Contents

[Tiramisu Problem 2](#_Toc88252141)

[Variables 2](#_Toc88252142)

[Solution Printer 4](#_Toc88252143)

[Constraint 1 5](#_Toc88252144)

[Constraint 2 6](#_Toc88252145)

[Constraint 3 6](#_Toc88252146)

[Constraint 4 7](#_Toc88252147)

[Constraint 5 7](#_Toc88252148)

[Constraint 6 7](#_Toc88252149)

[Constraint 7 8](#_Toc88252150)

[Constraint 8 8](#_Toc88252151)

[Constraint 9 8](#_Toc88252152)

[Solution that is printed 10](#_Toc88252153)

[Sudoku Solver 10](#_Toc88252154)

[Helper Routines 10](#_Toc88252155)

[Routine to return the number 1..9 10](#_Toc88252156)

[Routine to return the indices of all cells in a row 11](#_Toc88252157)

[Routine to return indices of all cells in a column 11](#_Toc88252158)

[Routine to return indices of all cells in a 3x3 sub-square 11](#_Toc88252159)

[Routine to return the indices of the top-left square of each 3x3 sub-square 11](#_Toc88252160)

# Tiramisu Problem

The Tiramisu problem was setup as follows

## Variables

Four sets of variables were created:

1. Person-Starter
2. Person-Maincourse
3. Person-Drink
4. Person-Dessert

To create the variables, a common routine was written to make repetitive tasks easier.

**def** **create\_variables\_and\_implicit\_constraints**(

model,

var\_list1: list,

var\_list2: list) -> dict:

"""Create a 2D variable array given the two axes

For example, given Person and Drink, create a 2D array

for each person and drink

Also create the implicit constraints that

1. each person must have a drink

2. each person can have exactly one drink

3. No two persons have the same drink

Args:

model ([type]): the CP SAT model

var\_list1 (list): list of items in first axes (eg. person names)

var\_list2 (list): list of items in second axis (eg. drink names)

Returns:

dict: [description]

"""

# Create the variables

ret\_dict = {}

**for** var1 **in** var\_list1:

ret\_dict[var1] = \

{var2: model.NewBoolVar(f"{var1}--{var2}") **for** var2 **in** var\_list2}

# Every item in var\_list1 has a different property from var\_list2

**for** i **in** range(len(var\_list1)):

**for** j **in** range(i+1, len(var\_list1)):

**for** k **in** range(len(var\_list2)):

model.AddBoolOr( \

[ \

ret\_dict[var\_list1[i]][var\_list2[k]].Not(), \

ret\_dict[var\_list1[j]][var\_list2[k]].Not() \

] \

)

# At least one item in var\_list2 for each item in var\_list1

**for** v1 **in** var\_list1:

model.AddBoolOr([ret\_dict[v1][v2] **for** v2 **in** var\_list2])

# Max one property for every item in var\_list1

**for** v1 **in** var\_list1:

**for** i **in** range(len(var\_list2)):

**for** j **in** range(i+1, len(var\_list2)):

model.AddBoolOr( \

[ \

ret\_dict[v1][var\_list2[i]].Not(), \

ret\_dict[v1][var\_list2[j]].Not() \

] \

)

**return** ret\_dict

The above function creates a 2D array, and also creates implicit constraints between the two axes.

Finally this function is called as follows:

model = cp\_model.CpModel()

person\_starter = create\_variables\_and\_implicit\_constraints( \

model, \

PERSON, \

STARTER)

person\_maincourse = create\_variables\_and\_implicit\_constraints( \

model, \

PERSON, \

MAINCOURSE)

person\_drink = create\_variables\_and\_implicit\_constraints( \

model, \

PERSON, \

DRINK)

person\_dessert = create\_variables\_and\_implicit\_constraints( \

model, \

PERSON, \

DESSERT)

These arrays are defined as follows:

PERSON = ["James", "Daniel", "Emily", "Sophie"]

STARTER = ["Prawn\_Cocktail", "Onion\_Soup", "Mushroom\_Tart", "Carpaccio"]

MAINCOURSE = ["Baked\_Mackerel", "Fried\_Chicken", "Filet\_Steak", "Vegan\_Pie"]

DRINK = ["Red\_Wine", "Beer", "White\_Wine", "Coke"]

DESSERT = ["Apple\_Crumble", "Ice\_Cream", "Chocolate\_Cake", "Tiramisu"]

## Solution Printer

The first task in the Tiramisu problem was the solution printer.

The solution printer does two things:

1. Prints the solution
2. Validates that the solution actually doesn’t contain anything that is not allowed. This is more of a double check for debugging purposes.

**class** **TiramisuSolutionPrinter**(cp\_model.CpSolverSolutionCallback):

**def** **\_\_init\_\_**(\

self,

person:list,

starter:list,

maincourse:list,

drink:list,

dessert:list,

person\_starter:dict,

person\_maincourse:dict,

person\_drink:dict,

person\_dessert:dict):

super().\_\_init\_\_()

self.person = person

self.starter = starter

self.maincourse = maincourse

self.drink = drink

self.dessert = dessert

self.person\_starter = person\_starter

self.person\_maincourse = person\_maincourse

self.person\_drink = person\_drink

self.person\_dessert = person\_dessert

self.solutions = 0

**def** **validate\_matrix**(self, matrix:dict, axis1:list, axis2:list):

"""[summary]

Args:

matrix (dict): [description]

axis1 (list): [description]

axis2 (list): [description]

"""

**for** v1 **in** axis1:

i = 0

**for** v2 **in** axis2:

**if** self.Value(matrix[v1][v2]): i = i + 1

**assert**(i == 1)

**for** v2 **in** axis2:

i = 0

**for** v1 **in** axis1:

**if** self.Value(matrix[v1][v2]): i = i + 1

**assert**(i == 1)

**def** **OnSolutionCallback**(self):

self.solutions = self.solutions + 1

print(f"Solution #{self.solutions:06d}")

print("----------------")

self.validate\_matrix(self.person\_dessert, self.person, self.dessert)

self.validate\_matrix(self.person\_drink, self.person, self.drink)

self.validate\_matrix(self.person\_maincourse, self.person, self.maincourse)

self.validate\_matrix(self.person\_starter, self.person, self.starter)

**for** person **in** self.person:

print(f"- {person}")

[print(f" - {dessert}") **for** dessert **in** self.dessert\

**if** self.Value(self.person\_dessert[person][dessert])]

[print(f" - {drink}") **for** drink **in** self.drink\

**if** self.Value(self.person\_drink[person][drink])]

[print(f" - {starter}") **for** starter **in** self.starter\

**if** self.Value(self.person\_starter[person][starter])]

[print(f" - {maincourse}") **for** maincourse **in** self.maincourse\

**if** self.Value(self.person\_maincourse[person][maincourse])]

**for** person **in** self.person:

**if** self.Value(self.person\_dessert[person]['Tiramisu']):

print(f"\n\n{person} has the Tiramisu")

**break**

print()

print()

## Constraint 1

# Explicit Constraint 1

# ---------------------

# Emily does not like prawn cocktail as starter,

# nor does she want baked mackerel as main course

model.AddBoolAnd([person\_starter["Emily"]["Prawn\_Cocktail"].Not()])

model.AddBoolAnd([person\_maincourse["Emily"]["Baked\_Mackerel"].Not()])

## Constraint 2

# Explicit Constraint 2

# ---------------------

# Daniel does not want the onion soup as starter and

# James does not drink beer

model.AddBoolAnd([person\_starter["Daniel"]["Prawn\_Cocktail"].Not()])

model.AddBoolAnd([person\_drink["James"]["Beer"].Not()])

## Constraint 3

In Constraint 3, I was not sure what exactly was meant. I could find three ways of interpreting it, so I implemented all three ways. I set the default as interpretation 1.

# ---------------------

# Explicit Constraint 3

# ---------------------

# Sophie will only have fried chicken as main course

# if she does not have to take the prawn cocktail as starter

#

# Interpretation 1:

# Or in other words Fried Chicken implies No Prawn Cocktail, and vice versa

#

# Interpretation 2:

# Another way to interpret this condition is to say that Sophie has

# either Prawn Cocktail or Fried Chicken, a xor condition.

#

# Interpretation 3:

# A third way to interpret this condition is to say that

# if she does not have prawn cocktail, she will definitely have fried

# chicken

# Or in other words, Not Prawn Cocktail implies Fried Chicken

#

#

**if** CONSTRAINT3\_INTERPRETATION\_1:

model.AddBoolOr( \

[ \

person\_starter["Sophie"]["Prawn\_Cocktail"].Not(), \

person\_maincourse["Sophie"]["Fried\_Chicken"].Not() \

] \

)

**elif** CONSTRAINT3\_INTERPRETATION\_2:

model.AddBoolXOr( \

[ \

person\_starter["Sophie"]["Prawn\_Cocktail"], \

person\_maincourse["Sophie"]["Fried\_Chicken"] \

] \

)

**elif** CONSTRAINT3\_INTERPRETATION\_3:

model.AddBoolAnd( \

[ \

person\_maincourse["Sophie"]["Fried\_Chicken"] \

] \

).OnlyEnforceIf(person\_starter["Sophie"]["Prawn\_Cocktail"].Not())

**else**:

**raise** Exception('At least one interpretation of constraint 3 must hold')

## Constraint 4

# Explicit constraint 4

# ---------------------

# The filet steak main course should be combined with the

# onion soup as starter and with the apple crumble for dessert

**for** person **in** PERSON:

model.AddBoolOr( \

[ \

person\_maincourse[person]["Filet\_Steak"].Not(), \

person\_starter[person]["Onion\_Soup"] \

])

model.AddBoolOr( \

[ \

person\_starter[person]["Onion\_Soup"].Not(), \

person\_maincourse[person]["Filet\_Steak"] \

])

model.AddBoolOr( \

[ \

person\_maincourse[person]["Filet\_Steak"].Not(), \

person\_dessert[person]["Apple\_Crumble"] \

])

model.AddBoolOr( \

[ \

person\_dessert[person]["Apple\_Crumble"].Not(), \

person\_maincourse[person]["Filet\_Steak"] \

])

## Constraint 5

# Explicit Constraint 5

# ---------------------

# The person who orders the mushroom tart as starter

# also orders the red wine

**for** person **in** PERSON:

model.AddBoolOr( \

[ \

person\_starter[person]["Mushroom\_Tart"].Not(), \

person\_drink[person]["Red\_Wine"] \

])

model.AddBoolOr( \

[ \

person\_starter[person]["Mushroom\_Tart"], \

person\_drink[person]["Red\_Wine"].Not() \

])

# ---------------------

## Constraint 6

# Explicit Constraint 6

# ---------------------

# The baked mackerel should not be combined with ice cream for dessert,

# nor should the vegan pie be ordered as main together with

# prawn cocktail or carpaccio as starter

**for** person **in** PERSON:

model.AddBoolOr( \

[ \

person\_maincourse[person]["Baked\_Mackerel"].Not(), \

person\_dessert[person]["Ice\_Cream"].Not() \

])

model.AddBoolOr( \

[ \

person\_maincourse[person]["Vegan\_Pie"].Not(), \

person\_starter[person]["Prawn\_Cocktail"].Not() \

])

model.AddBoolOr( \

[ \

person\_maincourse[person]["Vegan\_Pie"].Not(), \

person\_starter[person]["Carpaccio"].Not() \

])

## Constraint 7

# Explicit Constraint 7

# ---------------------

# The filet steak should be eaten with either beer or coke for drinks

**for** person **in** PERSON:

model.AddBoolOr( \

[ \

person\_maincourse[person]["Filet\_Steak"].Not(), \

person\_drink[person]["Beer"], \

person\_drink[person]["Coke"] \

])

## Constraint 8

# Explicit Constraint 8

# ---------------------

# One of the women drinks white wine, while the other

# prefers red wine for drinks

model.AddBoolOr( \

[ \

person\_drink["Emily"]["White\_Wine"], \

person\_drink["Emily"]["Red\_Wine"] \

])

model.AddBoolOr( \

[ \

person\_drink["Sophie"]["White\_Wine"], \

person\_drink["Sophie"]["Red\_Wine"] \

])

## Constraint 9

For constraint 9, I could think of three ways to interpret it, and I was not sure which is the correct way to understand the problem.

Three ways to understand it

1. One man has chocolate Cake, and the other can have either Coke or Ice cream or none of them. The man who has the chocolate cake can also have Coke.
2. Same as the previous interpretation except that the man who has the chocolate cake cannot have Coke and cannot have ice cream.
3. The Not is misplaced

Interpretation 1 gives multiple solutions, while 2 and 3 give only one solution. I’ve made interpretation 2 as default.

# Explicit Constraint 9

# ---------------------

# One of the men has chocolate cake for dessert while the other

# prefers not to have ice cream or coke but

# will accept one of the two if necessary

model.AddBoolXOr( \

[ \

person\_dessert["James"]["Chocolate\_Cake"], \

person\_dessert["Daniel"]["Chocolate\_Cake"] \

])

model.AddBoolOr( \

[ \

person\_dessert["James"]["Ice\_Cream"].Not(), \

person\_drink["James"]["Coke"].Not() \

]).OnlyEnforceIf(person\_dessert["Daniel"]["Chocolate\_Cake"])

model.AddBoolOr( \

[ \

person\_dessert["Daniel"]["Ice\_Cream"].Not(), \

person\_drink["Daniel"]["Coke"].Not() \

]).OnlyEnforceIf(person\_dessert["James"]["Chocolate\_Cake"])

# The problem statement doesn't say so, but probably the two conditions

# below are implicit. If the two conditions below are added,

# then we get only 1 solution.

#

# If they are discarded, we get multiple

# solutions, which satisfy all other criteria, except that the same man

# has both chocolate cake and coke.

#

# The man who has the chocolate cake doesn't have ice cream or coke

# Since there is already one condition that someone cannot have two

# desserts, we only need to cover for coke

**if** CONSTRAINT9\_INTERPRETATION\_2:

model.AddBoolAnd( \

[ \

person\_drink["James"]["Coke"].Not() \

]).OnlyEnforceIf(person\_dessert["James"]["Chocolate\_Cake"])

model.AddBoolAnd( \

[ \

person\_drink["Daniel"]["Coke"].Not() \

]).OnlyEnforceIf(person\_dessert["Daniel"]["Chocolate\_Cake"])

# Another way to arrive at a single solution (which incidentally is the

# same, is to assume that the 'Not' is misplaced, and assume that

# one man has chocolate Cake, and the other prefers to have Ice Cream

# Or Coke but cannot have both

# Since we've already added conditions that the men cannot have

# Ice cream and Coke both, we only need to add a condition that they

# have either of them when the other man has chocolate cake.

# Again, I'm not sure which of the three assumptions is correct

**elif** CONSTRAINT9\_INTERPRETATION\_3:

model.AddBoolOr( \

[ \

person\_dessert["James"]["Ice\_Cream"], \

person\_drink["James"]["Coke"] \

]).OnlyEnforceIf(person\_dessert["Daniel"]["Chocolate\_Cake"])

model.AddBoolOr( \

[ \

person\_dessert["Daniel"]["Ice\_Cream"], \

person\_drink["Daniel"]["Coke"] \

]).OnlyEnforceIf(person\_dessert["James"]["Chocolate\_Cake"])

solver = cp\_model.CpSolver()

status = solver.SearchForAllSolutions(model, solution\_printer)

print(solver.StatusName(status))

## Solution that is printed

- James

- Apple\_Crumble

- Coke

- Onion\_Soup

- Filet\_Steak

- Daniel

- Chocolate\_Cake

- Beer

- Carpaccio

- Fried\_Chicken

- Emily

- Ice\_Cream

- Red\_Wine

- Mushroom\_Tart

- Vegan\_Pie

- Sophie

- Tiramisu

- White\_Wine

- Prawn\_Cocktail

- Baked\_Mackerel

Sophie has the Tiramisu

# Sudoku Solver

## Helper Routines

For the sudoku solver, we first have a few helper routines to make the code more readable and reusable. These routines are self explanatory.

### Routine to return the number 1..9

**def** **numbers**() -> range:

"""This routine just makes it easier to loop

through the numbers 1..9

Returns:

range: [description]

"""

**return** range(1,10)

### Routine to return the indices of all cells in a row

**def** **row**(r:int) -> list:

"""Returns the list of tuples that specify the

indices for all squares of a row

This routine just makes it easier to iterate a row

Args:

r (int): tow number whose indices are to be generated

Returns:

list: list of tuples specifying the row indices, eg.

[(3, 0), (3, 1), (3, 2), ... ]

"""

**return** [(r, i) **for** i **in** range(9)]

### Routine to return indices of all cells in a column

**def** **column**(c:int) -> list:

"""Returns the list of tuples that specify the indices of

all squares for a given column.

This routine just makes it easier to iterate through

all squares of a column

Args:

c (int): the column whose indices are to be generated

Returns:

list: list of tuples specifying the column indices, eg.

[(0, 4), (1, 4), (2, 4), ... ]

"""

**return** [(i, c) **for** i **in** range(9)]

### Routine to return indices of all cells in a 3x3 sub-square

**def** **square**(ind: tuple) -> list:

"""Returns a list of tuples that specify the indices of all cells

inside a sub square

Args:

ind (tuple): specifies the indices of the top left cell of the

sub-square

Returns:

list: all cells in the sub-square. eg.

[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0),

(2, 1), (2, 2)]

"""

**return** [(i + ind[0], j + ind[1]) **for** i **in** range(3) **for** j **in** range(3)]

### Routine to return the indices of the top-left square of each 3x3 sub-square

**def** **square\_starts**() -> list:

"""Returns the index of the top left square of each cell

Returns:

list: [(0, 0), (0, 3), (0, 6), (3, 0), (3, 3), (3, 6),

(6, 0), (6, 3), (6, 6)]

Yields:

Iterator[list]: the tuple specifying the index

"""

**for** i **in** range(0,9,3):

**for** j **in** range(0,9,3): **yield** (i, j)

## Solution Printer

The solution printer validates the solution is indeed correct, and then prints it.

### Validate the solution is correct

**def** **validate\_all\_numbers\_present**(self, indices:dict):

"""validate that all numbers are present in the

set of indices provided

Args:

indices (dict): indices to look for all the numbers,

this could be all indices of a row,

all indices of a column, or all indices

of a sub-square

"""

s = set()

count = 0

**def** **update**(i, j, k):

**nonlocal** count, s

**if** self.Value(self.sudoku[i][j][k]):

count = count + 1

s.add(k)

[update(i, j, k) **for** i, j **in** indices **for** k **in** numbers()]

**if** (len(s) != 9 **or** count != 9):

print("Either all numbers not present, or some are repeated" + \

"in squares ", indices)

**assert**(9 == len(s) **and** 9 == count)

**def** **validate\_cell**(self, i, j):

"""Validate that each cell should have exactly one number

Args:

i ([type]): first axis index of the cell

j ([type]): second axis index of the cell

"""

# Each cell should have exactly one number

count = 0

**for** k **in** numbers():

**if** self.Value(self.sudoku[i][j][k]): count = count + 1

**if** (1 != count): print(f"sudoku[{i},{j}] has {count} values")

**assert**(count == 1)

**def** **validate\_solution**(self):

"""Validate few things things

1. for each cell, only one variable must be true

2. For each row, all numbers must be present, and only once

3. For each column, all numbers must be present and only once

4. For each sub-square, all numbers must be present and only once

"""

[self.validate\_cell(i, j) **for** i **in** range(9) **for** j **in** range(9)]

[self.validate\_all\_numbers\_present(row(i)) **for** i **in** range(9)]

[self.validate\_all\_numbers\_present(column(i)) **for** i **in** range(9)]

[self.validate\_all\_numbers\_present(square(sqs)) **for** \

sqs **in** square\_starts()]

### Print the solution

**def** **OnSolutionCallback**(self):

self.validate\_solution()

self.solutions = self.solutions + 1

print(f"Solution # {self.solutions}")

print("++=======++===+===+===+===+===+===+===+===+===++")

print("|| # ||-0-|-1-|-2-|-3-|-4-|-5-|-6-|-7 |-8-||")

print("++\*\*\*\*\*\*\*++\*\*\*+\*\*\*+\*\*\*+\*\*\*+\*\*\*+\*\*\*+\*\*\*+\*\*\*+\*\*\*++")

**for** i **in** range(9):

output\_line = f"|| {i} "

first = **True**

**for** j **in** range(9):

**for** k **in** numbers():

**if** self.Value(self.sudoku[i][j][k]):

**if** j % 3 == 0:

addstr = f" || {k}" **if** first **else** f" | {k}"

**else**:

addstr = f" || {k}" **if** first **else** f" . {k}"

first = **False**

output\_line = output\_line + addstr

**break**

output\_line = output\_line + f" ||"

print(output\_line)

**if** (i + 1) % 3 == 0:

print("++\*\*\*\*\*\*\*++\*\*\*\*\*\*\*\*\*\*\*+\*\*\*\*\*\*\*\*\*\*\*+\*\*\*\*\*\*\*\*\*\*\*++")

**else**:

print("++............................................++")

print()

print()