Decision Analytics
Assignment 1
Rajbir Bhattacharjee
R00195734

Contents

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()
```

Creating the Variables

The variables are a set of Boolean variables arranged in a 9x9x9 grid. For each row, for each column, the ith variable is true if i is in that (row, column).

```
def create_variables(model) -> dict:
"""Create the variables, the variables are a 3 D array
of 9x9x9
For each row, for each column, there are 9 boolean variables

Args:
model ([type]): [description]

Returns:
dict: dictionary of variables, can be indexed in a 3D way
"""
def get_inner_dict(i, j):
return {k: model.NewBoolVar(f"--[{i},{j}]->{k}--") for k in numbers()}

def get_outer_dict(i):
return {j: get_inner_dict(i, j) for j in range(9)}

return {i: get_outer_dict(i) for i in range(9)}
```

Constraints

The constraints use generic routines that can be reused. These helper routines will be described first

Helper Routine to ensure only one number per cell

```
def set_constraint_one_number_per_cell(model, sudoku:dict):
"""add constraint that for each row, column, only one of the variable
must be true. That is one cell can contain exactly one number
```

```
Args:
model ([type]): [description]
sudoku (dict): [description]
"""
for r in range(9):
for c in range(9):
model.AddBoolOr([sudoku[r][c][n] for n in numbers()])
```

Helper routine to ensure that there are no duplicates in a set of cells

```
def set_constraint_no_duplicates(model, sudoku:dict, indices):
"""Given a set of indices for cells (eg. all cells in one row,
or one column), ensure that there are no duplicates in those cells.

Args:
model ([type]): [description]
sudoku (dict): [description]
indices ([type]): indices of the cells in which there should
not be any duplicates
"""
def update(model, sudoku, i, j, n):
r1, c1 = indices[i]
r2, c2 = indices[j]
model.AddBoolOr(                                                \
[                                                       \
sudoku[r1][c1][n].Not(),                            \
sudoku[r2][c2][n].Not(),                            \
])

for i in range(len(indices)):
for j in range(i+1, len(indices)):
for n in numbers():
update(model, sudoku, i, j, n)
```

Helper routine to ensure that all numbers 1..9 are present in a set of squares

```
def set_constraint_all_numbers_present(model, sudoku:dict, indices):
"""Given a set of indices for cells (eg. all cells in one row,
or one column), ensure that all the numbers 1..9 are present.

Args:
model ([type]): [description]
sudoku (dict): [description]
indices ([type]): [description]
"""
for n in numbers():
model.AddBoolOr([sudoku[r][c][n] for r, c in indices])
```

Setting the Implicit Constraints

Finally the implicit constraints are set using these helper routines. Each row, column, and sub-square must satisfy the constraints.

```
# No duplicates in each row and each column
for i in range(9):
set_constraint_no_duplicates(model, sudoku, row(i))
set_constraint_no_duplicates(model, sudoku, column(i))
```

```
# No duplicates in each sub-square
for sqs in square_starts():
set_constraint_no_duplicates(model, sudoku, square(sqs))

# Every number in each row and each column
for i in range(9):
set_constraint_all_numbers_present(model, sudoku, row(i))
set_constraint_all_numbers_present(model, sudoku, column(i))

# Every number in each sub-square
for sqs in square_starts():
set_constraint_all_numbers_present(model, sudoku, square(sqs))
```

Setting the Explicit Constraints given in the problem Document

```
def set_explicit_constraints(model, sudoku:dict):
"""set the explicit constraints according to what is specified
in the assignment document

Args:
model ([type]): [description]
sudoku (dict): [description]
"""
explicit_constraints = \
{
0: {7: 3},
1: {0: 7, 2: 5, 4: 2},
2: {1: 9, 6: 4},
3: {5: 4, 8: 2},
4: {1: 5, 2: 9, 3: 6, 8: 8},
5: {0: 3, 4: 1, 7: 5},
6: {0: 5, 1: 7, 4: 6, 6: 1},
7: {3: 3},
8: {0: 6, 3: 4, 8: 5}
}

for r, val in explicit_constraints.items():
for c, n in val.items():
model.AddBoolAnd([sudoku[r][c][n]])

for i in range(9):
outstr = ""
for j in range(9):
try:
outstr = outstr + str(explicit_constraints[i][j]) + " "
except:
outstr = outstr + '. '
print(outstr)
```

Putting It All Together

```
def sudoku_main():
    model = cp_model.CpModel()

    # PART A: Identify the decision Variables
    sudoku = create_variables(model)
```

```python
    set_constraint_one_number_per_cell(model, sudoku)

    # Part C
    # No duplicates in each row and each column
    for i in range(9):
        set_constraint_no_duplicates(model, sudoku, row(i))
        set_constraint_no_duplicates(model, sudoku, column(i))

    # Part C
    # No duplicates in each sub-square
    for sqs in square_starts():
        set_constraint_no_duplicates(model, sudoku, square(sqs))

    # Part C
    # Every number in each row and each column
    for i in range(9):
        set_constraint_all_numbers_present(model, sudoku, row(i))
        set_constraint_all_numbers_present(model, sudoku, column(i))

    # Part C
    # Every number in each sub-square
    for sqs in square_starts():
        set_constraint_all_numbers_present(model, sudoku, square(sqs))

    # Part B: Specify the digits that were given in the problem
    set_explicit_constraints(model, sudoku)

    solver = cp_model.CpSolver()
    solution_printer = SudokuSolutionPrinter(sudoku)
    status = solver.SearchForAllSolutions(model, solution_printer)
    print(solver.StatusName(status))
```

Solution Printed

```
. . . . . . . 3 .
7 . 5 . 2 . . . .
. 9 . . . . 4 . .
. . . . . 4 . . 2
. 5 9 6 . . . . 8
3 . . . 1 . . 5 .
5 7 . . 6 . 1 . .
. . . 3 . . . . .
6 . . 4 . . . . 5


Solution # 1
++=======++===+===+===+===+===+===+===+===+===++
||   #   ||-0-|-1-|-2-|-3-|-4-|-5-|-6-|-7 |-8-||
++*******++***+***+***+***+***+***+***+***+***++
||   0   || 2 . 6 . 8 | 7 . 4 . 9 | 5 . 3 . 1 ||
++............................................++
||   1   || 7 . 4 . 5 | 1 . 2 . 3 | 6 . 8 . 9 ||
++............................................++
||   2   || 1 . 9 . 3 | 5 . 8 . 6 | 4 . 2 . 7 ||
++*******++***********+***********+***********++
||   3   || 8 . 1 . 7 | 9 . 5 . 4 | 3 . 6 . 2 ||
++............................................++
||   4   || 4 . 5 . 9 | 6 . 3 . 2 | 7 . 1 . 8 ||
++............................................++
||   5   || 3 . 2 . 6 | 8 . 1 . 7 | 9 . 5 . 4 ||
++*******++***********+***********+***********++
||   6   || 5 . 7 . 4 | 2 . 6 . 8 | 1 . 9 . 3 ||
++............................................++
||   7   || 9 . 8 . 1 | 3 . 7 . 5 | 2 . 4 . 6 ||
++............................................++
```

```
||   8   || 6 . 3 . 2 | 4 . 9 . 1 | 8 . 7 . 5 ||
++*******++***********+***********+***********++




Solution # 2
++=======++===+===+===+===+===+===+===+===+===++
||   #   ||-0-|-1-|-2-|-3-|-4-|-5-|-6-|-7 |-8-||
++*******++***+***+***+***+***+***+***+***+***++
||   0   || 2 . 6 . 1 | 9 . 4 . 8 | 5 . 3 . 7 ||
++............................................++
||   1   || 7 . 4 . 5 | 1 . 2 . 3 | 6 . 8 . 9 ||
++............................................++
||   2   || 8 . 9 . 3 | 7 . 5 . 6 | 4 . 2 . 1 ||
++*******++***********+***********+***********++
||   3   || 1 . 8 . 7 | 5 . 9 . 4 | 3 . 6 . 2 ||
++............................................++
||   4   || 4 . 5 . 9 | 6 . 3 . 2 | 7 . 1 . 8 ||
++............................................++
||   5   || 3 . 2 . 6 | 8 . 1 . 7 | 9 . 5 . 4 ||
++*******++***********+***********+***********++
||   6   || 5 . 7 . 8 | 2 . 6 . 9 | 1 . 4 . 3 ||
++............................................++
||   7   || 9 . 1 . 4 | 3 . 8 . 5 | 2 . 7 . 6 ||
++............................................++
||   8   || 6 . 3 . 2 | 4 . 7 . 1 | 8 . 9 . 5 ||
++*******++***********+***********+***********++


Solution # 3
++=======++===+===+===+===+===+===+===+===+===++
||   #   ||-0-|-1-|-2-|-3-|-4-|-5-|-6-|-7 |-8-||
++*******++***+***+***+***+***+***+***+***+***++
||   0   || 2 . 6 . 1 | 7 . 4 . 8 | 5 . 3 . 9 ||
++............................................++
||   1   || 7 . 4 . 5 | 9 . 2 . 3 | 6 . 8 . 1 ||
++............................................++
||   2   || 8 . 9 . 3 | 1 . 5 . 6 | 4 . 2 . 7 ||
++*******++***********+***********+***********++
||   3   || 1 . 8 . 7 | 5 . 9 . 4 | 3 . 6 . 2 ||
++............................................++
||   4   || 4 . 5 . 9 | 6 . 3 . 2 | 7 . 1 . 8 ||
++............................................++
||   5   || 3 . 2 . 6 | 8 . 1 . 7 | 9 . 5 . 4 ||
++*******++***********+***********+***********++
||   6   || 5 . 7 . 8 | 2 . 6 . 9 | 1 . 4 . 3 ||
++............................................++
||   7   || 9 . 1 . 4 | 3 . 8 . 5 | 2 . 7 . 6 ||
++............................................++
||   8   || 6 . 3 . 2 | 4 . 7 . 1 | 8 . 9 . 5 ||
++*******++***********+***********+***********++


Solution # 4
++=======++===+===+===+===+===+===+===+===+===++
||   #   ||-0-|-1-|-2-|-3-|-4-|-5-|-6-|-7 |-8-||
++*******++***+***+***+***+***+***+***+***+***++
||   0   || 2 . 6 . 1 | 8 . 4 . 7 | 5 . 3 . 9 ||
++............................................++
||   1   || 7 . 4 . 5 | 9 . 2 . 3 | 6 . 8 . 1 ||
++............................................++
||   2   || 8 . 9 . 3 | 1 . 5 . 6 | 4 . 2 . 7 ||
++*******++***********+***********+***********++
||   3   || 1 . 8 . 7 | 5 . 9 . 4 | 3 . 6 . 2 ||
++............................................++
||   4   || 4 . 5 . 9 | 6 . 3 . 2 | 7 . 1 . 8 ||
```

```
++.............................................++
||   5   || 3 . 2 . 6 | 7 . 1 . 8 | 9 . 5 . 4 ||
++*******++***********+***********+***********++
||   6   || 5 . 7 . 8 | 2 . 6 . 9 | 1 . 4 . 3 ||
++.............................................++
||   7   || 9 . 1 . 4 | 3 . 8 . 5 | 2 . 7 . 6 ||
++.............................................++
||   8   || 6 . 3 . 2 | 4 . 7 . 1 | 8 . 9 . 5 ||
++*******++***********+***********+***********++


Solution # 5
++=======++===+===+===+===+===+===+===+===+===++
||   #   ||-0-|-1-|-2-|-3-|-4-|-5-|-6-|-7 |-8-||
++*******++***+***+***+***+***+***+***+***+***++
||   0   || 2 . 8 . 6 | 7 . 4 . 9 | 5 . 3 . 1 ||
++.............................................++
||   1   || 7 . 4 . 5 | 1 . 2 . 3 | 6 . 8 . 9 ||
++.............................................++
||   2   || 1 . 9 . 3 | 5 . 8 . 6 | 4 . 2 . 7 ||
++*******++***********+***********+***********++
||   3   || 8 . 1 . 7 | 9 . 5 . 4 | 3 . 6 . 2 ||
++.............................................++
||   4   || 4 . 5 . 9 | 6 . 3 . 2 | 7 . 1 . 8 ||
++.............................................++
||   5   || 3 . 6 . 2 | 8 . 1 . 7 | 9 . 5 . 4 ||
++*******++***********+***********+***********++
||   6   || 5 . 7 . 4 | 2 . 6 . 8 | 1 . 9 . 3 ||
++.............................................++
||   7   || 9 . 2 . 1 | 3 . 7 . 5 | 8 . 4 . 6 ||
++.............................................++
||   8   || 6 . 3 . 8 | 4 . 9 . 1 | 2 . 7 . 5 ||
++*******++***********+***********+***********++
```

Project Planning

Reading the Excel

The Excel is read using pandas, and all the sheets are stored as dataframes:

```
# PART A: Read the Excel
def read_excel(self, excelfile:str) -> None:
"""Read an excel file and create dataframes for each sheet

Args:
excelfile (str): path to excel file
"""
# Load the excel with all the sheets
self.project_df = pd.read_excel(excelfile, sheet_name='Projects')
self.quote_df = pd.read_excel(excelfile, sheet_name='Quotes')
self.depend_df = pd.read_excel(excelfile, sheet_name='Dependencies')
self.value_df = pd.read_excel(excelfile, sheet_name='Value')

# Rename the columns
self.project_df.rename(columns={'Unnamed: 0':'Project'}, inplace=True)
self.quote_df.rename(columns={'Unnamed: 0':'Contractor'}, inplace=True)
self.depend_df.rename(columns={'Unnamed: 0':'Project'}, inplace=True)
self.value_df.rename(columns={'Unnamed: 0':'Project'}, inplace=True)
```

```
self.month_names = self.project_df.columns[1:].tolist()
self.job_names = self.quote_df.columns[1:].tolist()
self.project_names = self.project_df['Project'].tolist()
self.contractor_names = self.quote_df['Contractor'].tolist()
```

Variables

All variables are Boolean variables, and two sets of variables are created:

1. Projects (var_p) : This set of variables is organized as a linear array of variables, one for each project. If a project is chosen, then the variable is true.
2. Project-month-contractor-job (var_pmjc): These are a set of variables that are organized as a 4D grid. If a contractor picks up a job for a particular project in a particular month, then that entry is set to true, else false.

```
# PART B: Create teh variables
# For each project picked, have a T/F variable
def crtvars_p(self):
"""Create the 1D array of variables for which projects are picked up
"""
# Create a single variable for each project
# Also lookup the dependencies DF and add constraints accordingly
for p in self.project_names:
self.var_p[p] = self.model.NewBoolVar(f"{p}")
```

```
# PART B: Create teh variables
# Have a 4-D array of T/F variables, the dimensions specify the
# following:
# PROJECTS, MONTHS, JOBS, CONTRACTORS
# if a contractor picks up a job in a month for a project, then the
# corresponding variable is set to True
def crtvars_pmjc(self):
"""Create the 4D matrix of variables
PROJECTS, MONTHS, JOBS, CONTRACTORS
are the 4 axes
"""
# 4-D array of variables: Project, Month, Job, Contractor
for project in self.project_names:
prj_variables = {}
for month in self.month_names:
mnth_variables = {}
for job in self.job_names:
job_variables = {}
for contractor in self.contractor_names:
job_variables[contractor] = self.model.NewBoolVar(   \
f"{project}-{month}-{job}-{contractor}")
mnth_variables[job] = job_variables
prj_variables[month] = mnth_variables
self.var_pmjc[project] = prj_variables
```

Constraint: Not all contractors can do all jobs

If a contractor cannot do a job, then for that contractor and that job, the variables for all projects and months are set to false.

```
# PART B-2
# Add constraints to account for the fact that not all contractors
# can do all jobs
def crtcons_job_contractor(self)->None:
"""All contractors cannot do all jobs. If a contractor
cannot do a job, then the cost for that contractor and job in the
excel sheet is NaN.
"""
# Not all contractors can do all jobs

def add_constraint(c:str, j:str):
"""Add a constraint that contractor c cannot do job j
This is a simple constraint of negation

Args:
c (str): contractor (name)
j (str): job (name)
"""
# Given c, j set to false All p, m
variables = []
for p in self.project_names:
for m in self.month_names:
variables.append(self.var_pmjc[p][m][j][c].Not())
self.model.AddBoolAnd(variables)

for c in self.contractor_names:
for j in self.job_names:
cost = self.get_contractor_job_cost(c, j)
cannotdo = math.isnan(cost)
if cannotdo:
add_constraint(c, j)
```

Constraint: Contractors cannot work on two projects simultaneously

Here, the sum of all Booleans for a contractor should be less than or equal to 1. This simplifies the constraint.

```
# PART C: Contractors cannot work on two projects simultaneously
def crtcons_contractor_single_simult_project(self):
"""Implement a constraint that a contractor cannot work on two
projects on the same month.
For every contractor and month, the sum of jobs in all projects
must be at most 1
"""
# This constraint can be simplified, since a contractor can only do
# one project at a time, that implies he can only do one job
# at a time, and vice versa
# So we'll replace this by adding a constrint for one simultaneous job
# For each month, for each contractor -> count of jobs = 1
for m in self.month_names:
```

```
for c in self.contractor_names:
variables = []
for p in self.project_names:
for j in self.job_names:
variables.append(self.var_pmjc[p][m][j][c])
self.model.Add(sum(variables) <= 1)
```

Constraint: Only one constructor should be assigned to a job

Here, again, a sum of variables is used, along with a channeling constraint that is true only if the project is picked up in the first place. Otherwise we don't really care.

```
# PART D-1: Only one contractor should be assined to a job
# (for a project and month)
def crtcons_one_contractor_per_job(self)->None:
"""Only one contractor per job needs to work on it
"""
# Only one contractor per job

def add_constraint(p:str, j:str):
# Only one contractor per job for every project, in a given month
for m in self.month_names:
variables = []
for c in self.contractor_names:
variables.append(self.var_pmjc[p][m][j][c])
self.model.Add(sum(variables) <= 1).OnlyEnforceIf(self.var_p[p])

# rltn is a hashmap where every item is
# Project => [(month1, job1), (month2, job2), ...]
rltn = self.get_project_job_month_relationships()
for p, mjlist in rltn.items():
for m, j in mjlist:
add_constraint(p, j)
```

Constraint: If a project is not selected no one should work on it

Here, again, the sum of variables is used along with a channeling constraint

```
# PART E: If a project is not selected, no one should work on it
def crtcons_project_not_selected(self):
"""If a project is not selected, then no one should work on it
This means that

Not ProjectX => sum(all months, jobs, contractors for ProjectX) == 0
"""
# If a project is not selected none of its jobs should
# be done
for p in self.project_names:
variables = []
for m in self.month_names:
for j in self.job_names:
for c in self.contractor_names:
variables.append(self.var_pmjc[p][m][j][c])
self.model.Add(sum(variables) == 0)                            \
.OnlyEnforceIf(self.var_p[p].Not())
```

Constraint: Some projects have dependencies between them

```
# PART F: Add constraints for dependencies between projects
def crtcons_project_dependencies_conflicts(self):
"""Add constraints for dependencies between projects.
We have already stored the dependencies of projects in the
dataframe self.depend_df
"""
def add_required_dependency(p1:str, p2:str)->None:
# p1 implies p2
self.model.AddBoolOr(                                           \
[                                                       \
self.var_p[p1].Not(),                               \
self.var_p[p2]                                      \
])

def add_conflict_dependency(p1:str, p2:str)->None:
# P1 is incompatible with P2, so both of them cannot be TRUE
# NOT(A AND B) is written here as NOT A OR NOT B
self.model.AddBoolOr(                                           \
[                                                       \
self.var_p[p1].Not(),                               \
self.var_p[p2].Not()                                \
])

for p1 in self.project_names:
for p2 in self.project_names:
row_p1 = self.depend_df[self.depend_df['Project'] == p1]
e_p1_p2 = row_p1[p2].tolist()[0]
if (isinstance(e_p1_p2, str)):
if ('required' == e_p1_p2.lower()):
add_required_dependency(p1, p2)
if ('conflict' == e_p1_p2.lower()):
add_conflict_dependency(p1, p2)
```

Constraint: Profit margin should be at least 2160

```
# PART G: Add constraints so that difference between the value of
# projects delivered and the cost of all contractors is at least 2160
def crtcons_profit_margin(self, margin:int)->None:
"""Add constraints so that the difference between the value
of the projects delivered and the cost of all contractors is
at least margin.

Args:
margin (int): the margin
"""
revenue = []         # This is the value of the projects delivered
for p in self.project_names:
revenue.append(self.get_project_value(p) * self.var_p[p])

expenses = []        # This is the cost of all contractors
for p in self.project_names:
for m in self.month_names:
for j in self.job_names:
for c in self.contractor_names:
var = self.var_pmjc[p][m][j][c]
cost = self.get_contractor_job_cost(c, j)
cost = 0 if math.isnan(cost) else int(cost)
expenses.append(cost * var)

# Add the constraint that revenue is at least expenses + margin
self.model.Add(sum(revenue) >= sum(expenses) + margin)
```

Implicit Constraint: If a job is picked up in the 4D array, then its project must also be picked up in the 1D array

```
# Implicit constraint, if any contractor picks up any job
# for any project in any month in the 4D array, then the project
# must have been picked up in the 1D projects array
def crtcons_pmjc_p(self):
"""Implicit constraint, if any contractor picks up any job
for any project in any month in the 4D array, then the project
must have been picked up in the 1D projects array
"""
# if an entry in pmjc is True then the corresponding entry in p must
# also be true
for p in self.project_names:
for m in self.month_names:
for c in self.contractor_names:
for j in self.job_names:
self.model.AddBoolOr(                              \
[                                            \
self.var_pmjc[p][m][j][c].Not(),         \
self.var_p[p],                           \
])
```

Implicit Constraint: If a project is picked up in the 1D array, all jobs required to do the project must be done in the 4D array, in the month in which it needs to be done, by any contractor

```
# Implicit constraint, if a project is selected, then
# all jobs for the project must be done
# Also, only one contractor can do job in a month
def crtcons_complete_all_jobs_for_project(self):
"""Implicit constraint, if a project is selected, then
all jobs for the project must be done
Also, only one contractor can do a job in a month. This is
ensured by taking the sum of the variables for all contractors, for
a specified contractor, project and month
"""
# If a project is selected, then each job for the project must be
# done in the month specified
def add_constraint(p, j, m):
cvars = [self.var_pmjc[p][m][j][c] for c in self.contractor_names]
self.model.Add(sum(cvars) == 1).OnlyEnforceIf(self.var_p[p])

# rltn is a dictionary of the following format
# project -> [(month1, job1), (month2, job2), ...]
rltn = self.get_project_job_month_relationships()
for p, monthjoblist in rltn.items():
for monthjob in monthjoblist:
m, j = monthjob
add_constraint(p, j, m)
```

Solution

```
phantom@CND7182VPXZBOOK:/mnt/c/source/mtu-decision-analytics-assignment/assignment1$
./projects.py
```

```
Created all variables, calling solver...
Solution # 01
---------------------------
Project C [('M4', 'Job H', 'Contractor H'), ('M5', 'Job E', 'Contractor E'), ('M6', 'Job G', 'Contractor G'), ('M7', 'Job B', 'Contractor E'), ('M8', 'Job E', 'Contractor E')]
Project D [('M2', 'Job D', 'Contractor H'), ('M3', 'Job F', 'Contractor F'), ('M4', 'Job I', 'Contractor G'), ('M5', 'Job H', 'Contractor H')]
Project E [('M8', 'Job J', 'Contractor C'), ('M9', 'Job A', 'Contractor A')]
Project H [('M8', 'Job A', 'Contractor A'), ('M9', 'Job B', 'Contractor E'), ('M10', 'Job D', 'Contractor H'), ('M11', 'Job I', 'Contractor G')]
Project I [('M10', 'Job L', 'Contractor A'), ('M11', 'Job F', 'Contractor F'), ('M12', 'Job K', 'Contractor B')]
-------------
Profit = 2165.0


Solution # 02
---------------------------
Project C [('M4', 'Job H', 'Contractor H'), ('M5', 'Job E', 'Contractor A'), ('M6', 'Job G', 'Contractor G'), ('M7', 'Job B', 'Contractor E'), ('M8', 'Job E', 'Contractor E')]
Project D [('M2', 'Job D', 'Contractor H'), ('M3', 'Job F', 'Contractor F'), ('M4', 'Job I', 'Contractor G'), ('M5', 'Job H', 'Contractor H')]
Project E [('M8', 'Job J', 'Contractor C'), ('M9', 'Job A', 'Contractor A')]
Project H [('M8', 'Job A', 'Contractor A'), ('M9', 'Job B', 'Contractor E'), ('M10', 'Job D', 'Contractor H'), ('M11', 'Job I', 'Contractor G')]
Project I [('M10', 'Job L', 'Contractor D'), ('M11', 'Job F', 'Contractor F'), ('M12', 'Job K', 'Contractor B')]
-------------
Profit = 2165.0


Solution # 03
---------------------------
Project C [('M4', 'Job H', 'Contractor H'), ('M5', 'Job E', 'Contractor E'), ('M6', 'Job G', 'Contractor G'), ('M7', 'Job B', 'Contractor E'), ('M8', 'Job E', 'Contractor E')]
Project D [('M2', 'Job D', 'Contractor H'), ('M3', 'Job F', 'Contractor F'), ('M4', 'Job I', 'Contractor G'), ('M5', 'Job H', 'Contractor H')]
Project E [('M8', 'Job J', 'Contractor C'), ('M9', 'Job A', 'Contractor A')]
Project H [('M8', 'Job A', 'Contractor A'), ('M9', 'Job B', 'Contractor E'), ('M10', 'Job D', 'Contractor H'), ('M11', 'Job I', 'Contractor G')]
Project I [('M10', 'Job L', 'Contractor D'), ('M11', 'Job F', 'Contractor F'), ('M12', 'Job K', 'Contractor K')]
-------------
Profit = 2165.0


Solution # 04
---------------------------
Project C [('M4', 'Job H', 'Contractor H'), ('M5', 'Job E', 'Contractor E'), ('M6', 'Job G', 'Contractor G'), ('M7', 'Job B', 'Contractor E'), ('M8', 'Job E', 'Contractor E')]
Project D [('M2', 'Job D', 'Contractor H'), ('M3', 'Job F', 'Contractor F'), ('M4', 'Job I', 'Contractor G'), ('M5', 'Job H', 'Contractor H')]
Project E [('M8', 'Job J', 'Contractor C'), ('M9', 'Job A', 'Contractor A')]
Project H [('M8', 'Job A', 'Contractor A'), ('M9', 'Job B', 'Contractor E'), ('M10', 'Job D', 'Contractor H'), ('M11', 'Job I', 'Contractor G')]
Project I [('M10', 'Job L', 'Contractor D'), ('M11', 'Job F', 'Contractor F'), ('M12', 'Job K', 'Contractor B')]
-------------
Profit = 2175.0


Solution # 05
---------------------------
Project A [('M1', 'Job A', 'Contractor A'), ('M2', 'Job B', 'Contractor E'), ('M3', 'Job C', 'Contractor K')]
Project C [('M4', 'Job H', 'Contractor H'), ('M5', 'Job E', 'Contractor E'), ('M6', 'Job G', 'Contractor G'), ('M7', 'Job B', 'Contractor E'), ('M8', 'Job E', 'Contractor E')]
Project D [('M2', 'Job D', 'Contractor H'), ('M3', 'Job F', 'Contractor F'), ('M4', 'Job I', 'Contractor G'), ('M5', 'Job H', 'Contractor H')]
Project E [('M8', 'Job J', 'Contractor C'), ('M9', 'Job A', 'Contractor A')]
```

```
Project H [('M8', 'Job A', 'Contractor A'), ('M9', 'Job B', 'Contractor E'), ('M10', 'Job D',
'Contractor H'), ('M11', 'Job I', 'Contractor G')]
Project I [('M10', 'Job L', 'Contractor D'), ('M11', 'Job F', 'Contractor F'), ('M12', 'Job K',
'Contractor B')]
-------------
Profit = 2165.0


OPTIMAL
5 solutions
```