Computational Complexity Assignment I

Date Yao Faustin Dieudonne
t201d047@gunma-u.ac.jp

Teacher in Charge: Kazuyuki Amano
Title: Professor

Department of Electronics and Informatics
Graduate School of Science and Technology, Gunma University
Japan
June 8, 2020

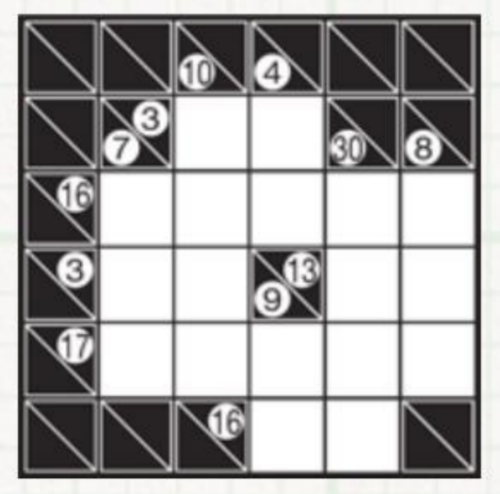
This report is divided in two parts. In the first part, we present our solution to the problem of formulating an instance of the kakuro puzzle [1] as an integer programming problem. In the second part we present a formulation of the integer partition problem also as an integer programming problem. As we have seen during lecture, an integer program is a linear program where all the variables have integrality constraints. Although integer programming is know to be NP-complete, the the Gurobi Optimizer is capable of solving such models using state-of-the-art mathematics and computer science [2]. We use the Python API provided with the Gurobi Optimizer to formulate and solve the problems. The jupyter notebook containing the code and the generated files for each model are available at [3].

1 Solving an instance of Kakuro with integer programming

The rules of the kakuro puzzle [1] are simple:

1. Each cell can contain integers from 1 through 9
2. The clues in the black cells tell the sum of the numbers next to that clue. (on the right or down)
3. The numbers in consecutive white cells must be unique.

The specific instance that we we solve here is given in image below.



1.1 Model Formulation

1.1.1 Decision variables

We choose to number the cells from top to bottom, left to right starting from 1. Thus the decision variables for this model are given by $assign_{ijv}$ for $(i, j, v) \in \{1, 2, 3, 4, 5, 6\}^2 \times \{1, \ldots, 9\}$. If $assign_{ijv} = 1$ then the cell with coordinates (i, j) is assigned the value v, otherwise $assign_{ijv} = 0$.

1.1.2 Constraints

The indications in the rules each give rise to some constraints. Fisrtly, since each cell must be assigned exactly one number we know that for all blank cells (i, j)

$$\sum_{v=1}^{9} assign_{ijv} = 1$$

Secondly, if two cells (i, j) and (k, l) belong to the same horizontal or vertical summation group then they must have different values assigned to each of them. We express this constraint as follow:

$$assign_{ijv} + assign_{klv} \leq 1$$

for each v. The inequality means that the value v may not be assigned to any of the two cells under consideration.

Finally, there are the constraints generated from the clues in the black cells.

1.2 Model deployment

We begin by importing the Gurobi Python Module. Then we initialize some data structures with the given data.

[1]:
```
import sys
import itertools

import gurobipy as gb

# A list of allowed values for the cells
values = list(range(1,10))

# A list of the blank cells
cells = [
           (2,3), (2,4),
    (3,2), (3,3), (3,4), (3,5), (3,6),
    (4,2), (4,3),        (4,5), (4,6),
    (5,2), (5,3), (5,4), (5,5), (5,6),
                  (6,4), (6,5)
]

# An encoding of the horizontal clues.
# The values for the cells in each sublist must must add up to the key.
data_horizontal = {
    3: [[(2,3), (2,4)],
        [(4,2), (4,3)]],
    16: [[(3,2), (3,3), (3,4), (3,5), (3,6)],
         [(6,4), (6,5)]],
```

```
    13: [[(4,5), (4,6)]],
    17: [[(5,2), (5,3), (5,4), (5,5), (5,6)]]
}

# An encoding of the vertical clues.
# The values for the cells in each sublist must must add up to the key.
data_vertical = {
    7: [[(3,2), (4,2), (5,2)]],
    10: [[(2,3), (3,3), (4,3), (5,3)]],
    4: [[(2,4), (3,4)]],
    9: [[(5,4), (6,4)]],
    30: [[(3,5), (4,5), (5,5), (6,5)]],
    8: [[(3,6), (4,6), (5,6)]]
}
```

Next, we instanciate a model, declare the decision variables and add the necessary constraints. We set the objective function to a constant. Then we instruct the Gurobi Optimizer to find an assignment of the variables that sasisfies the constraints.

[2]:
```
# declare and initialize model
m = gb.Model('kakuro')

# Decision variables for cell values
assign = m.addVars(cells, values, name="assign", vtype=gb.GRB.BINARY)

# Each cell must be assigned exactly one value
unique_assignement = m.addConstrs((assign.sum(x, y, '*') == 1
                                   for x, y in cells), 'unique_assignment')

# Horizontal clues constraints
for total, lists in data_horizontal.items():
    for i, l in enumerate(lists):
        expr = gb.LinExpr()
        for row, col in l:
            for v in values:
                expr.add(assign[row, col, v]*v)
        m.addConstr(expr == total, "hor_{}_{}".format(total, i))

# Vertical clues constraints
for total, lists in data_vertical.items():
    for i, l in enumerate(lists):
        expr = gb.LinExpr()
        for row, col in l:
            for v in values:
                expr.add(assign[row, col, v]*v)
        m.addConstr(expr == total, "vert_{}_{}".format(total, i))
```

```python
# Constraints for unicity of values in consecutive white cells
for lists in data_vertical.values():
    for l in lists:
        for (x1, y1), (x2, y2) in itertools.combinations(l, 2):
            m.addConstrs(assign[x1, y1, v] + assign[x2, y2, v] <= 1
                         for v in values)

for lists in data_horizontal.values():
    for l in lists:
        for (x1, y1), (x2, y2) in itertools.combinations(l, 2):
            m.addConstrs(assign[x1, y1, v] + assign[x2, y2, v] <= 1
                         for v in values)

# Set objective function to a constant
m.setObjective(1)

# Save generated model for inspection.
# available at https://github.com/faustind/gurobipy/blob/master/src/files/
 ↪kakuro.lp
m.write('files/kakuro.lp')

m.optimize()
```

```
Using license file /home/faustind/gurobi.lic
Academic license - for non-commercial use only
Gurobi Optimizer version 9.0.2 build v9.0.2rc0 (linux64)
Optimize a model with 426 rows, 162 columns and 1278 nonzeros
Model fingerprint: 0x5640372b
Variable types: 0 continuous, 162 integer (162 binary)
Coefficient statistics:
  Matrix range     [1e+00, 9e+00]
  Objective range  [0e+00, 0e+00]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 3e+01]
Presolve removed 426 rows and 162 columns
Presolve time: 0.00s
Presolve: All rows and columns removed

Explored 0 nodes (0 simplex iterations) in 0.03 seconds
Thread count was 1 (of 4 available processors)

Solution count 1: 1

Optimal solution found (tolerance 1.00e-04)
Best objective 1.000000000000e+00, best bound 1.000000000000e+00, gap 0.0000%
```

1.3 Cell Assignment

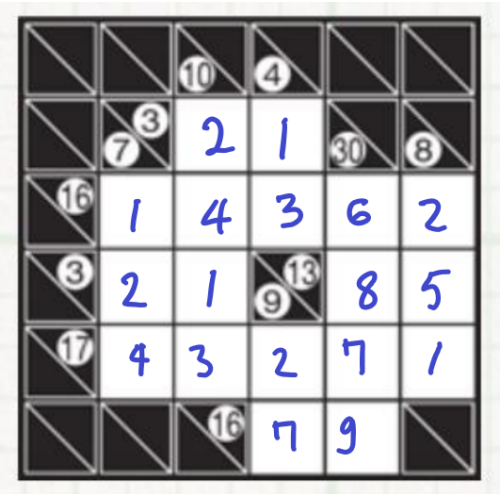
The optimization results in the assignment of appropriate values to each cell.

[3]:
```
# display the values assigned to each cell
for i,j,v in assign.keys():
    if (abs(assign[i,j,v].x) >= sys.float_info.epsilon):
        print('Cell({}, {}) = {}'.format(i,j,v))
```

```
Cell(2, 3) = 2
Cell(2, 4) = 1
Cell(3, 2) = 1
Cell(3, 3) = 4
Cell(3, 4) = 3
Cell(3, 5) = 6
Cell(3, 6) = 2
Cell(4, 2) = 2
Cell(4, 3) = 1
Cell(4, 5) = 8
Cell(4, 6) = 5
Cell(5, 2) = 4
Cell(5, 3) = 3
Cell(5, 4) = 2
Cell(5, 5) = 7
Cell(5, 6) = 1
Cell(6, 4) = 7
Cell(6, 5) = 9
```

1.4 Puzzle Solution

From the above assignments, we can solve the puzzle as shown in the image below.



2 The Integer Partition Problem

In the *integer partition* problem we seek to partition the elements of a set S in two sets A and B such that $\sum_{a \in A} a = \sum_{b \in B} b$ or alternatively make the difference as small as possible [4].

2.1 Model Formulation

2.1.1 Decision Variables

For each $x_i \in S$ we create two variables in_A^i and in_B^i. These variables can only take values in $\{0, 1\}$. And $in_P^i = 1$ if and only if $x_i \in P$ for $P \in \{A, B\}$.

2.1.2 Constraints

An element can only belong to one set in the resulting partition. So for each $x_i \in S$

$$in_A^i + in_B^i = 1.$$

In addition, the difference must be minimized, but remain greater than 0.

$$\sum_{a \in A} a - \sum_{b \in B} b \geq 0$$

2.1.3 Objective

Since a solution to the "strict" partition problem does not always exists, we will try to minimize the difference of the two partitions. When the solution exits, the difference is equal to $ 0 $ and it is the smallest it can get because of the non-negativity constraint.

$$Minimize \sum_{a \in A} a - \sum_{b \in B} b$$

2.2 Model Deployment

We begin by importing the necessary modules. Next we define a reusable function to run a model. The function `integer_partition` takes a list of integers defining an instance of the integer partition problem. It declares declare the decision variables then adds the constraints. The function `print_solution` just prints a solution returned by the function `integer_partition`.

[4]:
```
import sys
import random
import gurobipy as gb

from collections import defaultdict
```

```
from gurobipy import GRB


def integer_partition(S, fname=None):
    """Return two lists solution to the integer partition on S.

    Args:
        S: iterable of integers
        fname: OPTIONAL output file for the generated model
    """

    N = len(S)

    m = gb.Model('IP_{}'.format(fname))

    # decision variables for model integer partition
    partition = m.addVars(['A', 'B'], range(N)
                          , name="in"
                          , vtype=GRB.BINARY)

    # each elements must be in exactly one set in the partition
    separation = m.addConstrs((partition['A', i] + partition['B', i] == 1 for i␣
↪in range(N))
                              , name="separation")

    # Constraint to keep the difference non negative
    min_diff = m.addConstr(sum(S[i]*partition['A',i]
                               for i in range(N))
                           -sum(S[i]*partition['B',i]
                                for i in range(N)) >= 0
                          , name='minimize_difference')

    m.setObjective(sum(S[i]*partition['A',i] for i in range(N))
                   - sum(S[i]*partition['B',i] for i in range(N))
                   , GRB.MINIMIZE)

    # Save generated model for inspection.
    m.write('files/{}'.format(fname or 'integer_partition.lp'))
    m.optimize()
    result = defaultdict(list)

    # add each element of nums to the set to which it belongs in the partition
    for s, i in partition:
        if partition[s, i].x >= sys.float_info.epsilon:
            result[s].append(S[i])

    return result
```

```
def print_solution(result):
    """Print a solution to the integer partition problem."""
    for s, items in result.items():
        print('sum({}) = sum({}) = {}'.format(s, items, sum(items)))
```

2.3 Running the model

We now run the model on two instances, one for which we know it has at least two solutions and the other, a random instance with no exact solution.

[5]:
```
S = [423, 779, 434, 371, 244, 245, 753, 519, 106, 167, 34, 650,
     865, 605, 441, 190, 774, 512, 970, 394, 518, 887, 908, 971, 14]

result = integer_partition(S, 'ip_feasible.lp')

print('\nResult partition is: \n')
print_solution( result )
```

```
Gurobi Optimizer version 9.0.2 build v9.0.2rc0 (linux64)
Optimize a model with 26 rows, 50 columns and 100 nonzeros
Model fingerprint: 0x7190bcc4
Variable types: 0 continuous, 50 integer (50 binary)
Coefficient statistics:
  Matrix range     [1e+00, 1e+03]
  Objective range  [1e+01, 1e+03]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 1e+00]
Found heuristic solution: objective 1434.0000000
Presolve removed 26 rows and 50 columns
Presolve time: 0.00s
Presolve: All rows and columns removed

Explored 0 nodes (0 simplex iterations) in 0.03 seconds
Thread count was 1 (of 4 available processors)

Solution count 2: 0

Optimal solution found (tolerance 1.00e-04)
Best objective 0.000000000000e+00, best bound 0.000000000000e+00, gap 0.0000%

Result partition is:

sum(A) = sum([779, 244, 190, 512, 970, 394, 518, 887, 908, 971, 14]) = 6387
sum(B) = sum([423, 434, 371, 245, 753, 519, 106, 167, 34, 650, 865, 605, 441,
774]) = 6387
```

[6]:
```
S = [372, 734, 954, 124, 985, 759, 785, 462, 522, 70, 204,
     751, 343, 57, 152, 209, 724, 405, 867, 177, 701]

result = integer_partition(S, 'ip_infeasible.lp')

print('\nResult partition is: \n')
print_solution( result )
```

```
Gurobi Optimizer version 9.0.2 build v9.0.2rc0 (linux64)
Optimize a model with 22 rows, 42 columns and 84 nonzeros
Model fingerprint: 0x39dc88a7
Variable types: 0 continuous, 42 integer (42 binary)
Coefficient statistics:
  Matrix range     [1e+00, 1e+03]
  Objective range  [6e+01, 1e+03]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 1e+00]
Found heuristic solution: objective 3063.0000000
Presolve removed 22 rows and 42 columns
Presolve time: 0.00s
Presolve: All rows and columns removed

Explored 0 nodes (0 simplex iterations) in 0.03 seconds
Thread count was 1 (of 4 available processors)

Solution count 2: 1

Optimal solution found (tolerance 1.00e-04)
Best objective 1.000000000000e+00, best bound 1.000000000000e+00, gap 0.0000%

Result partition is:

sum(A) = sum([759, 785, 343, 57, 152, 209, 724, 405, 867, 177, 701]) = 5179
sum(B) = sum([372, 734, 954, 124, 985, 462, 522, 70, 204, 751]) = 5178
```

The results of running the model on the two instances, show that our implementation is capable of finding an exact soluion when it exists. And if it does not, it finds the best approximation as examplified by the last instance.

3 References

[1] The kakuro puzzle, https://www.puzzle-kakuro.com/
[2] The Gurobi Optimizer, https://www.gurobi.com/products/gurobi-optimizer/
[3] Source code and files repository, https://github.com/faustind/gurobipy
[4] The Algorithm Design Manual 13.10, 2nd Steven S. Skiena