4 Part A: Constraint Satisfaction Problems (CSPs) (50%)

In this section, you will be asked to find a seat for 18 passengers on a plane based on a number of constraints. The plane layout for this part is depicted in Figure 1.

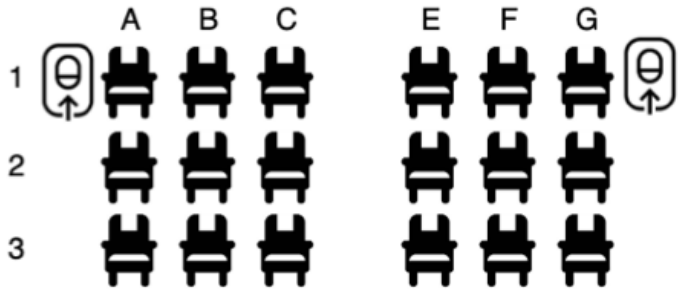


Figure 1: Plane layout for Part A

1. A flight is fully booked from Edinburgh to Kirkwall with 18 seats as shown above. The passengers are as follows:

   - 1 family (consisting of 1 young couple and 3 children)
   - 3 young couples
   - 1 elderly couple
   - 1 wheelchair user
   - 2 elderly individuals
   - 2 young individuals

   There are also a number of rules on how individuals may be seated based on their age and requirements.

   [R1] Children must be sat immediately next to at least one parent. They cannot be separated by another passenger or the aisle.

   [R2] There are emergency doors on the first row; these seats cannot be occupied by children, elderly passengers, or passengers with physical disabilities.

   [R3] Couples without children must be sat immediately next to each other.

   [R4] Families must be assigned to the same row.

   [R5] Passengers requiring a wheelchair must be assigned to an aisle seat.

   [R6] The elderly individual passengers have both requested window seats.

   [R7] The elderly couple has requested to be sat on the third row.

   [R8] One member of the elderly couple has requested a window seat.

(a) (5 points) Write down the variables and the domain (before applying any of the constraints) to translate this into a CSP problem.

Qa.

Variable = { FW, FH, FC_1, FC_2, FC_3, YW_1, YH_1, YW_2, YH_2, YW_3, YH_3, EW, EH, WU, EI_1, EI_2, YI_1, YI_2 }

where FW stand for family wife, FH stand for family husband, FCi stand for ith family children, YWi stand for ith couple wife, YHi stand for ith couple husband, EW and EH stand for elderly wife and husband, WU stand for wheelchair user, EIi stand for ith elderly individual, and YIi stand for ith young individual.

Domain D_i = {1A, 1B, 1C, 1E, 1F, 1G, 2A, 2B, 2C, 2E, 2F, 2G, 3A, 3B, 3C, 3E, 3F, 3G}

(b) (5 points) Of the above rules which ones can be represented as unary constraints? For each variable, write down the domain after applying the constraints.

Qb.

R_2, R_5, R_6, R_7 are unary constraints.

FW = {1A, 1B, 1C, 1E, 1F, 1G, 2A, ~~2B~~, 2C, 2E, 2F, 2G, 3A, 3B, 3C, 3E, 3F, 3G}

FH = {1A, 1B, 1C, 1E, 1F, 1G, 2A, ~~2B~~, 2C, 2E, 2F, 2G, 3A, 3B, 3C, 3E, 3F, 3G}

FC_1 = { 2A, ~~2B~~, 2C, 2E, 2F, 2G, 3A, 3B, 3C, 3E, 3F, 3G}

FC_2 = { 2A, ~~2B~~, 2C, 2E, 2F, 2G, 3A, 3B, 3C, 3E, 3F, 3G}

FC_3 = { 2A, ~~2B~~, 2C, 2E, 2F, 2G, 3A, 3B, 3C, 3E, 3F, 3G}

YW_1 = {1A, 1B, 1C, 1E, 1F, 1G, 2A, ~~2B~~, 2C, 2E, 2F, 2G, 3A, 3B, 3C, 3E, 3F, 3G}

YH_1 = {1A, 1B, 1C, 1E, 1F, 1G, 2A, ~~2B~~, 2C, 2E, 2F, 2G, 3A, 3B, 3C, 3E, 3F, 3G}

YW_2 = {1A, 1B, 1C, 1E, 1F, 1G, 2A, ~~2B~~, 2C, 2E, 2F, 2G, 3A, 3B, 3C, 3E, 3F, 3G}

YH_2 = {1A, 1B, 1C, 1E, 1F, 1G, 2A, ~~2B~~, 2C, 2E, 2F, 2G, 3A, 3B, 3C, 3E, 3F, 3G}

YW_3 = {1A, 1B, 1C, 1E, 1F, 1G, 2A, ~~2B~~, 2C, 2E, 2F, 2G, 3A, 3B, 3C, 3E, 3F, 3G}

$YH_3 = \{1A, 1B, 1C, 1E, 1F, 1G, 2A, 2B, 2C, 2E, 2F, 2G, 3A, 3B, 3C, 3E, 3F, 3G\}$

$EW = \{3A, 3B, 3C, 3E, 3F, 3G\}$

$EH = \{3A, 3B, 3C, 3E, 3F, 3G\}$

$WU = \{2C, 2E, 3C, 3E\}$

$EI_1 = \{2A, 2G, 3A, 3G\}$

$EI_2 = \{2A, 2G, 3A, 3G\}$

$YI_1 = \{1A, 1B, 1C, 1E, 1F, 1G, 2A, 2B, 2C, 2E, 2F, 2G, 3A, 3B, 3C, 3E, 3F, 3G\}$

$YI_2 = \{1A, 1B, 1C, 1E, 1F, 1G, 2A, 2B, 2C, 2E, 2F, 2G, 3A, 3B, 3C, 3E, 3F, 3G\}$

(c) (6 points) To define the constraints, write down pseudocode for functions that take a pair of seats and return a boolean value for the following relations **same_row** and **sat_adjacent**.

Qc.

pseudocode:

```
function same_row(seat1, seat2) return a boolean value
    if seat1's row is equal to seat2's row
        return True
    else
        return False.

function sat_adjacent(seat1, seat2) return a boolean value
    if same_row(seat1, seat2):
        if seat1's column is next to seat2's column:
            if there is no aisle between seat1 and seat2
                return True
    else:
        return False
```

(d) (8 points) The constraint associated with one of the above rules contains a ternary relation. Identify this rule and define the ternary relation using the relations defined above. By introducing a new variable convert the ternary relation into a set of binary relations.

Qd. This rule is R_1

Ternary relation:
for $i \in \{1,2,3\}$
$<(\bar{FC_i}, \bar{FW}, \bar{FH}), \{(c_i, w, h) \mid sat\_adjacent(c_i, w) \vee sat\_adjacent(c_i, h), \text{for } c_i \in FC_i, w \in FW, h \in FH\}$

We can now introduce a new variable FP, which is family parent. Its domain is $\{(f_w, f_h) \mid f_w \in FW, f_h \in FH\}$

Thus, we can convert this ternary relation into a set of binary relation.
For $i \in \{1,2,3\}$
$R_{FC_i,FP} = <(FC_i, \bar{FP}), \{(c_i, p) \mid sat\_adjacent(c_i, p[0]) \text{ or } sat\_adjacent(c_i, p[1]), \text{for } c_i \in FC_i, p \in FP\}>$
$R_{FW,FP} = <(\bar{FW}, FP), \{(w, p) \mid w = p[0] \text{ for } w \in FW, p \in FP\}>$
$R_{FH,FP} = <(\bar{FH}, FP), \{(h, p) \mid h = p[1], \text{for } h \in FH, p \in FP\}>$
The set of binary relation is $\{R_{FC_i,FP}, R_{FW,FP}, R_{FH,FP}\}$

(e) (6 points) Write down the list of binary constraints and draw the corresponding constraint graph.

Qe.

① Know from Rule 3, Couple without children must seat next to eachother, so

$R_{YW_i,YH_i} = \langle (YW_i, YH_i), \{(w_i, h_i) \mid sat\_adjacent(w_i, h_i), \text{for } w_i \in YW_i, h_i \in YH_i\} \rangle$, where $i \in \{1, 2, 3\}$

$R_{EW,EH} = \langle (EW, EH), \{(w, h) \mid sat\_adjacent(w, h), \text{for } w \in EW, h \in EH\} \rangle$

② Here's the Rule 1 written in Qd, children must seat next to one of the parents

For $i \in \{1, 2, 3\}$

$R_{FC_i,FP} = \langle (FC_i, FP), \{(c_i, p) \mid sat\_adjacent(c_i, p[0]) \text{ or } sat\_adjacent(c_i, p[1]), \text{for } c_i \in FC_i, p \in FP\} \rangle$

$R_{FW,FP} = \langle (FW, FP), \{(w, p) \mid w = p[0] \text{ for } w \in FW, p \in FP\} \rangle$

$R_{FH,FP} = \langle (FH, FP), \{(h, p) \mid h = p[1], \text{for } h \in FH, p \in FP\} \rangle$

③ It is absolutely correct that every one must sit in different seats.

$Alldiff = \langle \{(Var_1, Var_2) \mid Var_1 \in Variable, Var_2 \in Variable, Var_1 \neq Var_2\}, \{(a, b) \mid a \in Var_1, b \in Var_2, a \neq b\} \rangle$

④ Rule 4 said family must sit in same row. showing each pair of family member must be in same row

$SR_{family} = \langle \{(FM_1, FM_2) \mid FM_1, FM_2 \in FC_i \cup FH \cup FW, FM_1 \neq FM_2\}, \{(fm_1, fm_2) \mid same\_row(fm_1, fm_2), fm_1 \in FM_1, fm_2 \in FM_2\} \rangle$

⑤ We know from rule 8, one of the elderly member request a window seat, so either elderly wife or elderly husband will be in the Window Seat Set.

let WindowSeat = {1A, 1G, 2A, 2G, 3A, 3G}

$R_2EW, EH = \langle (EW, EH), \{(w, h) \mid [(w \in WindowSeat) \vee (h \in WindowSeat)] \wedge \neg[(w \in WindowSeat) \wedge (h \in WindowSeat)]\}\rangle$

⑥ Here in order to let the tree search be more efficient, we introduce a simple forward checking in FP.

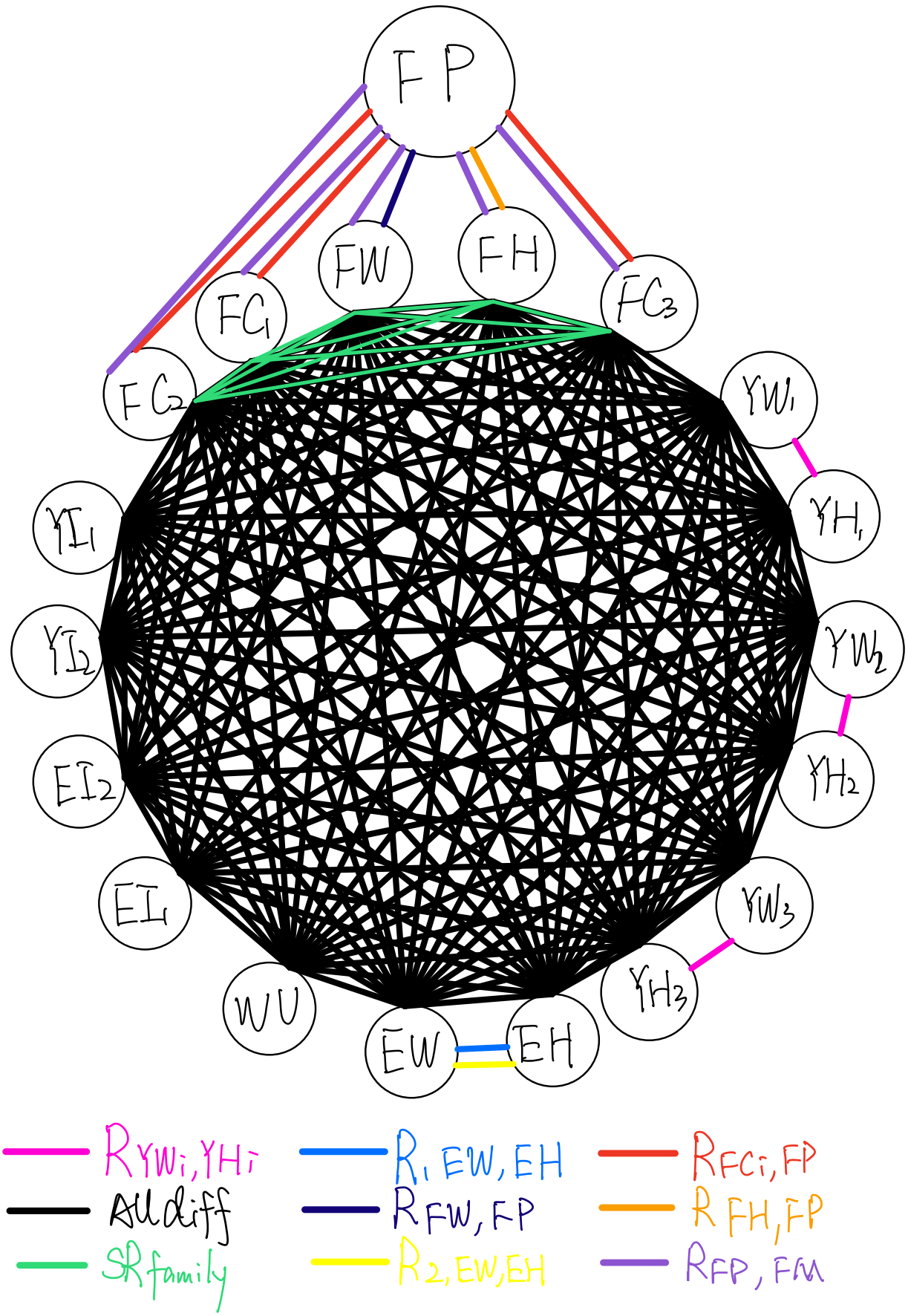
For every element in FP = (a, b)

we know that same_row(a, X) and same_row(b, X)

for $X \in \{FH, FW, FC_1, FC_2, FC_3\}$

$R_{FP,FM} = \{(FP, FM) \mid \{[(a,b), c] \mid (a,b) \in FP, c \in \{FH, FW, FC_1, FC_2, FC_3\}$ same_row(a, c) and same_row(b, c)$\}$

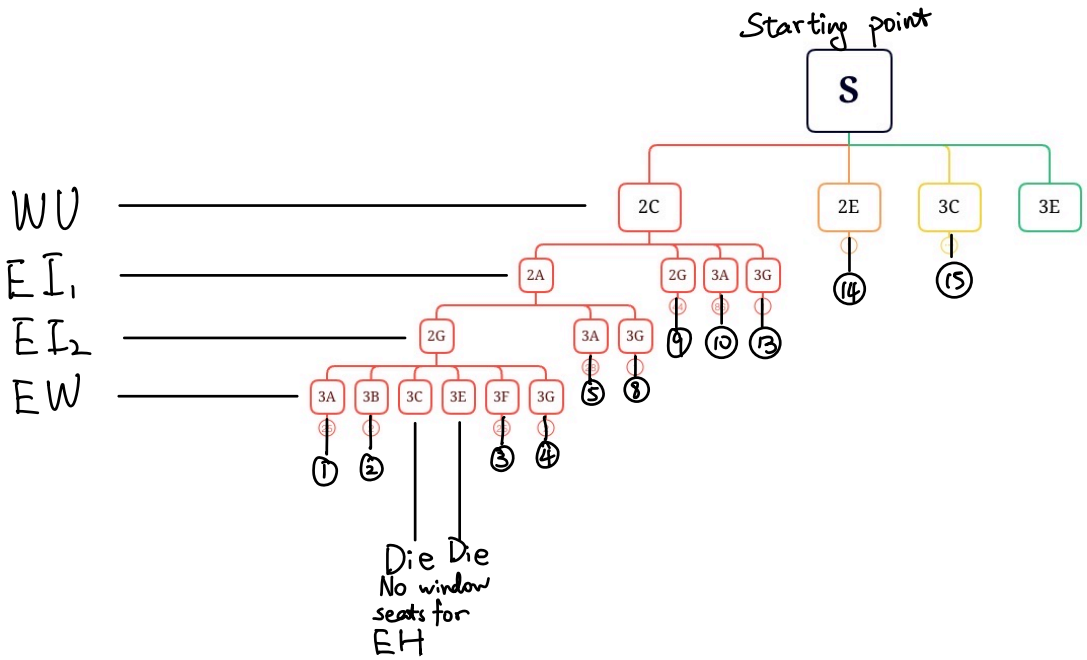
Thus, it can quickly eliminate the inproper family parent pairs.

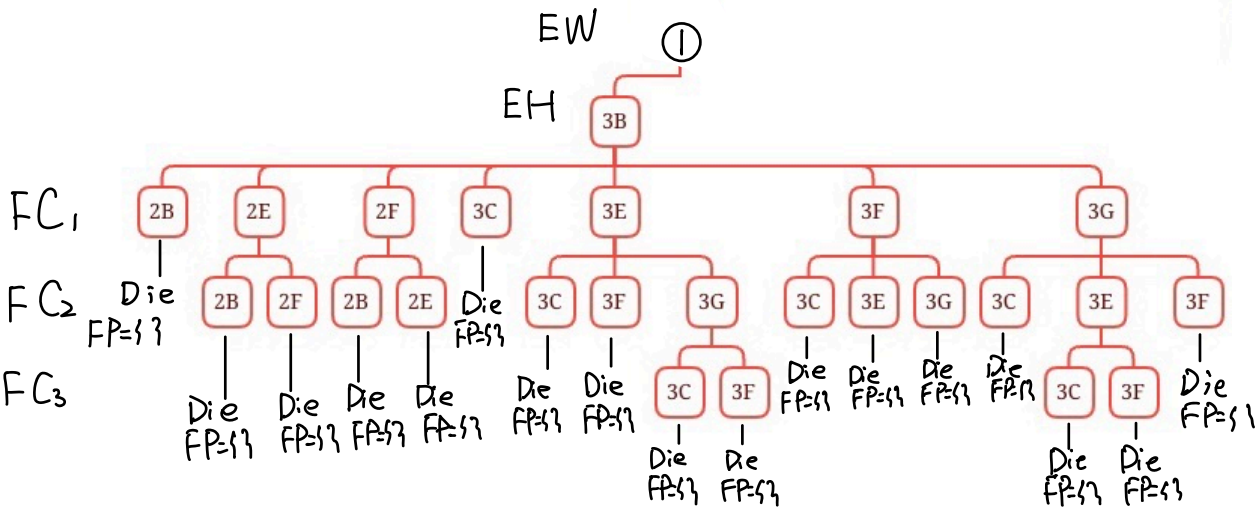

FP
FW
FH
FC1
FC3
FC2
YW1
YH1
YI1
YI2
YW2
EI2
YH2
EI1
YW3
YH3
WU
EW
EH
RYWi,YHi
Alldiff
SRfamily
R1,EW,EH
RFW,FP
R2,EW,EH
RFCi,FP
RFH,FP
RFP,FM

(f) (12 points) Use backtracking search with the Minimum-Remaining Values heuristic to find a solution. The domain values should be ordered row-wise and then aisle-wise. The search tree should be presented as in Figure 2. You should also show any additional working outs.

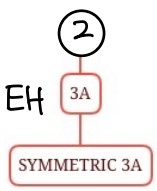
Since we use minimum-Remaining Values heuristic, if we have the same domain, then we use this order $FC_1, FC_2, FC_3, FW, FH_1, YW_1, YH_2, YW_2, YH_2, YW_3, YH_3, WU, EW, EH, EI_1, EI_2, YI_1, YI_2, FP$

Here FP must contain one constraint that $\langle (FP) | \{(a,b) | (a,b) \in FP, a \neq b\} \rangle$

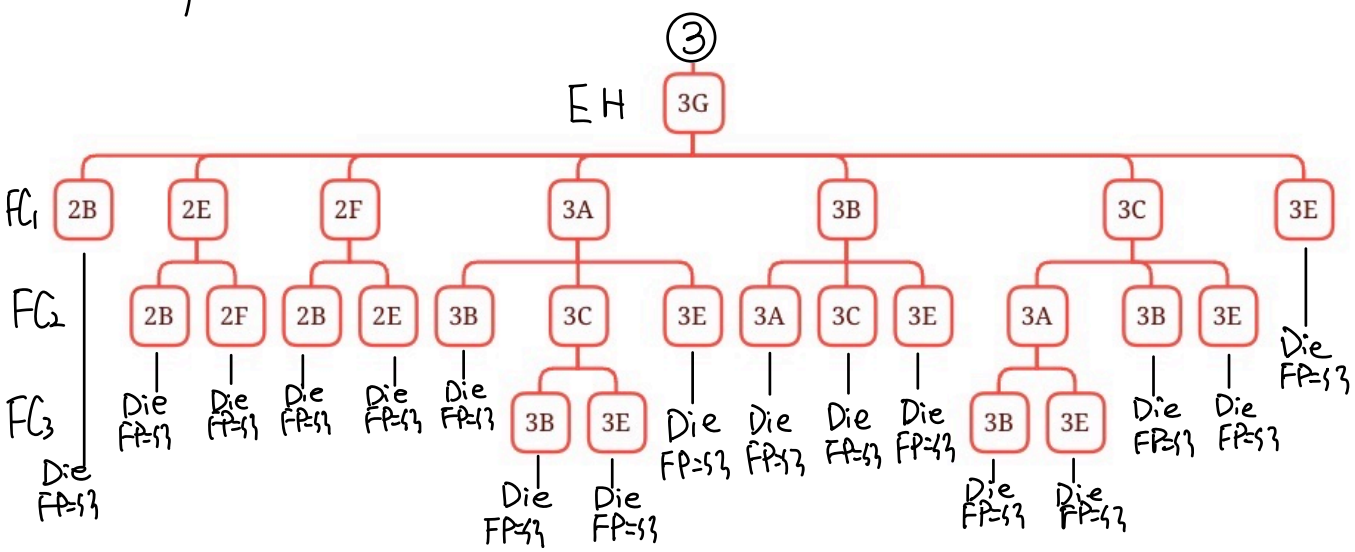


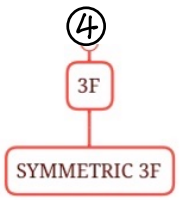


Here since there is no empty seats next to children, family parents will be an empty set, so the path die.

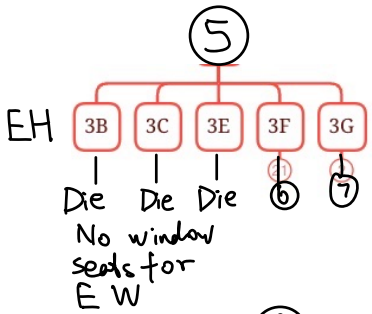


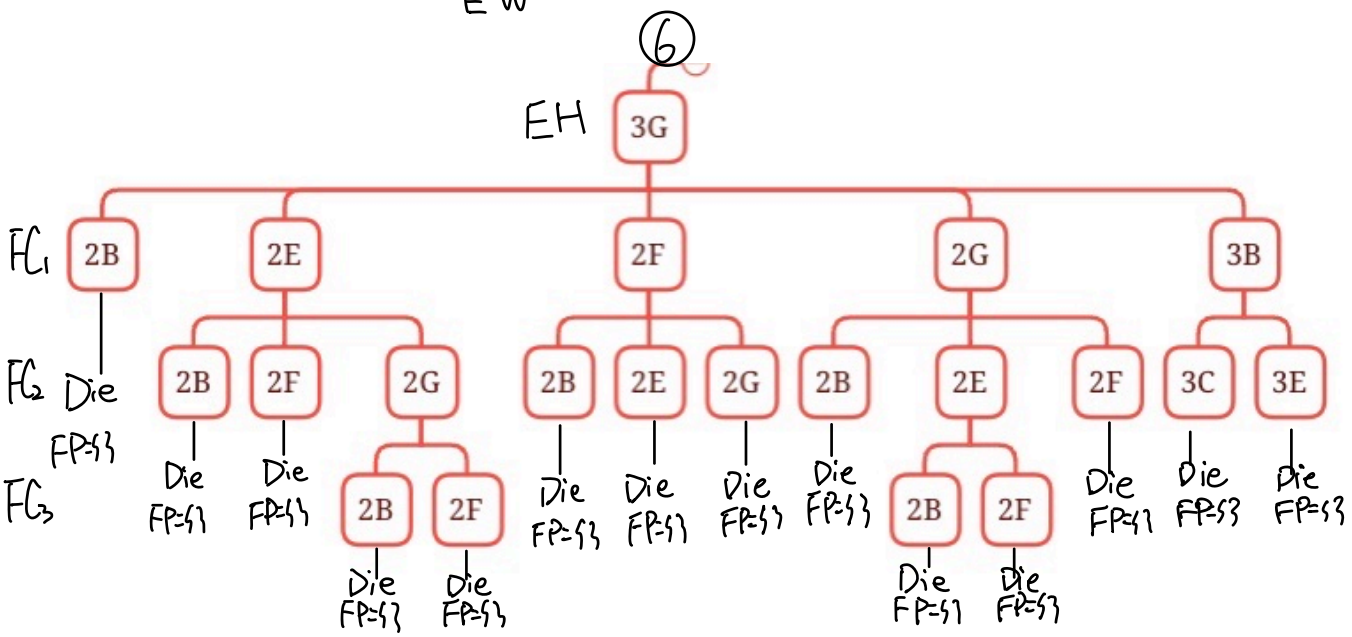
Here EW=3A EH=3B is symmetric with EW=3B EH=3A
Thus they die for same reason.

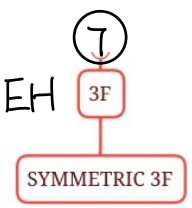




Here node ④ die for the same reason with node ③

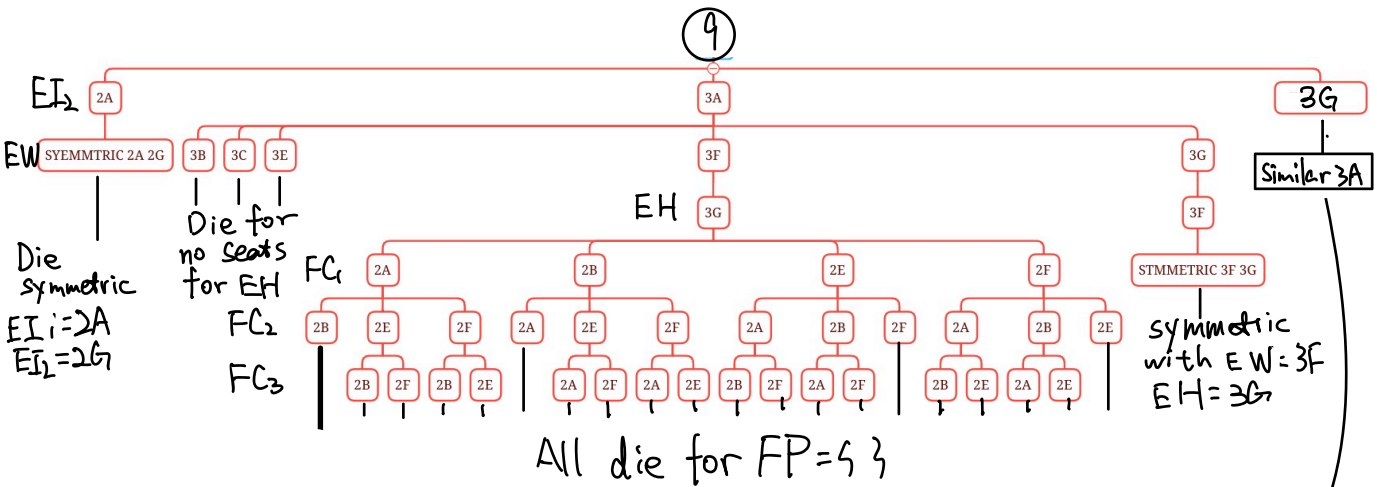




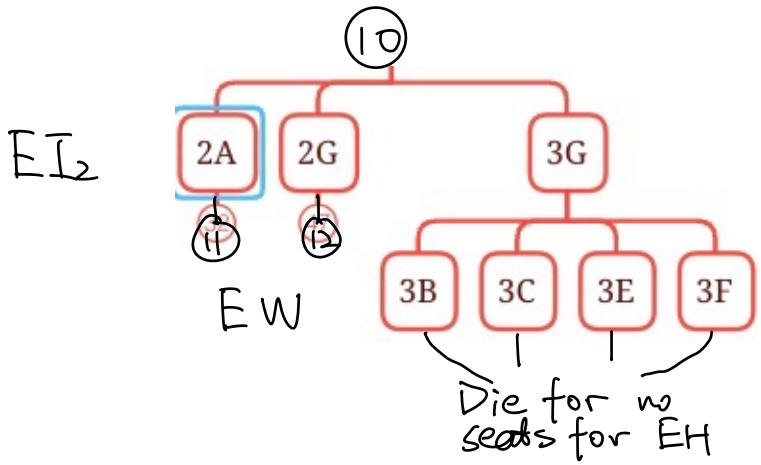


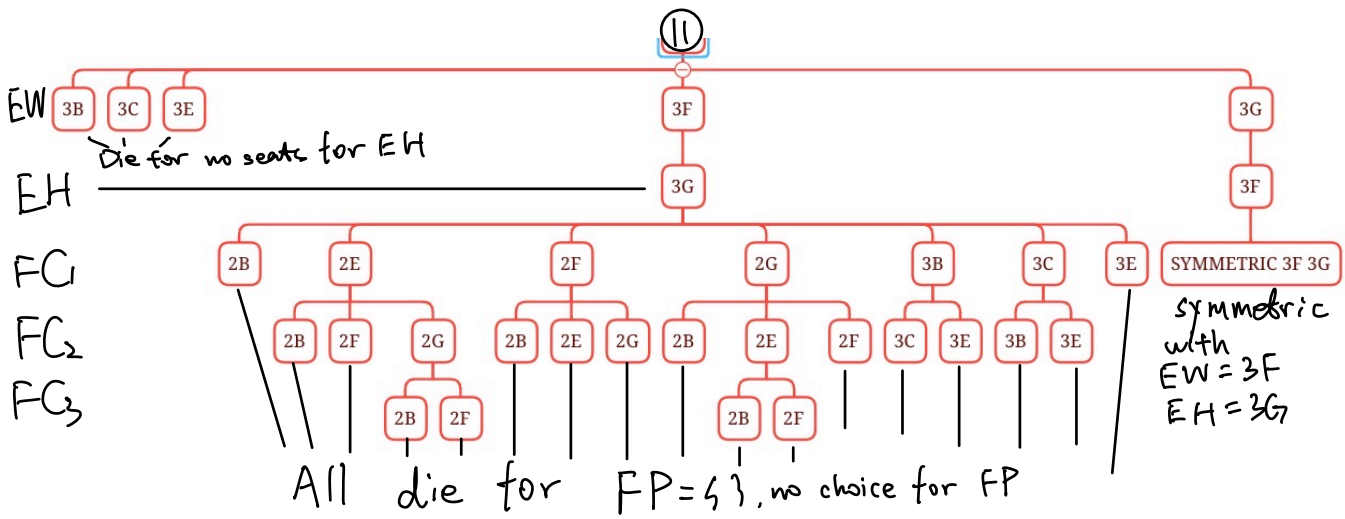
Here it is symmetric with EW=3F EH=3G

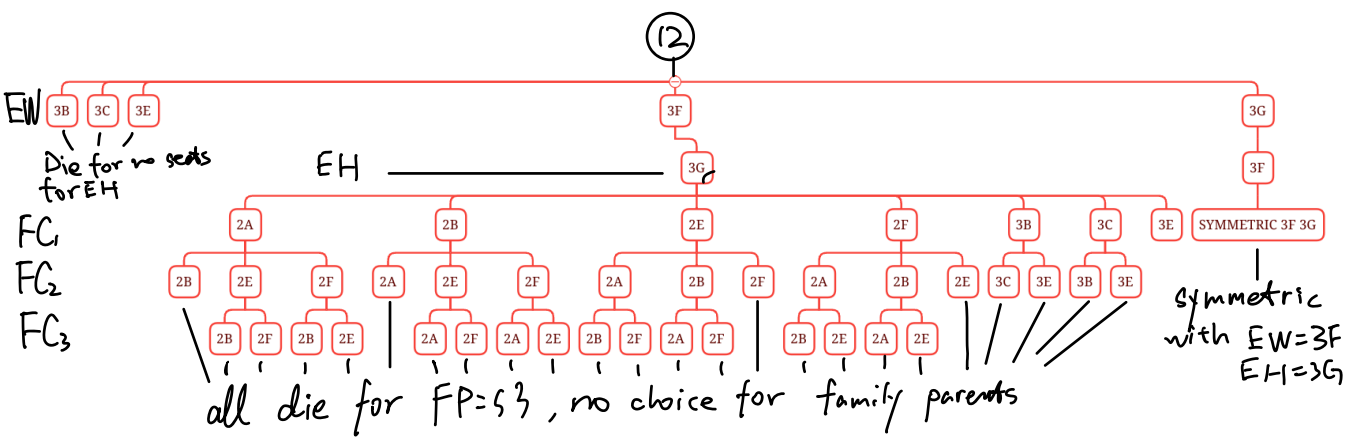
The reason for why it die similar with 3A is that In third row 3A and 3G are symmetrical, and the domain of elderly individuals, elderly couple and family children are also symmetrical in third row. Thus, since the selected seats and following variables are both symmetric, they will die for the similar reason..



Here we need to explain why 3G is similar to 3A It base with same reason which we dissussed before. Here 3A is symmetric with 3G in third row and the following variable's elderly couple and family chitdren are also symmetric in third row. Thus, they will die for the same reason with picked $EI_2 = 3A$


10
EI2
2A
2G
3G
11
12
EW
3B
3C
3E
3F
Die for no seats for EH


11
EW
3B
3C
3E
3F
3G
Die for no seats for EH
EH
3G
3F
FC1
FC2
FC3
2B
2E
2F
2G
3B
3C
3E
SYMMETRIC 3F 3G
symmetric with EW=3F EH=3G
All die for FP=53, no choice for FP


12
EW
3B
3C
3E
3F
3G
Die for no seats for EH
EH
3G
3F
FC1
FC2
FC3
2A
2B
2E
2F
3B
3C
3E
SYMMETRIC 3F 3G
symmetric with EW=3F EH=3G
all die for FP=53, no choice for family parents

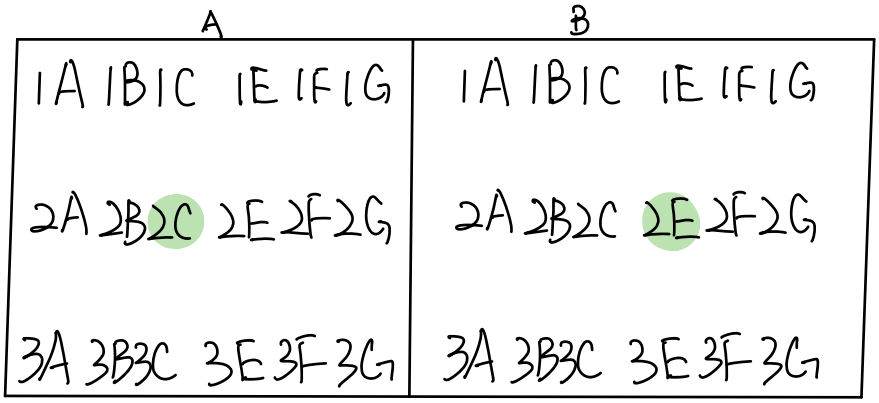
13

SIMILAR 3A

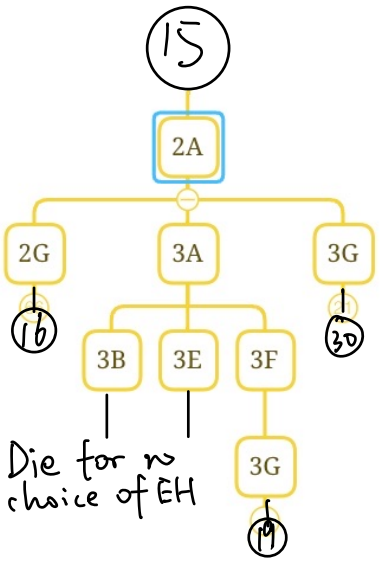
The reason for why choosing $EI_1 = 3G$ is similar to $EI_2 = 3A$ is that elderly individual domain is symmetric in these two case and the variables elderly couple and family children's domain are also symmetric. Thus, the reason for death is similar in two situation.

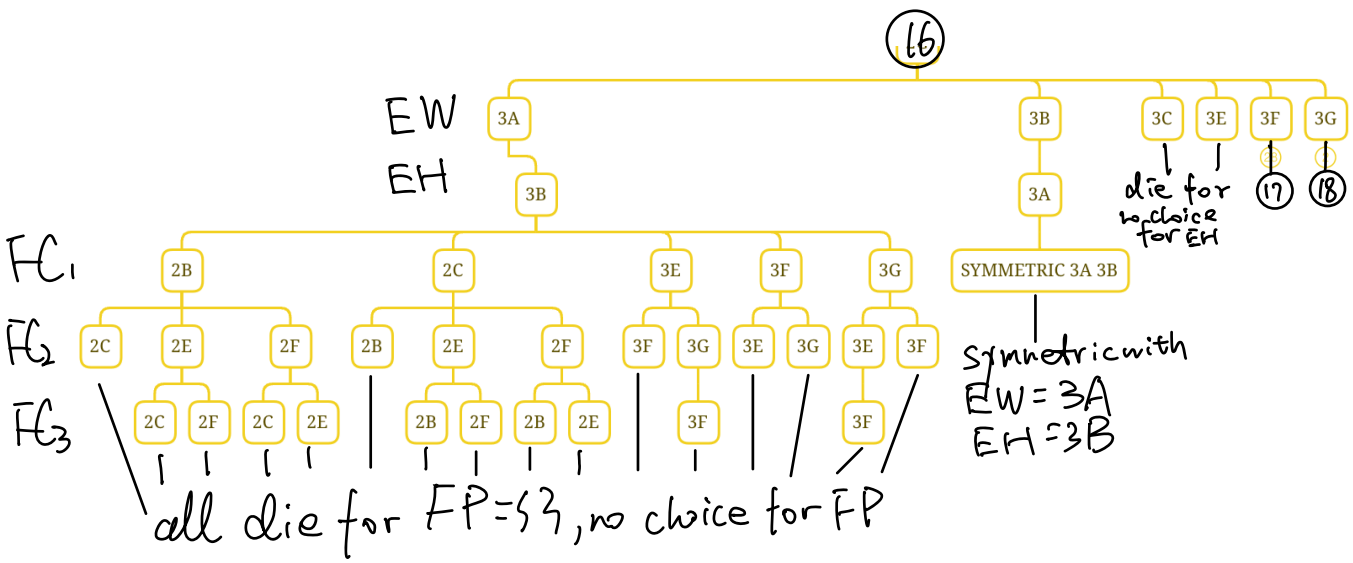
14

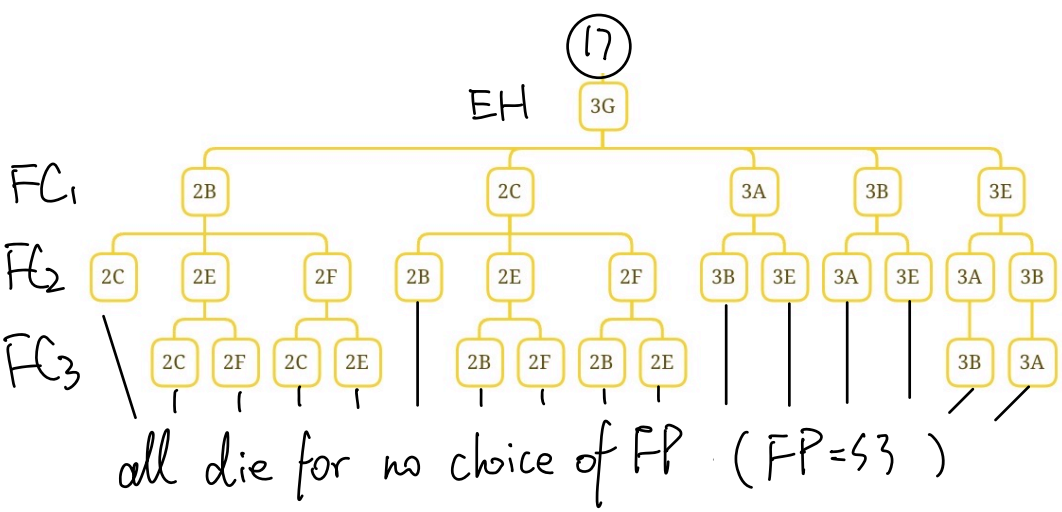
SIMILAR 2C



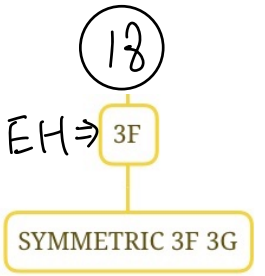
Here if we reflect picture B, we can get picture A. Thus, it is clearly true that these two situation are symmetric. What's more, the following variables are also symmetric. Here we have EI_1 and EI_2 and there domain are symmetric about the aisle. EW, EH and family children are also symmetric about aisle. Therefore, these two situation are symmetric, so they die for the same reason.

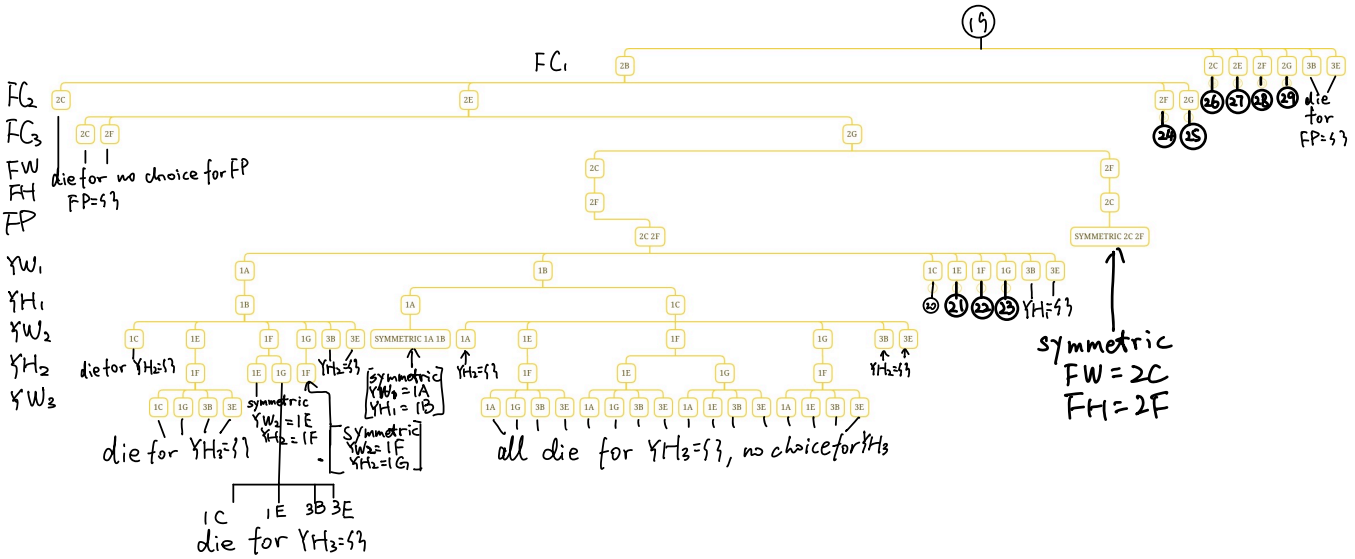

15
2A
2G
3A
3G
16
3B
3E
3F
20
Die for no choice of EH
3G
19


16
EW
EH
3A
3B
3B
3A
3C
3E
3F
3G
die for no choice for EH
17
18
FC1
FC2
FC3
2B
2C
3E
3F
3G
SYMMETRIC 3A 3B
symmetric with EW=3A EH=3B
all die for FP=53, no choice for FP

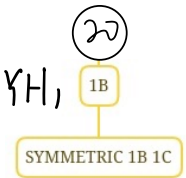

17
EH
3G
FC1
FC2
FC3
2B
2C
3A
3B
3E
all die for no choice of FP (FP=53)

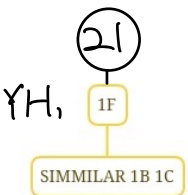
symmetric of
EW = 3F
EH = 3G



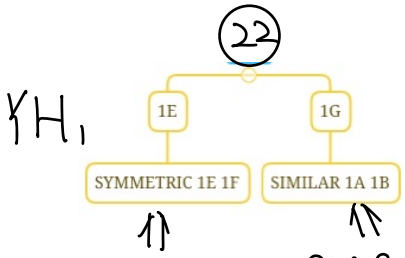


symmetric
YW1 = 1B
YH1 = 1C





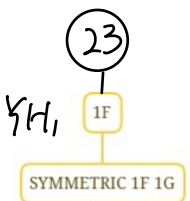
Here YW1 = 1E YH2 = 1F is similar to YW1 = 1B YH1 = 1C
In the plane if we reflect the plane about aisle, we know that these two situation are the same. Their children node should die for the similar reason, since YHi and YWi are also symmetric about aisle.



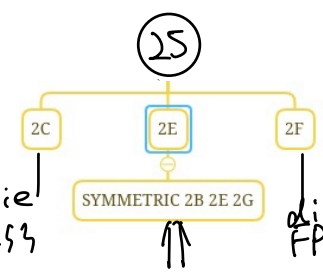
symmetric
$YW_1 = 1E$
$YH_1 = 1F$

same as the similar situation before.
Here $YW_1 = 1A$ $YH_1 = 1B$ is symmetric with $YW_1 = 1F$ $YH_1 = 1G$ about aisle.

symmetric
$YW_1 = 1F$ $YH_1 = 1G$



die for FP=53

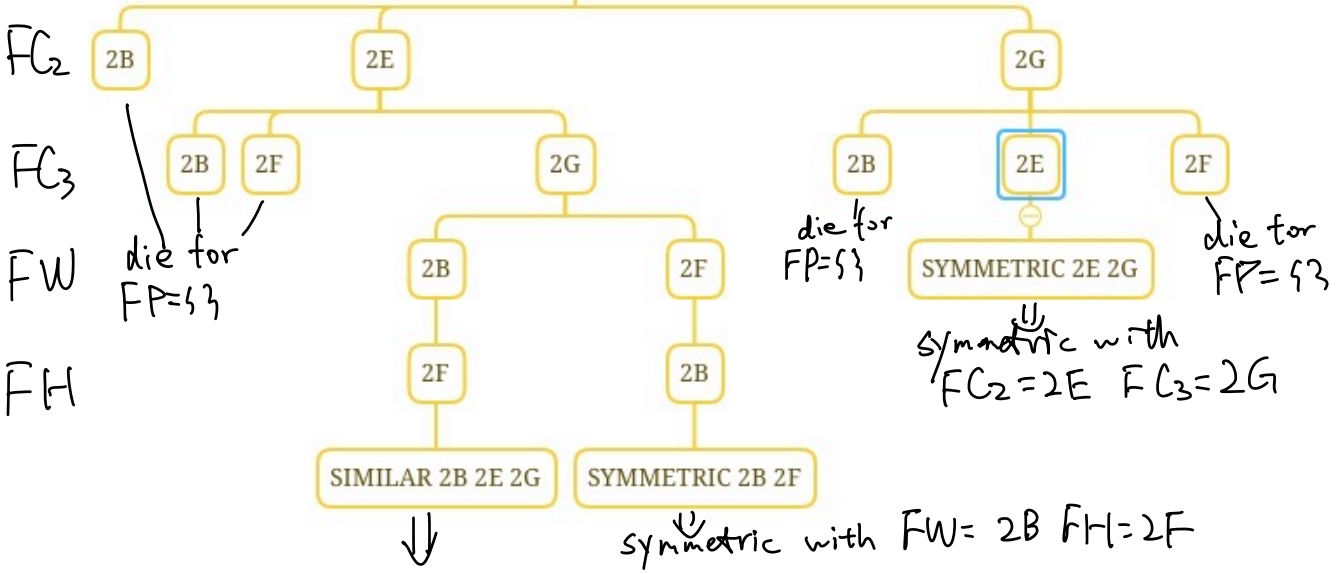


die FP=53

die FP=43

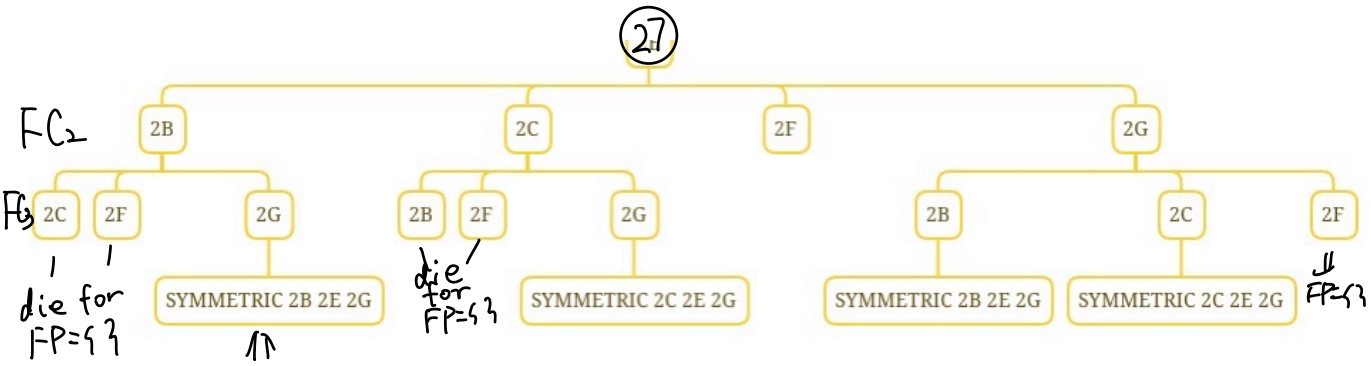
symmetric with $FC_1 = 2B$ $FC_2 = 2E$ $FC_3 = 2G$

(26)



Here why we say they are similar, because, in these situation, family members all sit in the row two from 2B to 2G. Thus, they can be seem as similar and remaining varibles' domain keep the same.

(27)



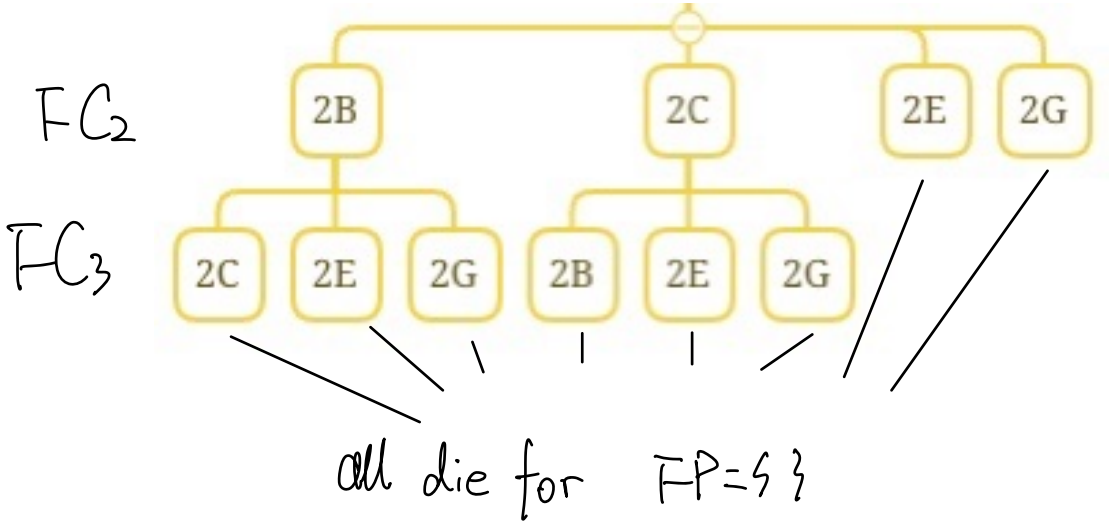
symmetric with
$FC_1 = 2B$
$FC_2 = 2E$
$FC_3 = 2G$

symmetric with
$FC_1 = 2C$
$FC_2 = 2E$
$FC_3 = 2G$

symmetric with
$FC_1 = 2B$
$FC_2 = 2E$
$FC_3 = 2G$

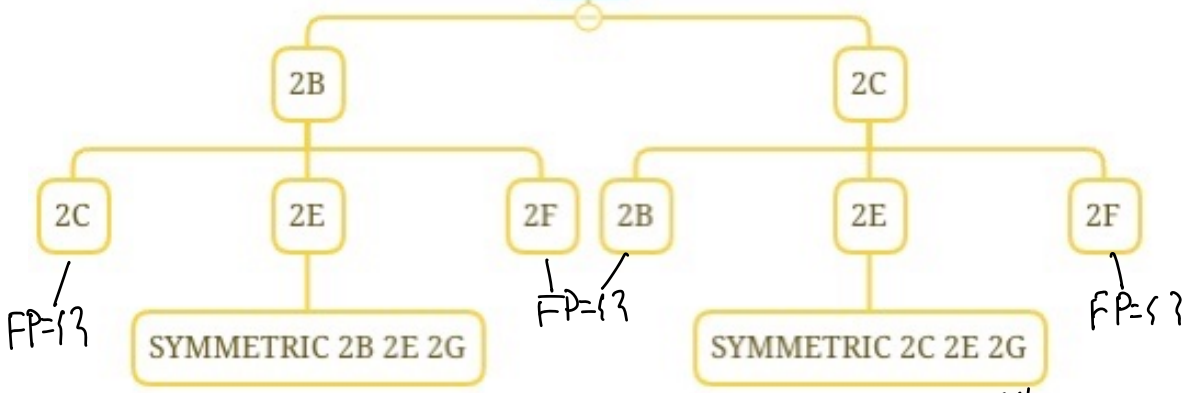
symmetric with
$FC_1 = 2C$
$FC_2 = 2E$
$FC_3 = 2G$

(28)



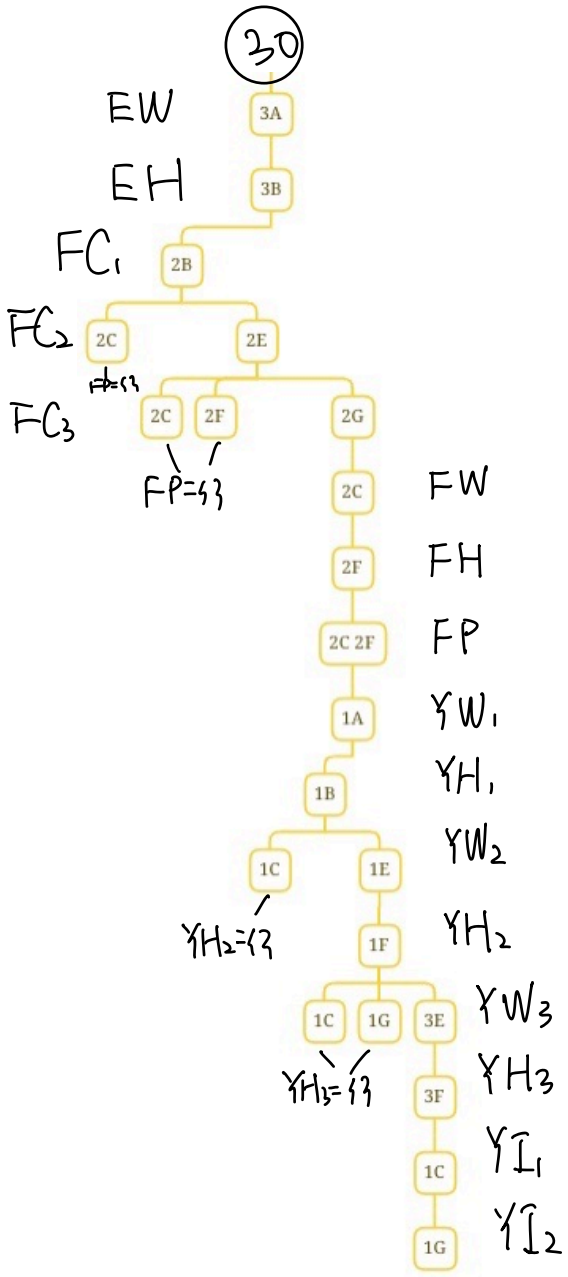
all die for FP=5

(29)



symmetric with
$FC_1 = 2B$
$FC_2 = 2E$
$FC_3 = 2G$

symmetric with
$FC_1 = 2C$
$FC_2 = 2E$
$FC_3 = 2G$



Here we find a solution then end.

$WU=3C$ $EI_1=2A$ $EI_2=3G$ $EW=3A$ $EH=3B$
$FC_1=2B$ $FC_2=2E$ $FC_3=2G$ $FW=2C$ $FH=2F$
$FP=(2C,2F)$ $FW_1=1A$ $FH_1=1B$ $FW_2=1E$
$FH_2=1F$ $FW_3=3E$ $FH_3=3F$ $YI_1=1C$
$YI_2=1G$

(g) (8 points) Discuss the pros and cons of using the Minimum-Remaining Values heuristic compared with the Degree heuristic combined with Lease Constraining Values for this problem. Which would you choose for future applications to a similar problem?

Q g. Pros of both algorithm

① Both algorithm can prune some unnecessary branch. Thus, the searching speed of both algorithm are faster than the algorithm without any heuristic.

pros of using the Minimum-Remaining Values heuristic:

① The Minimum-Remaining Values heuristic is simple and straightforward to implement. However, the DH-LCV needs you to find the most constraint variable and then find the least constraint value, which is hard to implement.

② The Minimum-Remaining Value heuristic require less computation overhead compare to Degree heuristic combined with least Constraining Values Thus, the complexity of Minimum-Remaining Values heuristic is less than the Degree heuristic.

③ The Minimum-Remaining Values heuristic is faster than the Degree heuristic combined with least Constraining Values at beginig, since the choice of variable in the minimum-Remaining Value are less than the degree heuristic. Thus, these steps speed up the search process by focusing on variable that more likely to be assigned a value that satisfies the constraints.

④ The Minimum-Remaining Values heuristic is faster than the another heuristic at the time of picking values, since the least constraint value heuristic will consider which one is least constraint value.

Cons: ① In this case, although the Minimum-Remaining heuristic is faster at begining, it cannot efficiently prune some cases that are clearly wrong or cannot be sdution by the same reason. Here we have several variables with same domain size, so the variable that constraint other one most cannot be picked first.

② The order of picking will affect the minimum-Remaining heurstic. For example, family parents picking from second row will find sdution quicker than picking from first row. The heuristic also cannot tell whether wheelchair user or elderly individual is better.

③ This minimum-Remaining heuristic might occur some worst case since the choice at first is less likely to be to correct choice that need to be chosen at final. Thus, the search-tree might go very deep and find this situation is totally wrong and then go back. For example, when the search tree go all the deepest branch of the wheel chair user, the algorithm can know wheelchair cannot be on second row. This will lead to a lot of redundant work.

I would like to choose Degree-heurstic with least Constrainting Values. First, this method would quickly help the most constrainting-varinbles find the seats that are less likely to have a conflict with other variables, so other varinbles will have more choices and the selected seats are more likely to the final choice. For example, in this case, we would choose family members first and we will immediately set them in the second row, so five of the seats are selected correctly and this will efficiently reduce the searching time for other variables Secondly, with the help of least-constrainting Value, this heuristic would quickly handle the most constraint variable with the value that let other variables have more choices Thus, other varinbles will more likely to find the suitable solution and are less likely to get an empty set.

5 Part B: Search (50%)

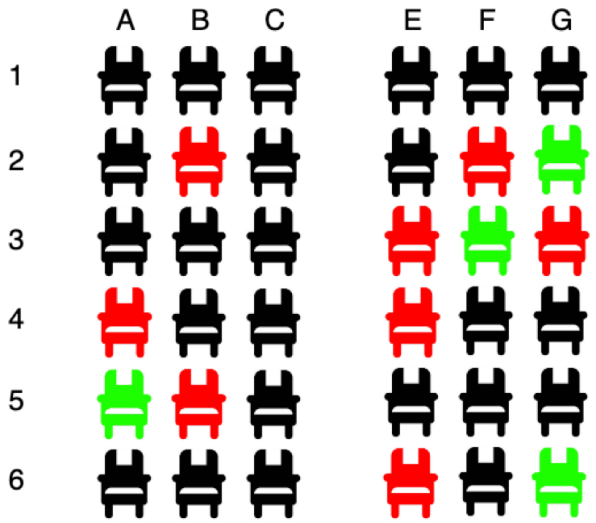


Figure 3: Plane layout for Part B. Red seats indicate sleeping passengers while green seats indicate the passengers who need to receive forms.

2. In this question you will be asked to find an efficient path to get a set of forms from the passenger seated in A1 to the passengers seated in green seats indicated in Figure 3. The forms can only be passed ahead, behind, left, or right, and not diagonally. In this problem, we will consider the distance between aisles C and E as double the distance between aisles B and C as the passengers.

   (a) (4 points) Define an appropriate measure for the distance between two passengers.

Qa.

We can use the manhattan distance as the measure for the distance between two passengers, which means the addition of the row distance and the column distance between two passengers. If two seats are in different sides of the plane such as 3C and 3E, we need add one more distance, which means count aisle as 1 distance.

(b) (6 points) A* is designed to find a path from a start state to a goal, in this case, we have multiple goals. Write pseudocode to define an algorithm that uses the above metric to find a suitable order in which to achieve the goals. Your algorithm should ignore the sleeping passengers.

Pseudocode:

```
# function used to find the distance between two points.
function dis(point A , point B) return the distance.
    if point A in left side of the plane and
    point B in right side of the plane then
    distance = the sum of the difference between the rows
    of two points and the difference between the
    columns of two points plusing one
    elif point A in right side of the plane and
    point B in the left side of the plane then.
    distance = the sum of the difference between the rows
    of two points and the difference between the
    columns of two points plusing one
    else:
        distance = the sum of the difference between
        the rows of two points and the difference
        between the columns of two points.
    return distance
```

```
# a tsp recursion function to find the shortest path
# to several goals with inputs goals(a set of goal
# states, length(current length of trying path), record_path(trying path)
# Initially, length is 0 and record list is [Point(1A)]
function tsp(goals, length, record_path) return length of shortest path, shortest path
    # terminal state of the recursion
    if length of goals is zero
        return length, record_path
    else:
        shortest = ∞    # set the initial length of shortest path be ∞
        solution_path = [ ] # set the initial shortest path be an empty list
        for each goal in goals do
            copy_length ← the copy of current length of trying path
            copy_record_path ← the copy of current trying path
            copy_goals ← the copy of current goals
            copy_length ← copy_length plus the dis(last element of
                          the copy_record_path, goal)
            add this goal to the copy_record_path
            remove this goal from the copy_goals
        # use recursion to find the length of each possible path
        possible path ← tsp(copy_goals, copy_length, copy_record_path)
        # use the first element in possible path, which is the length of
        # this possible path to compare with shortest, if possible
        # path is less than shortest, then replace shortest with
        # possible path
```

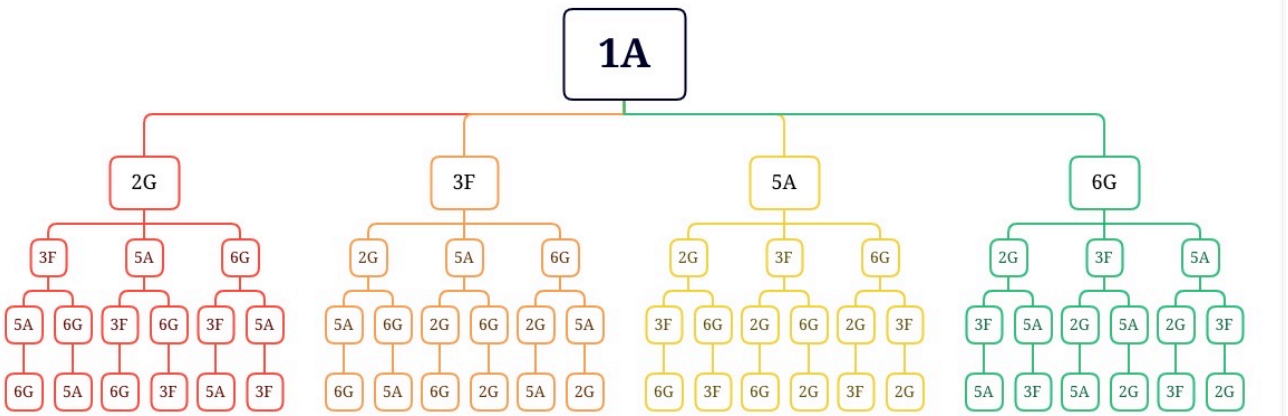
```
if the length of possible path < shortest then
    shortest ← the length of possible path
    solution_path ← the record path of possible path
return shortest , solution_path
```

This algorithm use recursion to find all possible path going through each goal in goals' set from starting point (A. Then, it compares all the length of possible path to find the shortest path, so this order(path) must be an optimal path for these goals.

(c) (10 points) Use the algorithm you defined in the previous part to find the order in which the goals should be met.

The order found in my algorithm is
{1A, 5A, 3F, 2G, 6G}

The basic theorem of the algorithm can be show as a tree



The result of each possible can be shown in python

```
23 [[1, 1], [2, 6], [3, 5], [5, 1], [6, 6]]
20 [[1, 1], [2, 6], [3, 5], [6, 6], [5, 1]]
27 [[1, 1], [2, 6], [5, 1], [3, 5], [6, 6]]
27 [[1, 1], [2, 6], [5, 1], [6, 6], [3, 5]]
22 [[1, 1], [2, 6], [6, 6], [3, 5], [5, 1]]
25 [[1, 1], [2, 6], [6, 6], [5, 1], [3, 5]]
25 [[1, 1], [3, 5], [2, 6], [5, 1], [6, 6]]
20 [[1, 1], [3, 5], [2, 6], [6, 6], [5, 1]]
27 [[1, 1], [3, 5], [5, 1], [2, 6], [6, 6]]
25 [[1, 1], [3, 5], [5, 1], [6, 6], [2, 6]]
24 [[1, 1], [3, 5], [6, 6], [2, 6], [5, 1]]
27 [[1, 1], [3, 5], [6, 6], [5, 1], [2, 6]]
19 [[1, 1], [5, 1], [2, 6], [3, 5], [6, 6]]
21 [[1, 1], [5, 1], [2, 6], [6, 6], [3, 5]]
17 [[1, 1], [5, 1], [3, 5], [2, 6], [6, 6]]
19 [[1, 1], [5, 1], [3, 5], [6, 6], [2, 6]]
17 [[1, 1], [5, 1], [6, 6], [2, 6], [3, 5]]
17 [[1, 1], [5, 1], [6, 6], [3, 5], [2, 6]]
24 [[1, 1], [6, 6], [2, 6], [3, 5], [5, 1]]
31 [[1, 1], [6, 6], [2, 6], [5, 1], [3, 5]]
26 [[1, 1], [6, 6], [3, 5], [2, 6], [5, 1]]
31 [[1, 1], [6, 6], [3, 5], [5, 1], [2, 6]]
29 [[1, 1], [6, 6], [5, 1], [2, 6], [3, 5]]
27 [[1, 1], [6, 6], [5, 1], [3, 5], [2, 6]]
```

The first integer in each row is the length of each possible path and the list followed behind is the possible path.

Here [1,1] = 1A [2,6] = 2G
[3,5] = 3F [5,1] = 5A
[6,6] = 6G

Here I'm going to show how we come out the first row.

Here we set tsp([[2,6],[3,5],[5,1],[6,6]],0,[[1,1]])

then it calls tsp function

Here goals is [[2,6],[3,5],[5,1],[6,6]] which is not zero then we go to else.

The first goal in goals is [2,6]. After copying the inputs from tsp function, we renew our copy-length variable adding the distance from initial state 1A to 2G. Here we call dis function with input (1A, 2G). Since 1A is in left side and 2G is in right side, the distance should be (|2-1|+|6-1|+1) = 7 . Then we add our first goal [2,6] to the copy-record-path. Now, copy-record_path = [[1,1],[2,6]]. Removing goal [2,6] from copy-goals, we have copy-goals ([[3,5],[5,1],[6,6])

Then we call tsp again, but with our new inputs:

tsp([[3,5],[5,1],[6,6]], 7, [[1,1],[2,6]])

Here we go into new tsp function. Goals is [[3,5],[5,1],[6,6]], which is not zero, so we go to else. Now our first goal is [3,5].

After copying the inputs from new tsp function, we renew our copy-length variable adding the distance from 2,G to 3F. Here we call dis function with inputs (2G, 3F). Since 2G and 3F all in same side, the distance between these two is (|3-2|+|5-6|)=2. Here copy-length become 7+2=9.

Then we add goal [3,5] to the copy-record_path. Now, copy-record-path = [[1,1], [2,6], [3,5]]. Removing goal [3,5] from copy-goals, our copy-goals become [[5,1],[6,6]].

Then we call tsp again with our new inputs:

tsp([[5,1],[6,6]], 9, [[1,1],[2,6],[3,5]])

Here we go into new tsp function. Goals is [[5,1],[6,6]], which is not zero, so we go to else. Now our first goal is [5,1]

After copying the inputs from new tsp function, we renew our copy_length variable adding the distance from 3F to 5A. Here we call dis function with inputs (3F, 5A). Since 3F in left side and 5A in right side, the distance between these two is (|5-3|+|1-5|+1)=7. Here copy_length become 9+7=16

Then we add goal [5,1] to the copy_record_path. Now, copy_record_path=[[1,1],[2,6],[3,5],[5,1]]. Removing goal [5,1] from copy_goals, our copy_goals become [[6,6]]

Then we call tsp again with our new inputs:

tsp([[6,6]], 16, [[1,1],[2,6],[3,5],[5,1]])

Here we go into new tsp function. Goals is [[6,6]], which is not zero, so we go to else. Now our first goal is [6,6]

After copying the inputs from new tsp function, we renew our copy_length variable adding the distance from 5A to 6G. Here we call dis function with inputs (5A, 6G). Since 5A is in left side and 6G is in right side, the distance between these two is (|6-5|+|6-1|+1)=7. Here copy_length become 16+7=23

Then we add goal [6,6] to the copy_record_path. Now, copy_record_path=[[1,1],[2,6],[3,5],[5,1],[6,6]] Removing goal [6,6] from copy_goals, our copy_goals become []

Then we call tsp again with our new inputs:

tsp([] , 23, [[1,1], [2,6], [3,5], [5,1], [6,6])

Here goals is zero, then we return 23 and [[1,1], [2,6], [3,5], [5,1], [6,6]]. Since we go to the end we back to this point of calling. Here we set possible path as (23, [[1,1], [2,6], [3,5], [5,1], [6,6]]). Then we need to compare 23 with shortest variable (∞ at first). Then we replace shortest by 23 and solution_path by [[1,1], [2,6], [3,5], [5,1], [6,6]]. Then our algorithm go to last for loop from [5,1], [6,6]] and choose 6G.

Keeping doing this step by step and compare each possible path with shortest. Finally we will get the shortest path.

length of possible path | possible path | shortest | solution_path

23	[[1, 1], [2, 6], [3, 5], [5, 1], [6, 6]]	23	[[1, 1], [2, 6], [3, 5], [5, 1], [6, 6]]
20	[[1, 1], [2, 6], [3, 5], [6, 6], [5, 1]]	20	[[1, 1], [2, 6], [3, 5], [6, 6], [5, 1]]
27	[[1, 1], [2, 6], [5, 1], [3, 5], [6, 6]]	20	[[1, 1], [2, 6], [3, 5], [6, 6], [5, 1]]
27	[[1, 1], [2, 6], [5, 1], [6, 6], [3, 5]]	20	[[1, 1], [2, 6], [3, 5], [6, 6], [5, 1]]
22	[[1, 1], [2, 6], [6, 6], [3, 5], [5, 1]]	20	[[1, 1], [2, 6], [3, 5], [6, 6], [5, 1]]
25	[[1, 1], [2, 6], [6, 6], [5, 1], [3, 5]]	20	[[1, 1], [2, 6], [3, 5], [6, 6], [5, 1]]
25	[[1, 1], [3, 5], [2, 6], [5, 1], [6, 6]]	20	[[1, 1], [2, 6], [3, 5], [6, 6], [5, 1]]
20	[[1, 1], [3, 5], [2, 6], [6, 6], [5, 1]]	20	[[1, 1], [2, 6], [3, 5], [6, 6], [5, 1]]
27	[[1, 1], [3, 5], [5, 1], [2, 6], [6, 6]]	20	[[1, 1], [2, 6], [3, 5], [6, 6], [5, 1]]
25	[[1, 1], [3, 5], [5, 1], [6, 6], [2, 6]]	20	[[1, 1], [2, 6], [3, 5], [6, 6], [5, 1]]
24	[[1, 1], [3, 5], [6, 6], [2, 6], [5, 1]]	20	[[1, 1], [2, 6], [3, 5], [6, 6], [5, 1]]
27	[[1, 1], [3, 5], [6, 6], [5, 1], [2, 6]]	20	[[1, 1], [2, 6], [3, 5], [6, 6], [5, 1]]
19	[[1, 1], [5, 1], [2, 6], [3, 5], [6, 6]]	19	[[1, 1], [5, 1], [2, 6], [3, 5], [6, 6]]
21	[[1, 1], [5, 1], [2, 6], [6, 6], [3, 5]]	19	[[1, 1], [5, 1], [2, 6], [3, 5], [6, 6]]
17	[[1, 1], [5, 1], [3, 5], [2, 6], [6, 6]]	17	[[1, 1], [5, 1], [3, 5], [2, 6], [6, 6]]
19	[[1, 1], [5, 1], [3, 5], [6, 6], [2, 6]]	17	[[1, 1], [5, 1], [3, 5], [2, 6], [6, 6]]
17	[[1, 1], [5, 1], [6, 6], [2, 6], [3, 5]]	17	[[1, 1], [5, 1], [3, 5], [2, 6], [6, 6]]
17	[[1, 1], [5, 1], [6, 6], [3, 5], [2, 6]]	17	[[1, 1], [5, 1], [3, 5], [2, 6], [6, 6]]
24	[[1, 1], [6, 6], [2, 6], [3, 5], [5, 1]]	17	[[1, 1], [5, 1], [3, 5], [2, 6], [6, 6]]
31	[[1, 1], [6, 6], [2, 6], [5, 1], [3, 5]]	17	[[1, 1], [5, 1], [3, 5], [2, 6], [6, 6]]
26	[[1, 1], [6, 6], [3, 5], [2, 6], [5, 1]]	17	[[1, 1], [5, 1], [3, 5], [2, 6], [6, 6]]
31	[[1, 1], [6, 6], [3, 5], [5, 1], [2, 6]]	17	[[1, 1], [5, 1], [3, 5], [2, 6], [6, 6]]
29	[[1, 1], [6, 6], [5, 1], [2, 6], [3, 5]]	17	[[1, 1], [5, 1], [3, 5], [2, 6], [6, 6]]
27	[[1, 1], [6, 6], [5, 1], [3, 5], [2, 6]]	17	[[1, 1], [5, 1], [3, 5], [2, 6], [6, 6]]

Left graph shows how the algorithm work.

Here, at the end of comparing, we get our final shortest and solution_path `(17, [[1, 1], [5, 1], [3, 5], [2, 6], [6, 6]])`

(17, [1A, 5A, 3F, 2G, 6G])

(d) (12 points) Use A* search to find a path from 1A to each of the goals in the order you determined in the previous part. Assume that the passengers in red seats are asleep and cannot be passed the forms. Write down the final set of paths, the nodes searched in each search run, and the final frontier as well as any additional working.

Qd

the order should be 1A, 5A, 3F, 2G, 6G

We define $f(n) = h(n) + g(n)$ $h(n)$ is the cost from start point $g(n)$ is the mahattan distance from start point to goal point.

First we use A* search from 1A to 5A

① in first step we have two choices 1B and 2A

$f(1B) = 1+5 = 6$ $f(2A) = 1+3 = 4$, so we choose 2A

② We have 1B and 3A $f(1B) = 6$ $f(3A) = 2+2 = 4$, so we choose 3A

③ We have 1B and 3B $f(1B) = 6$ $f(3B) = 3+3 = 6$ we pick by row-wise, which is 1B

④ We have 3B and 1C $f(3B) = 6$ $f(1C) = 2+6 = 8$ we choose 3B

⑤ We have 4B and 1C and 3C

$f(4B) = 4+2 = 6$ $f(1C) = 8$ $f(3C) = 4+4 = 8$

We pick 4B

⑥ We have 1C and 3C and 4C

$f(1C) = 8$ $f(3C) = 8$ $f(4C) = 5+3 = 8$

By row-wise, we pick 1C

⑦ We have 2C, 3C, 4C, 1E

$f(2C) = 3+5 = 8$ $f(3C) = 8$ $f(4C) = 8$ $f(1E) = 4+8 = 12$

We pick 2C

⑧ We have 3C, 4C, 1E, 2E

f(3C)=8 f(4C)=8 F(1E)=12 f(2E)=5+7=12

We pick 3C

⑨ We have 4C, 1E, 2E

f(4C)=8 F(1E)=12 f(2E)=12

We pick 4C

⑩ We have 5C, 1E, 2E

f(5C)=6+2=8 F(1E)=12 f(2E)=12

We pick 5C

⑪ We have 6C, 1E, 2E, 5E

f(6C)=7+3=10 F(1E)=12 f(2E)=12 f(5E)=8+4=12

We pick 6C

⑫ We have 6B, 1E, 2E, 5E

f(6B)=8+2=10 F(1E)=12 f(2E)=12 f(5E)=12

We pick 6B

⑬ We have 1E, 2E, 5E, 6A

F(1E)=12 f(2E)=12 f(5E)=12 f(6A)=9+1=10

We pick 6A

⑭ We have 1E, 2E, 5E, 5A

F(1E)=12 f(2E)=12 f(5E)=12 f(5A)=10

We pick 5A, since 5A is final goal

Final frontier {1E, 2E, 5E}

We find a solution from 1A to 5A with path

{1A, 2A, 3A, 3B, 4B, 4C, 5C, 6C, 6B, 6A, 5A}

Then by the order that we found before, we choose 5A as start and 3F as goal

① First we have 6A
$f(6A) = 1+8 = 9$
we pick 6A

② we have 6B
$f(6B) = 2+7 = 9$
we pick 6B

③ we have 6C
$f(6C) = 3+6 = 9$
we pick 6C

④ we have 5C
$f(5C) = 4+5 = 9$
we pick 5C

⑤ we have 4C, 5E
$f(4C) = 5+4 = 9$ $f(5E) = 6+3 = 9$
we pick 4C due to row-wise.

⑥ we have 3C, 4B, 5E
$f(3C) = 6+3 = 9$ $f(4B) = 6+5 = 11$ $f(5E) = 9$
we pick 3C

⑦ we have 2C, 3B, 4B, 5E
$f(2C) = 7+4 = 11$ $f(3B) = 7+4 = 11$
$f(4B) = 11$ $f(5E) = 9$
we pick 5E

⑧ we have 2C, 3B, 4B, 5F

f(2C)=11 f(3B)=11 f(4B)=11 f(5F)=6+3=9

we pick 5F

⑨ we have 2C, 3B, 4B, 4F, 5G, 6F

f(2C)=11 f(3B)=11 f(4B)=11

f(4F)=8+1=9 f(5G)=8+3=11 f(6F)=8+3=11

we pick 4F

⑩ we have 2C, 3B, 3F, 4B, 4G, 5G, 6F

f(2C)=11 f(3B)=11 f(3F)=9 f(4B)=11

f(4G)=9+2=11 f(5G)=11 f(6F)=11

we pick 3F. Here we found our goal 3F

Final frontier {2C, 3B, 4B, 4G, 5G, 6F}

The path is {5A, 6A, 6B, 6C, 5C, 5E, 5F, 4F, 3F}

Here we find our second goal, then we need to find next goal 2G from 3F

① we have 4F

F(4F)=1+3=4

we pick 4F

② we have 4G, 5F

F(4G)=2+2=4 F(5F)=2+4=6

we pick 4G

③ we have 5F, 5G

F(5F)=6 F(5G)=3+3=6

we pick 5F

④ we have 5E, 5G, 6F
F(5E)=3+5=8 F(5G)=3+3=6 F(6F)=4+5=9
we pick 5G

⑤ We have 5E, 6G, 6F
F(5E)=8 F(6G)=4+4=8 F(6F)=9
we pick 5E

⑥ We have 5C, 6G, 6F
F(5C)=5+7=12 F(6G)=8 F(6F)=9
We pick 6G

⑦ We have 5C, 6F
F(5C)=12 F(6F)=9
We pick 6F

⑧ We have 5C
F(5C)=12
We pick 5C

⑨ We have 4C, 6C
F(4C)=6+6=12 F(6C)=6+8=14
We pick 4C

⑩ We have 3C, 4B, 6C
F(3C)=7+5=12 F(4B)=7+7=14 F(6C)=14
We pick 3C

⑪ We have 2C, 3B, 4B, 6C
F(2C)=8+4=12 F(3B)=8+6=14
F(4B)=14 F(6C)=14
We pick 2C

(12) We have 1C, 2E, 3B, 4B, 6C
F(1C) = 9+5=14 F(2E) = 10+2=12
F(3B) = 14 F(4B) = 14 F(6C) = 14
we pick 2E

(13) We have 1C, 1E, 3B, 4B, 6C
F(1C) = 14 F(1E) = 11+3=14 F(3B) = 14
F(4B) = 14 F(6C) = 14
we pick 1C

(14) We have 1B, 1E, 3B, 4B, 6C
F(1B) = 10+6=16 F(1E) = 14 F(3B) = 14
F(4B) = 14 F(6C) = 14
we pick 1E

(15) We have 1B, 1F, 3B, 4B, 6C
F(1B) = 16 F(1F) = 12+2=14 F(3B) = 14
F(4B) = 14 F(6C) = 14
we pick 1F

(16) We have 1B, 1G, 3B, 4B, 6C
F(1B) = 16 F(1G) = 13+1=14 F(3B) = 14
F(4B) = 14 F(6C) = 14
We pick 1G

(17) We have 1B, 2G, 3B, 4B, 6C
F(1B) = 16 F(2G) = 14+0=14 F(3B) = 14
F(4B) = 14 F(6C) = 14

We pick 2G and 2G is our goal we end this steps of search.

Final frontier = {1B, 3B, 4B, 6C}
Path = {3F, 4F, 5F, 5E, 5C, 4C, 3C, 2C, 2E, 1E, 1F, 1G, 2G}

Now we start our final step from 2G to 6G

① We have 1G
F(1G) = 1+5 = 6
We pick 1G

② We have 1F
F(1F) = 2+6 = 8

③ We have 1E
F(1E) = 3+7 = 10
We pick 1E

④ We have 1C, 2E
F(1C) = 5+9 = 14 F(2E) = 4+6 = 10
we pick 2E

⑤ We have 1C, 2C
F(1C) = 5+9 = 14 F(2C) = 6+8 = 14
we pick 1C

⑥ We have 1B, 2C
F(1B) = 6+10 = 16 F(2C) = 14
we pick 2C

⑦ We have 1B, 3C
F(1B) = 16 F(3C) = 7+7 = 14
we pick 3C

⑧ We have 1B, 3B, 4C
F(1B)=16 F(3B)=8+8=16 F(4C)=8+6=14
we pick 4C

⑨ We have 1B, 3B, 4B, 5C
F(1B)=16 F(3B)=16 F(4B)=9+7=16
F(5C)=9+5=14
we pick 5C

⑩ We have 1B, 3B, 4B, 5E, 6C
F(1B)=16 F(3B)=16 F(4B)=16
F(5E)=11+3=14 F(6C)=10+4=14
we pick 5E

⑪ We have 1B, 3B, 4B, 5F, 6C
F(1B)=16 F(3B)=16 F(4B)=16
F(5F)=12+2=14 F(6C)=14
we pick 5F

⑫ We have 1B, 3B, 4B, 4F, 5G, 6C, 6F
F(1B)=16 F(3B)=16 F(4B)=16
F(4F)=13+3=16 F(5G)=13+1=14 F(6C)=14 F(6F)=13+1=14
we pick 5G

⑬ We have 1B, 3B, 4B, 4F, 4G, 6C, 6F, 6G
F(1B)=16 F(3B)=16 F(4B)=16 F(4F)=16
F(4G)=14+2=16 F(6C)=14 F(6F)=14 F(6G)=14
we pick 6C

⑭ We have 1B, 3B, 4B, 4F, 4G, 6B, 6F, 6G

F(1B)=16 F(3B)=16 F(4B)=16 F(4F)=16
F(4G)=16 F(6B)=11+5=16 F(6F)=14 F(6G)=14
we pick 6F

(15) We have 1B, 3B, 4B, 4F, 4G, 6B, 6G
F(1B)=16 F(3B)=16 F(4B)=16 F(4F)=16
F(4G)=16 F(6B)=16 F(6G)=14
we pick 6G, since 6G is our goal
We have found our final goal.
The Path is {2G, 1G, 1F, 1E, 2E, 2C, 3C, 4C, 5C
5E, 5F, 5G 6G}

Find frontier = {1B, 3B, 4B, 4F, 4G, 6B}

The find set of Path is {1A, 2A, 3A, 3B, 4B, 4C, 5C,
6C, 6B, 6A, 5A, 6A, 6B, 6C, 5C, 5E, 5F, 4F, 3F, 4F,
5F, 5E, 5C, 4C, 3C, 2C, 2E, 1E, 1F, 1G, 2G, 1G, 1F
1E, 2E, 2C, 3C, 4C, 5C, 5E, 5F, 5G, 6G}

(e) (4 points) What are some of the limitations of the approach outlined in parts (b)-(d). Can you suggest some potential improvements?

① In part b, we find an optimal (shortest) path for several goals. but, in part d, there are more limitations in the plane such as not passing through red seat. Therefore, the order calculated in part b is not suitable for part d now. For example, with order {1A, 5A, 3F, 2G, 6G}. obviously, from 3F to 2G is no longer shorter than from 3F to 6G due to the red seat. Thus, it will take much more work from 3F to 2G and back to 6G.

② Although the heuristic is admissible, it cannot be a useful heuristic, since the distance from one goal to another might actually larger than the heuristic a lot. Then, the a star might find the goal slower.

③ On the way of find one goal, we might find another goal is on the path of another goal. Thus, this will occur redundant path going back to this goal again.

To figure out ① problem, we can search all possible path in the new given situation and compare them one by one as the algorithm in part b. Therefore, we will find the optimal path with this given restriction.

for problem ②, we might use another heuristic that count the sleeping passenger into the heuristic. For example, we could also let sleeping passenger have heavy weight in the heuristic.

For problem ③, if we go thrayh one goal and we can add it to the current path and delete it from the goals set that still need to be found.

However, the solution of ② and ③ might not be optimal, then we just use the solution of ① in the 2f.

(f) (6 points) Define an algorithm in pseudocode based on the suggestions in the previous parts.

Pseudocode:

```
# function used to find the manhattan distance between two points.
function dis(point A, point B) return the distance.
    if point A in left side of the plane and
    point B in right side of the plane then
    distance = the sum of the difference between the rows
    of two points and the difference between the
    columns of two points plusing one
  elif point A in right side of the plane and
    point B in the left side of the plane then.
    distance = the sum of the difference between the rows
    of two points and the difference between the
    columns of two points plusing one
  else:
    distance = the sum of the difference between
    the rows of two points and the difference
    between the columns of two points.
  return distance

# construct path from parent to current node
# and then from parent's parent to parent till the start
function reconstruct_path(cameFrom, current)
        total_path ← current
        while current in cameFrom.keys:
```

```
        current ← cameFrom[current]
        total_path.prepend(current)
    return total_path

# Use A* find the shortest path from each element
# in goals to other element in goals and the start point
# to each element.
function A_star(restriction, start, end) return a solution, or failure
    open_set ← an empty list
# Here restriction is the seats surrounding these 18 seats, and red seats.
    close_set ← restriction
    add start to open_set and set its score to be 0
# cameFrom[n] is the node immediately preceding it on cheapest path from start to current
    cameFrom ← an empty map
# cost[n] is the cost of the cheapest path from start to n
    cost ← map with default value of infinity
    cost[start] ← 0
# score[n] is the sum of cost[n] and mahattan distance
    score ← map with default value of infinity
    score[start] = dis(start, end)
    while open_set is not empty
        current ← the node in open_set with lowest score
        if current is end then:
```

```
        # track the parent of current till the start
        return reconstruct_path(cameFrom, current)
    else:
        remove current from open_set
        add current to close_set
        for each neighbour of current then
            # tentative_score is the distance from start to neighbour
            tentative_score = cost[current] + dis(current, neighbour)
            if neighbour is in close_set then
                jump this neighbour and choose next one
            if tentative_score < cost[neighbour]
                # this path to neighbour is better than previous
                cameFrom[neighbour] <- current
                cost[neighbour] <- tentative_score
                score[neighbour] <- tentative_score + dis(neighbour, end)
                if neighbour not in open_set
                    add neighbour to open_set
return failure

# count the length of path.
function path_length(path): return the length of the path
    total_length <- 0
    while path is not just one element:
        total_path  total-path + dis(path[0], path[1])
        remove path[0] from path
```

```
return total_length

# a tsp_restriction recursion function to find the shortest path
# to several goals with inputs goals(a set of goal
# states, length(current length of trying path), record_path(trying path),
# restriction(red seats and seats surrounding 18 seats
#Initially, length is 0 and record list is [Point(1A)]
  function tsp_restriction(goals, length, record_path, restriction)
                                return length of shortest path, shortest path or failure

  # terminal state of the recursion
    if length of goals is zero
       return length, record_path
    else:
       shortest = ∞      # set the initial length of shortest path be ∞
       solution_path = [] # set the initial shortest path be an empty list
       for each goal in goals do
           copy_length ← the copy of current length of trying path
           copy_record_path ← the copy of current trying path
           copy_goals ← the copy of current goals
         # if there is no path from one goal to another, then there will be no solution
           if path(A_star(restriction, last element of copy of record path, goal) is equal to failure:
              return failure
           copy_length ← copy_length plus the path_length(A_star(
                                restriction, last element of copy_record_path, goal))
```

add this goal to the copy_record_path

remove this goal from the copy_goals

```
# use recursion to find the length of each possible path
possible_path ← tsp_restriction(copy_goals, copy_length, copy_record_path, restriction)
# use the first element in possible_path, which is the length of
# this possible path to compare with shortest, if possible
# path is less than shortest, then replace shortest with
# possible path
if the length of possible_path < shortest then
    shortest ← the length of possible_path
    solution_path ← the record path of possible_path
return shortest, solution_path
```

(g) (8 points) Compare how the two approaches would perform with different configurations of the same problem, i.e. with different goals and sleeping passengers. [***Hint:*** *A high level comparison is sufficient. You don't need to provide a search tree for each configuration.*]

Complexity

First, the former algorithm's searching speed is much more faster than new algorithm.

The worst-case performace of former algorithm is $\Theta(n!) + O(|E| \cdot n)$

The worst-case performace of new algorithm is $O(n! \cdot |E|)$

Absolutely, new algorithm will be much more slower than former algorithm with a larger n. Thus, if we have a large number of goals, the new algorithm will perform slower than former one.

If the sleeping passengers surrounding the goals, then each a star search will be much slower than the situation that sleeping passengers are not on the path from one goal to another. Thus, the new algorithm will be much slower than the former one, since the new algorithm use a star more than the former one.

Secondly, like the time complexity, the space complexity of new algorithm is much more complex than former one.

The space complexity for former one is
$O(n!) + O(|V| \cdot n)$

The space complexity for new one is
$O(n! \cdot |V|)$

These all depends on n, if n is large $n!$ will be much larger than n showing that the space complexity of the new algorithm will extremely larger than the former one.

If the sleeping passengers affects the path a lot, then the space complexity of the new algorithm will be higher than the former one.

Optimal

Here the former algorithm is likely to get a solution which is not optimal. Since the order found in this algorithm is based on the plane without sleeping passengers. Therefore, the situation that two closed passengers might be split by the sleeping passengers will ocurr, which will make the path much longer than going to the second nearest passenger first without sleeping passenger.

However, the new algorithm find out the optimal path. The new algorithm find out all the possible path within the plane with sleeping passengers and then compare all the path to find out the shortest path. Since it lists all the path in the given plane, the path picked in the final must be optimal.

Complete

Two algorithm are both complete. They will all find the path. f there is no path from one goal to another, two algorithm will both return failure. There won't be any infinite loop

Conclusion: Both algorithms can find a path, so both of them are complete. The difference is in the complexity and optimality. If you want optimality, you will get a high complexity. For a huge problem with huge amounts of goals and sleeping passengers, this problem is NP-hard and cannot be solved by the computer. Thus, we need to sacrifice the optimality to get the solution by reducing the complexity. If we have a small amount of goals, the new algorithm is a better choice for finding a optimal choice.