MINI PROJECT – FUNDAMENTALS OF OPTIMIZATION

SEMESTER EXAM SCHEDULE

Group 18

1. *Tran Thanh Truong – 20214938*
2. *Hoang Dinh Dung – 20214882*
3. *Phan Cong Anh – 20210078*

TABLE OF CONTENTS

1. INTRODUCTION AND DESCRIBTION

Scheduling semester exam is one of the most popular task at school. To help teachers do this work more conveniently, we give some algorithms that will be explained in the next section to optimize this problem.

PROBLEM DESCRIBTION

- N: Number of courses to be scheduled
- d(i): Number of students participate in course n(i)
- M: Number of rooms
- c(j): Number of seats of room m(j)
- K: Number of of pairs of courses cannot be grouped together (conflicting courses)
- Pairs of conflicting courses

```
20
45 39 54 58 23 51 27 47 52 28 48 32 48 53 37 50 27 30 54 25
6
29 44 24 48 60 26
10
1 15
7 10
1 6
13 10
19 7
15 9
16 17
20 7
16 14
15 12
```

We have four periods in a day and the problem is minimizing the number of days scheduled for examination.

Data Generator

```
def genData(N, M, turnoutRange, capacityRange, K):
    assert False not in [arg > 0 for arg in (N, M, *turnoutRange, *capacityRange)], 'N, M, *turnoutRange, and *capacityRange should be positive integers.'
    assert K >= 0 and K <= comb(N, 2), 'K should be a natural number at most N choose 2.'

    turnouts = [str(rd.randint(turnoutRange[0], turnoutRange[1])) for i in range(N)]
    #generate a number of large halls which can occupy all candidates of any exam and a number of halls which cannot
    numLargeHalls = rd.randint(1, M)
    smallHalls = [str(rd.randint(capacityRange[0], turnoutRange[1])) for i in range(M - numLargeHalls)]
    largeHalls = [str(rd.randint(turnoutRange[1], capacityRange[1])) for i in range(numLargeHalls)]
    capacities = smallHalls + largeHalls
    rd.shuffle(capacities)
    #generate all possible pairs of exams with common candidates and pick K random pairs
    conflicts = list(combinations(range(1, N + 1), 2))
    rd.shuffle(conflicts)
    conflicts = [[str(i), str(j)] for i, j in conflicts[:K]]
    for pair in conflicts:
        rd.shuffle(pair)
```

Read Data

```
def readData(filename):
    with open(filename) as f:
        content = [[int(j) for j in i.split()] for i in f.read().splitlines()]
    N, d, M, c, K = content[0][0], content[1], content[2][0], content[3], content[4][0]
    p = [[content[5 + i][0] - 1, content[5 + i][1] - 1] for i in range(K)]
    print(f'N = {N}', f'd = {d}', f'M = {M}', f'c = {c}', f'K = {K}', f'p = {p}', sep = '\n')
    return N, d, M, c, K, p
```

2. MODELLING AND ALGORITHMS IMPLEMENTATION

MIXED INTEGER PROGRAMMING

CONSTRAINT PROGRAMMING

BACKTRACKING

UNDERSTANDING

HEURISTIC 1

HEURISTIC 3

HEURISTIC 2

• MIXED INTEGER PROGRAMMING

1. Define variables

- $X[i][j][k]$: variable that put course n(i) to room m(j) at period k

$$0 \le i, k \le N, 0 \le j \le M, X[i][j][k] \in \{0; 1\}$$

- y: number of periods

- C: list of conflicting courses

2. Constraints

- **Constraint 1: Pairs of conflicting courses cannot be put in the same period**

$$\mathbf{0} \le \boldsymbol{X}[\boldsymbol{i1}][\boldsymbol{j}][\boldsymbol{k}] + \boldsymbol{X}[\boldsymbol{i2}][\boldsymbol{j}][\boldsymbol{k}] \le \mathbf{1}, \forall(\boldsymbol{i1}, \boldsymbol{i2}) \in \boldsymbol{C};\ \boldsymbol{j} = \mathbf{0}, \dots, \boldsymbol{M} - \mathbf{1}; \boldsymbol{k} = \mathbf{0}, \dots, \boldsymbol{N} - \mathbf{1}$$

```python
# Constraint 1: Pairs of conflicting courses may not be put in the same period
for i in range(K):
  u, v = p[i][0], p[i][1]
  for k in range(N):
    constraint = mip_solver.Constraint(0, 1)
    for j1 in range(M):
      for j2 in range(M):
        if j1 != j2:
          constraint.SetCoefficient(x[u][j1][k], 1)
          constraint.SetCoefficient(x[v][j2][k], 1)
```

- **Constraint 2: An course room be putted at most one course in a period**

$$0 \le \sum_{i=0}^{N-1} X[i][j][k] \le 1, \forall j = 0, \ldots, M-1; k = 0, \ldots, N-1$$

```
# Constraint 2: An course room may be assigned at most one course in a period
for j in range(M):
  for k in range(N):
    constraint = mip_solver.Constraint(0, 1)
    for i in range(N):
      constraint.SetCoefficient(x[i][j][k], 1)
```

- **Constraint 3: The number of periods**

$$-\infty \le k \times X[i][j][k] - y \le 0, \forall j = 0, \ldots, M-1; k = 0, \ldots, N-1$$

```
# Constraint 3: The number of periods (k.x[i,j,k] - y <= 0)
for i in range(N):
  for j in range(M):
    for k in range(N):
      constraint = mip_solver.Constraint(-INF, 0)
      constraint.SetCoefficient(y, -1)
      constraint.SetCoefficient(x[i][j][k], k)
```

- **Constraint 4: A course be conducted at most one time in an room**

$$0 \le \sum_{j=0}^{M-1} \sum_{k=0}^{N-1} X[i][j][k] \le 1, \forall i = 0, \dots, N-1$$

```python
# Constraint 4: A course may be conducted at most one time in an course room
for i in range(N):
    constraint = mip_solver.Constraint(1, 1)
    for j in range(M):
        for k in range(N):
            constraint.SetCoefficient(x[i][j][k], 1)
```

- **Constraint 5: A course n(i) must be put into a room m(j) with capacity c(j)**

$$0 \le \sum_{k=0}^{N-1} X[i][j][k] \times d[i] \le c[j], \forall i = 0, \dots, N-1; j = 0, \dots, M-1$$

```python
# Constraint 5: A course n_i must be put into a room m_j with capacity c(j)
for i in range(N):
    for j in range(M):
        constraint = mip_solver.Constraint(0, c[j])
        for k in range(N):
            constraint.SetCoefficient(x[i][j][k], d[i])
```

3. Objective

$$\Longrightarrow Minimize(y)$$

```
# Define objective
obj = mip_solver.Objective()
obj.SetCoefficient(y, 1)
obj.SetMinimization()
```

• CONSTRAINT PROGAMMING

1. Define variables

- $X[i]$: period of course n(i)

$$i \in \{1, 2, \dots, n\}, X[i] \in \{1, 2, \dots, n\}$$

- $Y[i][j]$: course n(i) be putted in room m(j)

$$i \in \{1, 2, \dots, N\}, j \in \{1, 2, \dots, M\}, Y[i][j] \in \{0; 1\}$$

- m: number of periods that need for scheduling exam
- p: list of pairs conflicting courses

2. Constraint

- **Constraint 1: Pairs of conflicting courses may not be put in the same period**

$$X[i] \neq X[j], \forall (i, j) \in p$$

- **Constraint 2: An course room is assigned at most one course in a period**

$$\sum_{j=1}^{M} Y[i][j] = 1, \forall i = 1, \dots, N$$

```python
# Constraint 1: Pairs of conflicting courses may not be put in the same period
for pair in p:
    model.Add(x[pair[0]] != x[pair[1]])

# Constraint 2: An course room may be assigned at most one course in a period
for i in range(N):
    model.Add(sum(y[i]) == 1)
```

- **Constraint 3: Courses with same period cannot use the same room**

$$X[i] = X[j] \Rightarrow Y[i][k] + Y[j][k] \leq 1, \forall i, j \in \{1, \dots, N\}; k$$

```
# Constraint 3: Courses with same period may not share an course room
for j in range(M):
  for i1 in range(N - 1):
    for i2 in range(i1 + 1, N):
      b = model.NewBoolVar(f'b[{j}][{i1}][{i2}]')
      model.Add(y[i1][j] + y[i2][j] <= 1).OnlyEnforceIf(b)
      model.Add(x[i1] == x[i2]).OnlyEnforceIf(b)
      model.Add(x[i1] != x[i2]).OnlyEnforceIf(b.Not())
```

- **Constraint 4: The attendance of course n(i) must be smaller than capacity of room**

$$d[i] \leq \sum_{j=1}^{M} Y[i][j] \times c[j], \forall i = 1, \dots, N$$

```
# Constraint 4: A course n_i must be put into a room m_j with adequate capacity c(j)
for i in range(N):
  model.Add(sum([y[i][j] * c[j] for j in range(M)]) >= d[i])
```

3. Objective

$\Longrightarrow Minimize(m)$ **with** $m = max(X)$

```
# Define objective
cp_obj = model.NewIntVar(1, N, 'obj')
model.AddMaxEquality(cp_obj, x)
model.Minimize(cp_obj)

# Instantiate a CP solver
cp_solver = cp_model.CpSolver()
cp_solver.parameters.max_time_in_seconds = 40.0
```

- ***BACKTRACKING***

```
end := a very very large number
define function: dfs(u,slot):
  if u == N:
    end = min(slot, end)
    return
  if slot > end:
    return
  for each room:
    if room is free:
      for each course:
        if course can be putted and attendance <= capacity:
          put course to that room
          put course to that slot
          dfs(u+1,slot)
          free that room
          free that slot
  dfs(u,slot+1)
  return
```

```python
end = 100000000
conflict = [[] for _ in range(N)]

for k in p:
  u, v = k[0], k[1]
  conflict[u].append(v)
  conflict[v].append(u)

# assign period
period = [-1] * N

# room
room = []
for _ in range(N):
  room.append([-1] * M)

def isPlaceable(u, slot):
  if period[u] >= 0:
    return False
  for v in conflict[u]:
    if period[v] == slot:
      return False
  return True
```

```python
def dfs(u, slot):
  global end
  if u == N:
    end = min(end, slot)
    return
  if slot > end:
    return
  for j in range(M):
    if room[slot][j] == -1:
      for i in range(N):
        if isPlaceable(i, slot) and d[i] <= c[j]:
          period[i], room[slot][j] = slot, i
          dfs(u + 1, slot)
          period[i], room[slot][j] = -1, -1
  dfs(u, slot + 1)
  return

# Solve
start_time = time.process_time()
dfs(0, 0)
end_time = time.process_time()

# Solution
if end != 100000000:
  print(f'Objective value: {end + 1}')
else:
  print('No solution.')
print('-----------------')
print(f'Used time: {1000*(end_time - start_time)} milliseconds')
```

• *HEURISTIC 1*

```
sort list of (capacity,room) in ascending order of capacity
for each Course:
   for each Period:
      if Course test cannot be putted in any existing Period:
         add new Period
      if other course in Period conflicts with Course:
         consider the next Period
      else:
         for each sorted Room:
            if attendants <= capacity and at that Room and Period have no test:
               put Course to that Room and Period
               consider the next Course
```

```python
# List of (capacity, room) are sorted by capacity in ascending order
sorted_c = sorted([(c[i], i) for i in range(M)])

# Conflicts
conflicts = {} # conflicts[i] = list of courses that cannot be administered in the same period as course i+1
for pair in p:
    conflicts.setdefault(pair[0], []).append(pair[1])
    conflicts.setdefault(pair[1], []).append(pair[0])
```

```python
def greedy_2():
  result = [[-1] * M] # initiate with first period
                      # Result[i, k] = course exam administered in period i+1 and room k+1
  for exam in range(N): #sequentially assign a period and a room to each course
    nextCourse = False
    for period in range(len(result) + 1): #consider existing periods first
      if period == len(result):
        #if this exam cannot be held in any existing period, create a new period
        result.append([-1] * M) # new period with M rooms
      not_ThisPeriod = False
      if exam in conflicts:
        for otherCourse in result[period]:
          if otherCourse in conflicts[exam]:
            not_ThisPeriod = True
            break
        if not_ThisPeriod == True:
          continue
      for room  in range(M): #consider smaller rooms first to save bigger ones for other courses
        capacity = sorted_c[room][0]
        roomIndex = sorted_c[room][1]
        if result[period][roomIndex] == -1 and capacity >= d[exam]:
          result[period][roomIndex] = exam
          nextCourse = True
          break
      if nextCourse == True:
        break
  return len(result), result
```

• HEURISTIC 2

```
list_of_exam = sorted([(attendant, i)])
while list_of_exam not empty:
    allocate a new period for remaining exams
    for each Room:
        for exam in list_of_exam:
            if attendant <= capacity:
                if have no exam scheduled conflicts in this period:
                    put exam in this Room and this period
                    consider the next room
```

```python
conflicts = {} #conflicts[i] = list of exams that cannot be administered in the same period as exam i+1
for pair in p:
    conflicts.setdefault(pair[0], []).append(pair[1])
    conflicts.setdefault(pair[1], []).append(pair[0])

print('\nPeriod', 'Room', 'Exam', sep='\t')

sortedExams = sorted([(d[i], i) for i in range(N)], reverse=True) #sort exams in ascending order of capacity

schedule = [] #schedule[i, k] = exam administered in period i+1 and hall k+1
period = 0
startTime = time.process_time()
while sortedExams: #sequentially fill each period with as many exams as possible until all exams have been scheduled
    schedule.append([None] * M)
    for room in range(M):
        for exam in sortedExams: #consider more popular exams first
            if exam[0] <= c[room]: #if a hall has adequate capacity
                #check if any exam already scheduled in this period has common candidates with the one being considered
                noConflict = True
                if exam[1] in conflicts:
                    for scheduledExam in schedule[period]:
                        if scheduledExam in conflicts[exam[1]]:
                            noConflict = False
                            break
                if noConflict: #schedule exam in period and hall and remove from list of exams to schedule
                    schedule[period][room] = exam[1]
                    sortedExams.remove(exam)
                    break
    period += 1
```

• HEURISTIC 3

```
sort list of rooms in ascending order of capacity
for each Course:
   for each Period:
      for each Room:
         if capacity >= attendants and no exam scheduled at Period and Room yet:
            if no other course  in period conflicts with Course:
               put Course to Period and Room
               consider the next Course
      if Course exam cannot be held in any Period:
         add new Period
```

```python
conflicts = {} #conflicts[i] = list of exams that cannot be administered in the same period as exam i+1
for pairss in p:
    conflicts.setdefault(pairss[0], []).append(pairss[1])
    conflicts.setdefault(pairss[1], []).append(pairss[0])

sortedRooms = sorted([(c[i], i) for i in range(M)]) #sort rooms in ascending order of capacity
result = [[None] * M] #result[i, k] = exam administered in period i+1 and room k+1
print('\nExam', 'Period', 'Room', sep='\t')
for exam in range(N): #sequentially assign a period and a room to each exam
    stop = False
    for period in range(len(result) + 1): #consider existing periods first
        for room in range(M): #consider smaller rooms first to save bigger ones for other exams
            capacity = sortedRooms[room][0]
            roomIndex = sortedRooms[room][1]
            if capacity >= d[exam] and result[period][roomIndex] == None:
                noConflict = True
                if exam in conflicts:
                    for otherExam in result[period]:
                        if otherExam in conflicts[exam]:
                            noConflict = False
                            break
                if noConflict:
                    result[period][roomIndex] = exam
                    print(exam + 1, period + 1, room + 1, sep='\t') #print schedule by exam
                    stop = True
                    break
        if stop:
            break
        if period == len(result) - 1:
            #if this exam cannot be held in any existing period, set up a new period
            result.append([None] * M)
```

3. ANALYSIS AND CONCLUSION

OUR RESULTS

Test 1: With some small datasets (N/M < 4):

- MIP, CP work quite well. Meanwhile, BT takes much time to find the result.

N	MIP value	MIP time	CP value	CP time	BT value	BT time
6	3	32,01	3	34,1	3	109,38
9	3	54,02	3	88,13	3	4273000
12	5	57,44	5	170,44	-	INF
16	4	1039,88	4	30147	-	INF
20	7	203,01	7	30312	-	INF

*time unit: millisecond
value unit: period

Test 1: **With some small datasets (N/M < 4):**

- MIP, CP work quite well. Meanwhile, BT takes much time to find the result.

CP, MIP

Backtracking

OUR RESULTS

Test 1: **With some small datasets (N/M < 4):**

- **Heuristic 1, Heuristic 2 and Heuristic 3 outperform MIP, CP, BT. They likely give the same results with an small amount of running time**

N	Heu 1 value	Heu 1 time	Heu 2 value	Heu 2 time	Heu 3 value	Heu 3 time
6	3	~0	3	~0	3	~0
9	3	~0	3	~0	3	~0
12	5	~0	5	~0	4	~0
16	4	0,999	4	0,997	4	0,81
20	7	1,004	7	1,002	7	0,9

***time unit: millisecond**
value unit: period

OUR RESULTS

Test 2: With Larger datasets (N/M > 4):

- **MIP, CP become worse and worse.**

N	K	MIP value	MIP time	CP value	CP time
50	20	50	39187	6	31291
100	45	53	30433	-	Timed out
150	60	-	Timed out	-	Timed out
200	100	-	Timed out	-	Timed out
200	200	-	Timed out	-	Timed out

***time unit: millisecond**
value unit: period
time limit = 30000 milisecond

OUR RESULTS

Test 2: **With Larger datasets (N/M > 4): Heuristic algorithms still run fast with OPTIMAL solutions with nearly the same running time.**

N	K	Heu 1 Value	Heu 1 Time	Heu 2 Value	Heu 2 Time	Heu 3 Value	Heu 3 Time
50	20	6	0,997	6	0,1	7	0,996
100	45	8	1,001	7	0,969	10	1,01
150	60	8	4,117	9	0,01	8	2,00
200	100	9	6,228	9	1,757	9	4,98
200	200	10	10,09	10	4,038	10	2,99

*time unit: millisecond
value unit: period

OUR RESULTS

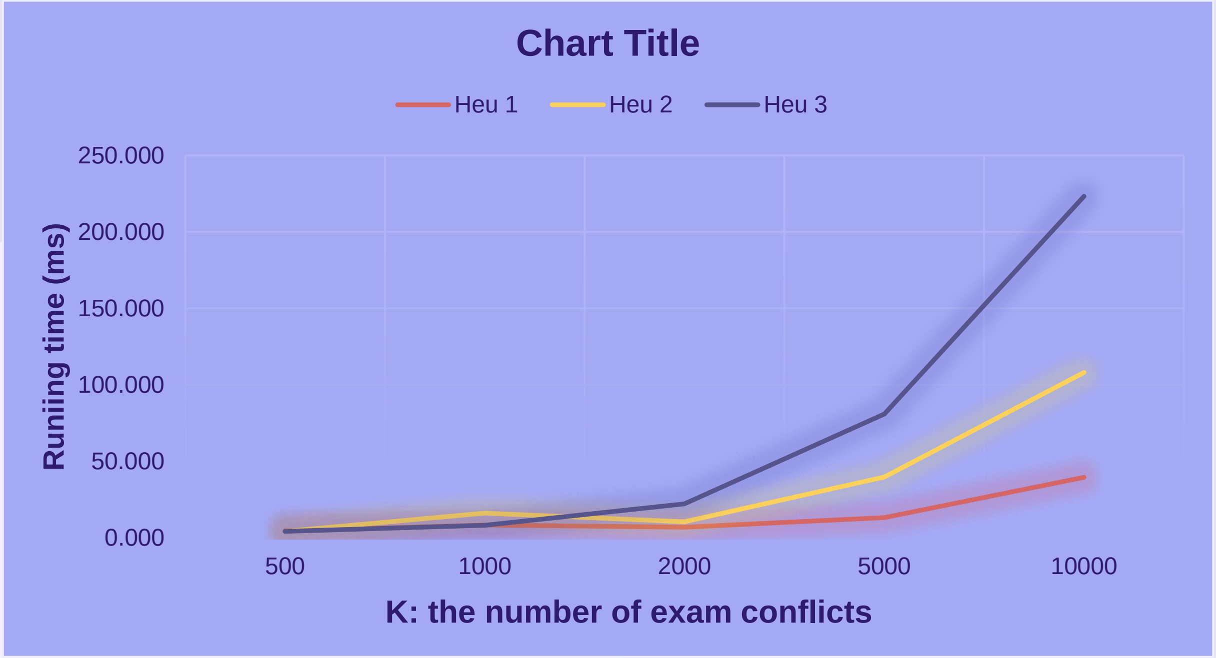
Test 3: **Keep N, M at 200, 25; Increase the value of K (number of exams' conflict): There are some small differences between heuristic algorithms**

N	K	Heu 1 Value	Heu 1 Time	Heu 2 Value	Heu 2 Time	Heu 3 Value	Heu 3 Time
200	500	9	4,516	10	4,154	9	4.02
200	1000	9	8,238	10	15,999	10	7,98
200	2000	11	6,792	11	10,425	11	21,94
200	5000	19	12,997	20	39,699	20	80,8
200	10000	36	39,476	37	108,015	39	223,4

*time unit: millisecond
value unit: period

OUR RESULTS

Test 3: **Keep N, M at 200, 25; Increase the value of K (number of exams' conflict): There are considerable gaps between running time of algorithms**



AWESOME WORDS