (and x2y2 x3y3))))
(assert (not (or (and x0y1 x1y2) (and x0y1 x2y3) (and x1y2 x2y3))))
(assert (not (or (and x0y2 x1y3))))
(assert (not (or (and x1y0 x2y1) (and x1y0 x3y2) (and x2y1 x3y2))))
(assert (not (or (and x2y0 x3y1))))
(assert (not (or (and x3y0 x2y1) (and x3y0 x1y2) (and x3y0 x0y3) (and x2y1 x1y2) (and x2y1 x0y3) (and x1y2 x0y3))))
(assert (not (or (and x2y0 x1y1) (and x2y0 x0y2) (and x1y1 x0y2))))
(assert (not (or (and x1y0 x0y1))))
(assert (not (or (and x3y1 x2y2) (and x2y1 x1y3) (and x2y2 x1y3))))
```

```
    (assert (not (or (and x3y2 x1y3))))

    ;; Check if the generate model is satisfiable and output a
   model.
    (check-sat)
    (get-model)
```

And the solution can be given as for this problem:

```
sat
(
  (define-fun x3y1 () Bool
    true)
  (define-fun x0y0 () Bool
    false)
  (define-fun x3y2 () Bool
    false)
  (define-fun x1y0 () Bool
    true)
  (define-fun x0y3 () Bool
    false)
  (define-fun x0y1 () Bool
    false)
  (define-fun x2y3 () Bool
    true)
  (define-fun x2y0 () Bool
    false)
  (define-fun x1y2 () Bool
    false)
  (define-fun x3y0 () Bool
    false)
  (define-fun x3y3 () Bool
    false)
  (define-fun x0y2 () Bool
    true)
  (define-fun x1y3 () Bool
    false)
  (define-fun x2y1 () Bool
    false)
  (define-fun x2y2 () Bool
    false)
  (define-fun x1y1 () Bool
    false)
)
```

4 Layout Problem

We have to first specify the rectangle fitting problem[6]. The number of rectangles vary from 1 to n which is shown with i. Then, we have to introduce some variables for each rectangle:

- w_i: width of rectangle i
- h_i: height of rectangle i

- x_i: x-coordinate of the left lower corner of rectangle i
- y_i: y-coordinate of the left lower corner of rectangle i

Now, we have to write all the requirements in order to obtain our code.

4.1 Requirements

1. **Width/height requirements**: Assume we have a ith rectangle with width w and height of h, then we can obtain a formula for all rectangles as:

$$(w_i = w \wedge h_i = h) \vee (w_i = h \wedge h_i = w) \tag{17}$$

for all $i = 1, ..., n$

This formula indicates that for all the rectangles, the 90 degrees orientation of rectangle has been also considered which is defined after "or" operator.

2. **All rectangles has to fit the big rectangle:** If we consider the lower left corner as the origin $(0, 0)$, and the width and height of the big rectangle respectively is W, H. For each ith rectangle x_i should be greater or equal to zero. And not too far to the right that is $x_i + w_i$ should be less or equal than W. This requirement is written as in equation 18. And the same situation is written for height requirement and it is given in equation 19.

$$x_i \geq 0 \wedge x_i + w_i \leq W \tag{18}$$

and

$$y_i \geq 0 \wedge y_i + h_i \leq H \tag{19}$$

for all $i = 1, ..., n$

3. **No overlap requirement:** First see the figure given in 4 and specify an equation for overlapping.

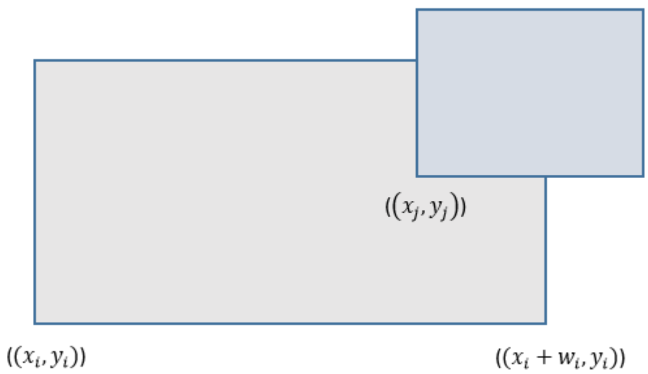


Figure 4: Overlapping rectangles

Therefore, if the conditions given below are satisfied overlapping exist[6].

A Right side of rectangle i is right from left side of rectangle j.

$$x_i + w_i > x_j \tag{20}$$

B Left side of rectangle i is left from right side of rectangle j.

$$x_i < x_j + w_j \tag{21}$$

C Top of rectangle i is above from bottom of rectangle j.

$$y_i + h_i > y_j \tag{22}$$

D Bottom of rectangle i is below top of rectangle j.

$$y_i < y_j + h_j \tag{23}$$

Since, we want no overlapping we should take the negation of the conditions defined above for all $i, j = 1, ..., n, i < j$,

$$\neg (x_i + w_i > x_j \wedge x_i < x_j + w_j \wedge y_i + h_i > y_j \wedge y_i < y_j + h_j) \tag{24}$$

or,equivalently, by removing the negation,

$$x_i + w_i \leq x_j \vee x_j + w_j \leq x_i \vee y_i + h_i \leq y_j \vee y_j + h_j \leq y_i \tag{25}$$

The summary of all the requirements can be written as:

$$\begin{aligned} &\bigwedge_{i=1}^{n} ((w_i = W_i \wedge h_i = H_i) \vee (w_i = H_i \wedge h_i = W_i)) \\ &\wedge \bigwedge_{i=1}^{n} (x_i \geq 0 \wedge x_i + w_i \leq W \wedge y_i \geq 0 \wedge y_i + h_i \leq H) \\ \wedge &\bigwedge_{1 \leq i < j \leq n} (x_i + w_i \leq x_j \vee x_j + w_j \leq x_i \vee y_i + h_i \leq y_j \vee y_j + h_j \leq y_i) \end{aligned} \tag{26}$$

4.2 Codes/Solution

After analyzing the problem,the code is written with respect to the obtained requirements. We know that we have to formulate as an CSP problem. Therefore, respectively, variables,domains,constraints and requirements were defined as in [7]. The problem is solved using backtracking search. The number of visited nodes and the xplored nodes is taken into consideration while obtaining the algoirthm. Rectangles are defined as array of size nxm.

Each block is indicated with a letter, for example 2x2 block is indicated with a letter. By using the code given below, 4 rectangular blocks of size 2x2 into a 8x8 grid are placed as:

```
Number of nodes visited : 4
Number of nodes explored : 14
Number of Inconsistencies Encountered : 0

_ _ _ _ a a _ _
_ _ _ _ a a _ _
_ _ _ y y z z _
_ _ _ y y z z _
_ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _
_ _ _ _ x x _ _
_ _ _ _ x x _ _
```

Then, number of blocks, the sizes of the blocks, and the size of the grid are varied as:

```
Number of nodes visited : 6
Number of nodes explored : 13
Number of Inconsistencies Encountered : 0

y y y y y y _ _ _ _ _ d d d d
y y y y y y _ _ _ _ _ d d d d
y y y y y y _ _ _ _ _ d d d d
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ b b b b b b b b b b b b
_ _ _ b b b b b b b b b b b b
x x _ _ _ _ _ _ _ _ _ _ _ _ _
x x _ a a a a a a a a _ _ _ _
x x _ a a a a a a a a z z z z
x x _ a a a a a a a a z z z z
```

And the code is given below:

```python

import matplotlib.pyplot as plt

# Formulate as an CSP problem
class ConstraintSatisfactionProblem:

    # Variables, domains, constraints has to be defined in an CSP.
    def __init__(self, variables, domains, constraints, mrv, lcv,
   mac):
        self.variables = variables
        self.domains = domains
        self.constraints = constraints
        self.removed = {}
        self.mrv = mrv
        self.lcv = lcv    # mrv: Minimum-Remaining_Value
        self.mac = mac    # Maintaining Arc Consistency
        self.nodes_visited = 0
        self.nodes_explored = 0
        self.num_of_inconsistency = 0

    # Backtrack caller function
    def backtrack_search(self):
        sol = self.backtrack({})
        return sol

```

```python
    # Backtrack helper function: assigns a value to a variable at every
    # recursive call made to the function.If the assignment turns out
    # to be a failure (no legal value can be assigned to a variable),the
    # function returns None, recurs to its parent node, and tries out a
    # different value assignment.
    def backtrack(self, assignment):  #
        if self.is_complete(assignment):
            return assignment

        self.nodes_visited += 1

        var = self.select_unassigned_var(assignment)
        self.order_domain(assignment, var)

        for val in self.order_domain(assignment, var):
            self.nodes_explored += 1

            if self.is_consistent(assignment, var, val):
                domains_reserve = {}
                assignment[var] = val
                self.set_domain(domains_reserve, var, val)

                if self.test_inference(assignment, var, domains_reserve):
                    result = self.backtrack(assignment)

                    if result is not None:
                        return result

                del assignment[var]

                self.restore_domain(domains_reserve)

        self.num_of_inconsistency += 1
        return None

    # When a value is assigned to the variable, remove every other
    # value from its domain and add it to the domain reserve so
    # that the domain can be restored when desired.
    def set_domain(self, domains_reserve, var, val):
        domains_reserve[var] = set()
        domains_reserve[var] = domains_reserve[var].union(self.domains[var])
        domains_reserve[var].remove(val)

        self.domains[var].clear()
        self.domains[var].add(val)

    # Restore the variables' domain as noted above.
    def restore_domain(self, domains_reserve):
        for key in domains_reserve:
            self.domains[key] = self.domains[key].union(
```

```
domains_reserve[key])

    # The assignment is complete when the dictionary has an assigned value
    # for every variable.
    def is_complete(self, assignment):
        return len(assignment.keys()) == len(self.variables)

    # If MCV flag is on, search through the variable list, and
    # return the most constrained variable, and if MCV flag is
    # off, just return the first variable you come across that
    # has not been assigned a value yet.
    def select_unassigned_var(self, assignment):
        min_count = float("inf")
        mcv_index = None

        for var in range(len(self.variables)):

            if var in assignment.keys():
                continue

            if not self.mrv:
                return var

            count = 0
            for key in assignment.keys():
                for constraint in self.constraints[(var, key)]:
                    if constraint[1] == assignment[key]:
                        count += len(self.constraints[(var, key)])

            mcv_index = var if count < min_count else mcv_index
            min_count = min(min_count, count)

        return mcv_index

    # If LCV flag is on, sort your domain according to the number of
    # legal moves it leaves for its neighboring variable.
    # If not, return a listified default domain
    def order_domain(self, assignment, var):
        if self.lcv:
            ret = []
            to_be_sorted = []

            for val in self.domains[var]:
                count = 0
                for variable in range(len(self.variables)):
                    if variable not in assignment.keys():
                        for x in self.constraints[(var, variable)]:
                            if val == x[0]:
                                count += 1
                        # count += len(self.constraints[(var, variable)])
                to_be_sorted.append((val, count))

            to_be_sorted.sort(key=lambda tup: tup[1], reverse=True)
```

```
            # Reinitialize the domain
            for entry in to_be_sorted:
                ret.append(entry[0])

            return ret
        return list(self.domains[var])

    # Loop through the constraints dictionary, and see if value assignment is legal or not.
    def is_consistent(self, assignment, var, val):
        for key in assignment.keys():
            if key != var and (val, assignment[key]) not in self.constraints[(var, key)]:
                return False
        return True

    # If MAC flag is on, loop through the queue, and determine whether the
    # domain has been modified to enforce arc
    # consistency.
    def test_inference(self, assignment, var, domains_reserve):
        if self.mac:
            queue = self.build_arc_queue(assignment, var)

            while len(queue) > 0:
                x = queue.pop()

                if self.revise(assignment, x[0], x[1], domains_reserve):
                    if len(self.domains[x[0]]) == 0:
                        return False

                    for var_2 in range(len(self.variables)):
                        if len(self.constraints[(var, var_2)]) > 0 and var_2 not in x:
                            queue.add((var_2, x[0]))

        return True

    def build_arc_queue(self, assignment, var):
        queue = set()

        for var_2 in range(len(self.variables)):
            if var_2 == var or not (len(self.constraints[(var, var_2)]) > 0 and var_2 not in assignment):
                continue
            queue.add((var, var_2))

        return queue

    def revise(self, assignment, var_1, var_2, domains_reserve):
        revised = False
        to_be_removed = []

        for d_1 in self.domains[var_1]:
```

```
            constrained = False
            for d_2 in self.domains[var_2]:
                if (d_1, d_2) in self.constraints[(var_1, var_2)]:
                    constrained = True
            if not constrained:
                to_be_removed.append(d_1)
                revised = True

        for d_1 in to_be_removed:
            self.domains[var_1].remove(d_1)

            if var_1 not in domains_reserve.keys():
                domains_reserve[var_1] = []

            domains_reserve[var_1].add(d_1)

        return revised


class RectangleFitProblem(ConstraintSatisfactionProblem):
    # No mrv, lcv or mac is used for this problem.
    def __init__(self, variables, board, mrv=False, lcv=False, mac=False):
        self.variables = variables
        self.domains = self.build_domains(board)
        self.constraints = self.build_constraints(board)
        self.mrv = mrv
        self.lcv = lcv
        self.mac = mac
        self.nodes_visited = 0
        self.BOARD = board
        self.nodes_explored = 0
        self.num_of_inconsistency = 0

    # The domain for each component would be a set of possible coordinates
    # of the component's bottom-left corner.
    def build_domains(self, board):
        domains = {}
        for v in range(len(self.variables)):
            domains[v] = set()

            x = len(board) - len(self.variables[v])
            y = len(board[0]) - len(self.variables[v][0])

            for i in range(x + 1):
                for j in range(y + 1):
                    domains[v].add((i, j))

        return domains

    # The constraint for each pair of component pieces is a set of
    # possible coordinates of the two pieces
    # within each's domain where the two pieces do not overlap.
    def build_constraints(self, board):
        constraints = {}
```

```python

        for i in range(len(self.variables)):
            for j in range(len(self.variables)):
                constraints[(i, j)] = set()
                v_1 = self.variables[i]
                v_2 = self.variables[j]

                if v_1 == v_2: continue

                for d_1 in self.domains[i]:
                    for d_2 in self.domains[j]:
                        upperbound_x = max(d_1[0] + len(v_1), d_2[0] + len(v_2))
                        upperbound_y = max(d_1[1] + len(v_1[0]), d_2[1] + len(v_2[0]))

                        lowerbound_x = min(d_1[0], d_2[0])
                        lowerbound_y = min(d_1[1], d_2[1])

                        if (upperbound_x - lowerbound_x >= len(v_1) + len(v_2)) or (upperbound_y - lowerbound_y >= len(v_1[0]) + len(v_2[0])):

                            if upperbound_x <= len(board) and upperbound_y <= len(board[0]):
                                constraints[(i, j)].add((d_1, d_2))

        return constraints

    def solve(self):
        # Solve the problem using backtracking search.
        solution = self.backtrack_search()
        return solution

    def __str__(self):
        solution = self.solve()

        if solution is None:
            return "No solution exists"

        board = [['_' for i in range(len(self.BOARD[0]))] for j in range((len(self.BOARD)))]

        for key in solution.keys():
            for x in range(len(self.variables[key])):
                for y in range(len(self.variables[key][0])):
                    x_index = x + solution[key][0]
                    y_index = y + solution[key][1]

                    board[x_index][y_index] = self.variables[key][x][y]

        res = "MRV=" + str(self.mrv) + " LCV=" + str(self.lcv) + " MAC-3=" + str(self.mac) + "\n"
        res += "Number of nodes visited : " + str(self.nodes_visited) + "\n"
```

```
        res += "Number of nodes explored : " + str(self.nodes_explored) + "\n"
        res += "Number of Inconsistencies Encountered : " + str(self.num_of_inconsistency) + "\n"
        for j in range(len(board[0]) - 1, -1, -1):
            for i in range(len(board)):
                res += board[i][j]
            res += '\n'

        return res


if __name__ == '__main__':
    matrices = [[['x', 'x'] for i in range(2)], [['y', 'y'] for i in range(2)], [['z', 'z'] for i in range(2)],
                [['a', 'a'] for i in range(2)]]
    board = [['_' for i in range(8)] for j in range(8)]
    test = RectangleFitProblem(matrices, board)
    print(test)

    matrices_2 = [[['x', 'x', 'x', 'x'] for i in range(2)], [['y', 'y','y'] for i in range(6)],
                  [['z', 'z'] for i in range(4)], [['d', 'd', 'd'] for i in range(4)],
                  [['a', 'a', 'a'] for i in range(8)], [['b','b'] for i in range(12)]]
    board_2 = [['_' for i in range(10)] for j in range(15)]
    test_2 = RectangleFitProblem(matrices_2, board_2)
    print(test_2)
```

References

[1] Getting started with z3: A guide. URL https://jfmc.github.io/z3-play/.

[2] CS 573. Reductions and np. https://courses.grainger.illinois.edu/cs573/fa2013/lec/slides/02_notes.pdf.

[3] Eight queens problem - sat/smt basics, sat examples, . URL https://www.coursera.org/learn/automated-reasoning-sat/lecture/KZzKe/eight-queens-problem.

[4] philzook. z3 tutorial. https://github.com/philzook58/z3_tutorial/blob/master/Z3Tutorial.ipynb.

[5] Victor Nicolet. Csc410 tutorial: solving sat problems with z3. URL http://www.cs.toronto.edu/~victorn/tutorials/z3_SAT_2019/index.html.

[6] Rectangle fitting - smt applications, . URL https://www.coursera.org/learn/automated-reasoning-sat/lecture/vW7l9/rectangle-fitting.

[7] songjon93. Constraint-satisaction-problem/csp at master · songjon93/constraint-satisaction-problem. URL https://github.com/songjon93/Constraint-Satisaction-Problem/tree/master/CSP.