Map_and_Block library

In [1]:
```
import numpy as np
import copy
```

We declare `Block` class, whose objects are blocks of the game.
A block is defined by its *coordinate*, *direction* (either **vertical** or **horizontal**), *length* (either **2** or **3**).
Also, a function `move` is defined to help the block move in the map

In [2]:
```
class Block:
    """
    x range from 0 to 5
    y range from 0 to 5
    """
    def __init__(self, x, y, direction, length):
        global index
        self.start_point_x = x
        self.start_point_y = y
        self.direction = direction
        self.length = length
        self.indx = index
        index += 1          #when you initialize next item, its index will raise

    def __str__(self):
        a = 'start_point_x = %d\
        start_point_y = %d\
        direction = %s\
        length = %d\
        index  = %d'\
            % (int(self.start_point_x),\
            int(self.start_point_y),\
            self.direction,\
            int(self.length),\
            int(self.indx))
        return a

    def move(self, step, direct):
        """
        :param step: int
        :param direct: -1/+1
        :return: new block
        """
        if self.direction == "h":
            self.start_point_x += step * direct
        if self.direction == "v":
            self.start_point_y += step * direct
```

Now, `Map` class, represented by a numpy **6x6 matrix**
Some basic functions for `Map` class:

- `add_block` : add an object from class `Block` to our map
- `possible_move` : declare **every** possible move of **every** block in the map

In [3]:
```
class Map:
    def __init__(self):
        self.map = np.array([[0 for i in range(6)] for i in range(6)])

    def __str__(self):
        return str(self.map)

    def add_block(self, blk):
        x, y, direction, length, index = (
            blk.start_point_x,
            blk.start_point_y,
            blk.direction,
            blk.length,
            blk.indx,
        )
        if direction == "h":
            for i in range(length):
                self.map[y][x + i] = index
        if direction == "v":
            for i in range(length):
                self.map[y + i][x] = index

    def possible_move(self, blk):
        x, y, direction, length, index = (
            blk.start_point_x,
            blk.start_point_y,
            blk.direction,
            blk.length,
            blk.indx
        )
        move_list = []
        if direction == "v":
            if y == 0:
                up = 0
            else:
```

```
                for up in range(1, y + 1):
                    if self.map[y - up][x] != 0:
                        up -= 1
                        break
                    else:
                        move_list.append((up, -1, index))

            if y + length == 6:
                down = 0
            else:
                for down in range(1, 7 - y - length):
                    if self.map[y + length + down - 1][x] != 0:
                        down -= 1
                        break
                    else:
                        move_list.append((down, +1, index))

        if direction == "h":
            if x == 0:
                left = 0
            else:
                for left in range(1, x + 1):
                    if self.map[y][x - left] != 0:
                        left -= 1
                        break
                    else:
                        move_list.append((left, -1, index))
            if x + length == 6:
                right = 0
            else:
                for right in range(1, 7 - x - length):
                    if self.map[y][x + length + right - 1] != 0:
                        right -= 1
                        break
                    else:
                        move_list.append((right, +1, index))
        return move_list
```

Finally, `State` class, the **core of the searching algorithms**. A state contains the map and all its blocks.
Its functions are:

- `GetMap` returns the map itself for future functions and then `Display` prints the map - `GetNextMoves` returns the list of possible moves, calculated in `possible_move` function above - `NextStates` with parameter **move** from `GetNextMove`, returns the class `State` of the map obtained **after the move**

In [4]:

```
class State():
    def __init__(self, AllBlocks):
        self.AllBlocks = AllBlocks
        self.GameMap = Map()
        for block in self.AllBlocks:
            self.GameMap.add_block(block)
    def GetMap(self):
        return self.GameMap

    def GetNextMoves(self):
        All_Moves_list = []
        a = self.GetMap()
        for blk in self.AllBlocks:
            All_Moves_list += a.possible_move(blk)
        return All_Moves_list

    def Display(self):
        return self.GameMap.__str__()

    def NextStates(self, move):
        New = []
        for block in self.AllBlocks:
            TempBlock = copy.copy(block)
            if TempBlock.indx == move[2]:
                TempBlock.move(move[1], move[0])
            New.append(TempBlock)
        return State(New)
```

Function `read_input` helps initializing the files *inp(x).txt* where x is the number of input set

In [5]:

```
def read_input(file_name): # read a text file and return an instance of Map class
    m = Map()
    AllBlocks = []
    with open(file_name) as f:
        list_of_blocks = f.readlines()[:]
        number_of_blocks = len(list_of_blocks)
        for i in range(number_of_blocks):
            if i == number_of_blocks-1:
                block = list_of_blocks[i].strip()
            else:
                block = list_of_blocks[i][:-1].strip()
            x = int(block[0])
            y = int(block[2])
            direc = block[4]
            length = int(block[-1])
            blk = Block(x, y, direc, length)
            AllBlocks.append(blk)
```

```
            m.add_block(blk)
    return m, AllBlocks
```

The map is displayed as below

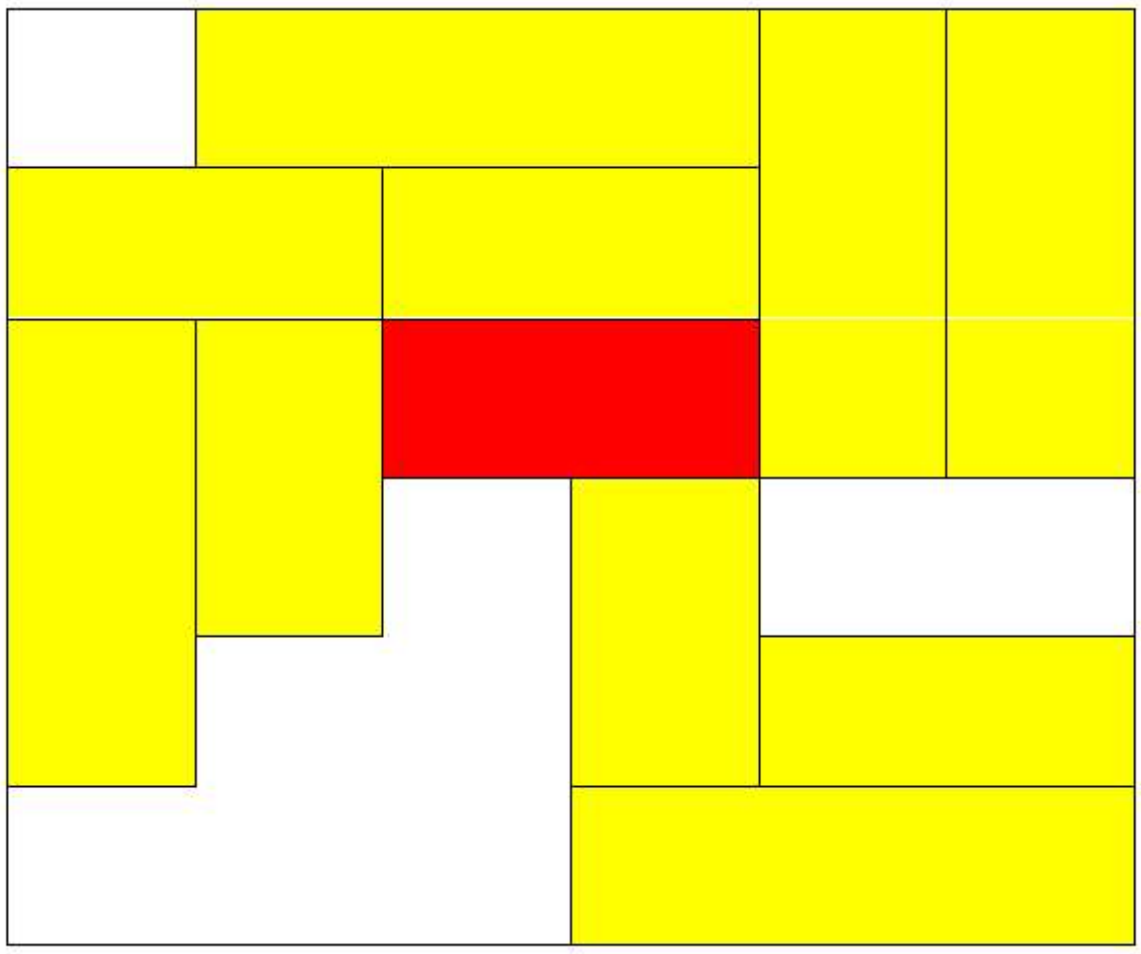
In [6]:
```
index = 1
MapTag = '55'
to_solve = read_input('/content/testcases/inp%s.txt' % MapTag)
print (to_solve[0])
```

```
[[ 0  9  9  9  5  4]
 [ 7  7  6  6  5  4]
 [ 3  1 11 11  5  4]
 [ 3  1  0  2  0  0]
 [ 3  0  0  2  8  8]
 [ 0  0  0 10 10 10]]
```

For reference and comparison, we've also included images of the map

In [7]:
```
from google.colab.patches import cv2_imshow
import cv2
img = cv2.imread('/content/testcases/map%s.PNG' % MapTag)
print ('This is the image of the map')
cv2_imshow(img)
```

```
This is the image of the map
```



Classification of maps:

- 1-7: Unsolvable instances
- 8-10: Already goal state

Please note that the above map instances are only used to test Breadth-first search
Our tests mainly focus on the following maps:

- 11-30: Simple test cases (which are requiring less than 6 moves to solve)
- 31-45: Hard test cases, in terms of looking for a move towards the solution. Their optimal solutions requires about 15 to 25 steps
- 46-55: These test cases were generated by generator. We'll put it in the appendix.

Uninformed search

Breadth-first search

Firstly, we define some functions that could help us perform Breadth- first search:

- `Goal_state` function returns True **whenever the prison block can escape**.
- `SuccGen` returns the nodes opened by **current state**, also it eliminates repeated nodes by using `AdjacentStates` dictionary.
- `Trace` helps returning step-by-step solution with a list named `path`.
- And finally `BFS` does breadth-first search.

In [8]:
```
def Goal_state(node):
    start = node.AllBlocks[-1].start_point_x
    if sum(node.GameMap.map[2, start: 6]) == node.AllBlocks[-1].indx * node.AllBlocks[-1].length:
        return True
```

```
    return False

def SuccGen(CurrState):
    ChildLst = []
    for move in CurrState.GetNextMoves():
        NextState = CurrState.NextStates(move)
        if NextState.GameMap.__str__() not in AdjacentStates:
            AdjacentStates[NextState.GameMap.__str__()] = CurrState.GameMap.__str__()
            ChildLst.append(NextState)
    return ChildLst

def Trace():
    CurrPos = FinishNode
    while CurrPos != InitState.GameMap.__str__():
        path.append(CurrPos)
        CurrPos = AdjacentStates[CurrPos]

def BFS(root):
    if Goal_state(root):
      return 'Already goal'
    else:
      Queue = SuccGen(root)
      while len(Queue) != 0:
          CurrNode = Queue.pop(0)
          if Goal_state(CurrNode):
              return CurrNode.GameMap.__str__()
          else:
              for ChildNode in SuccGen(CurrNode):
                  Queue.append(ChildNode)
      return 'Failure'
```

Map solver

Here we perform Breadth-first search on our game map.

In [9]:

```
index = 1
MapTag = '30'
to_solve = read_input('/content/testcases/inp%s.txt' % MapTag)
```

In [10]:

```
index = 1
All_Blocks = to_solve[1]
InitState = State(All_Blocks)



AdjacentStates = dict()
path = []
print(InitState.GameMap)
FinishNode = BFS(InitState)

if FinishNode == 'Already goal':
  print('Goal reached already')

elif FinishNode != 'Failure':
    Trace()

    print('Number of steps: %d' %(len(path)))

    for i in range(len(path) -1, -1, -1):
        print('Step %d' % (len(path) - i))
        print(path[i])
        print()
else:
    print('We have been stucked! No ways to escape!')
```

```
[[ 1  1  1  2  0  0]
 [ 3  0  0  2  4  4]
 [ 3  0 11 11  5  6]
 [ 0  0  0  0  5  6]
 [ 0  0  0  0  7  8]
 [ 9  9 10 10  7  8]]
Number of steps: 5
Step 1
[[ 1  1  1  2  0  0]
 [ 3  0  0  2  4  4]
 [ 3 11 11  0  5  6]
 [ 0  0  0  0  5  6]
 [ 0  0  0  0  7  8]
 [ 9  9 10 10  7  8]]

Step 2
[[ 1  1  1  0  0  0]
 [ 3  0  0  0  4  4]
 [ 3 11 11  0  5  6]
 [ 0  0  0  2  5  6]
 [ 0  0  0  2  7  8]
 [ 9  9 10 10  7  8]]

Step 3
[[ 1  1  1  0  0  0]
 [ 3  0  4  4  0  0]
 [ 3 11 11  0  5  6]
```

```
 [ 0  0  0  2  5  6]
 [ 0  0  0  2  7  8]
 [ 9  9 10 10  7  8]]

Step 4
[[ 1  1  1  0  5  0]
 [ 3  0  4  4  5  0]
 [ 3 11 11  0  0  6]
 [ 0  0  0  2  0  6]
 [ 0  0  0  2  7  8]
 [ 9  9 10 10  7  8]]

Step 5
[[ 1  1  1  0  5  6]
 [ 3  0  4  4  5  6]
 [ 3 11 11  0  0  0]
 [ 0  0  0  2  0  0]
 [ 0  0  0  2  7  8]
 [ 9  9 10 10  7  8]]
```

Testcases - Analysis

Here's some analysis of the testcases.

Note that map 1-7 are unsolvable testcases and 8-10 are at their goal states already, so we skipped them.

In [11]:

```
import time
for map_index in range (11,56):

    index = 1
    to_solve = read_input('/content/testcases/inp%02i.txt' % map_index)
    All_Blocks = to_solve[1]
    InitState = State(All_Blocks)
    AdjacentStates = dict()
    path = []

    start = time.time()

    FinishNode = BFS(InitState)
    Trace()

    end = time.time()

    print('Map: %i  Number of steps: %02d  Fringe nodes: %05d  Time: %s sec' %(map_index, len(path), len(AdjacentStates),
```

```
Map: 11  Number of steps: 02  Fringe nodes: 00052  Time: 0.04730 sec
Map: 12  Number of steps: 02  Fringe nodes: 00278  Time: 0.20319 sec
Map: 13  Number of steps: 02  Fringe nodes: 00343  Time: 0.28695 sec
Map: 14  Number of steps: 02  Fringe nodes: 00166  Time: 0.11670 sec
Map: 15  Number of steps: 02  Fringe nodes: 00079  Time: 0.04990 sec
Map: 16  Number of steps: 04  Fringe nodes: 00414  Time: 0.56198 sec
Map: 17  Number of steps: 04  Fringe nodes: 00242  Time: 0.35282 sec
Map: 18  Number of steps: 04  Fringe nodes: 00041  Time: 0.04313 sec
Map: 19  Number of steps: 04  Fringe nodes: 00124  Time: 0.13075 sec
Map: 20  Number of steps: 04  Fringe nodes: 00038  Time: 0.05976 sec
Map: 21  Number of steps: 04  Fringe nodes: 01955  Time: 5.21748 sec
Map: 22  Number of steps: 04  Fringe nodes: 00402  Time: 1.39788 sec
Map: 23  Number of steps: 04  Fringe nodes: 01261  Time: 1.78243 sec
Map: 24  Number of steps: 04  Fringe nodes: 00060  Time: 0.05233 sec
Map: 25  Number of steps: 04  Fringe nodes: 00045  Time: 0.03839 sec
Map: 26  Number of steps: 04  Fringe nodes: 00026  Time: 0.02199 sec
Map: 27  Number of steps: 06  Fringe nodes: 00129  Time: 0.22845 sec
Map: 28  Number of steps: 04  Fringe nodes: 00058  Time: 0.06750 sec
Map: 29  Number of steps: 06  Fringe nodes: 00134  Time: 0.15315 sec
Map: 30  Number of steps: 05  Fringe nodes: 01270  Time: 1.58408 sec
Map: 31  Number of steps: 15  Fringe nodes: 01820  Time: 3.23228 sec
Map: 32  Number of steps: 17  Fringe nodes: 00212  Time: 0.31324 sec
Map: 33  Number of steps: 14  Fringe nodes: 01588  Time: 2.61875 sec
Map: 34  Number of steps: 15  Fringe nodes: 02093  Time: 3.85826 sec
Map: 35  Number of steps: 17  Fringe nodes: 01647  Time: 3.73634 sec
Map: 36  Number of steps: 14  Fringe nodes: 00078  Time: 0.08789 sec
Map: 37  Number of steps: 14  Fringe nodes: 00828  Time: 1.34687 sec
Map: 38  Number of steps: 17  Fringe nodes: 05007  Time: 12.45066 sec
Map: 39  Number of steps: 18  Fringe nodes: 02729  Time: 5.42620 sec
Map: 40  Number of steps: 17  Fringe nodes: 06676  Time: 16.34544 sec
Map: 41  Number of steps: 20  Fringe nodes: 02167  Time: 4.50310 sec
Map: 42  Number of steps: 21  Fringe nodes: 01247  Time: 2.21646 sec
Map: 43  Number of steps: 21  Fringe nodes: 04868  Time: 11.07466 sec
Map: 44  Number of steps: 22  Fringe nodes: 40933  Time: 124.12419 sec
Map: 45  Number of steps: 20  Fringe nodes: 05993  Time: 14.13217 sec
Map: 46  Number of steps: 19  Fringe nodes: 02076  Time: 4.18629 sec
Map: 47  Number of steps: 08  Fringe nodes: 02050  Time: 2.82450 sec
Map: 48  Number of steps: 11  Fringe nodes: 01099  Time: 2.11932 sec
Map: 49  Number of steps: 18  Fringe nodes: 00782  Time: 1.25849 sec
Map: 50  Number of steps: 05  Fringe nodes: 03610  Time: 5.34319 sec
Map: 51  Number of steps: 05  Fringe nodes: 01541  Time: 2.05831 sec
Map: 52  Number of steps: 08  Fringe nodes: 00913  Time: 1.48515 sec
Map: 53  Number of steps: 09  Fringe nodes: 01862  Time: 4.29877 sec
Map: 54  Number of steps: 10  Fringe nodes: 01075  Time: 2.09361 sec
Map: 55  Number of steps: 10  Fringe nodes: 02886  Time: 6.01412 sec
```

As observed above, most of the time, we obtain quite a acceptable runtime, *except for map 44*

Iterative Deepening search

Not deleting repetitive nodes: Map solver (for map 11 to 30)

Here we define some functions, some are previously defined in the Breadth-first search section.

- `DLS` perform recursive depth-limited search for the node.
- Then `IDS` increases the maximum depth of `DLS` until we find the solution.

Note: because of high number of recursive call, we only consider

In [12]:
```
def Goal_state(node):
    start = node.AllBlocks[-1].start_point_x
    if sum(node.GameMap.map[2, start: 6]) == All_Blocks[-1].indx * All_Blocks[-1].length:
        return True
    return False

def SuccGen_dfs(CurrState):
    ChildLst_dfs = []
    for move in CurrState.GetNextMoves():
        NextState = CurrState.NextStates(move)
        ChildLst_dfs.append(NextState)
    return ChildLst_dfs

def DLS(root, max_level):
    path_ids.append(root.GameMap.__str__())
    if Goal_state(root):
        return path_ids
    if len(path_ids) == max_level:
        return False
    for ChildNode in SuccGen_dfs(root):
        if DLS(ChildNode, max_level):
            return path_ids
        path_ids.pop()
    return False

def IDS(root, depth = 1):
    global path_ids
    while not DLS (root, depth):
        depth+= 1
        path_ids = []
```

In [13]:
```
index = 1
MapTag = '28'
to_solve = read_input('/content/testcases/inp%s.txt' % MapTag)
```

In [14]:
```
path_ids = []
index = 1

All_Blocks = to_solve[1]
InitState = State(All_Blocks)
start = time.time()
print(InitState.GameMap)
IDS(InitState)
print ('Number of steps:',len(path_ids)-1)
for step in range(1, len (path_ids)):
    print()
    print('Step',step)
    print(path_ids[step])
end = time.time()
print ('Map:', MapTag, 'runtime:', '{:.5f}'.format(end-start))
```

```
[[ 1  2  2  0  0  0]
 [ 1  0  3  4  4  4]
 [12 12  3  0  5  6]
 [ 7  7  8  8  5  6]
 [ 9  0  0 11  5  0]
 [ 9 10 10 11  0  0]]
Number of steps: 4

Step 1
[[ 1  0  0  2  2  0]
 [ 1  0  3  4  4  4]
 [12 12  3  0  5  6]
 [ 7  7  8  8  5  6]
 [ 9  0  0 11  5  0]
 [ 9 10 10 11  0  0]]

Step 2
[[ 1  0  3  2  2  0]
 [ 1  0  3  4  4  4]
 [12 12  0  0  5  6]
 [ 7  7  8  8  5  6]
 [ 9  0  0 11  5  0]
 [ 9 10 10 11  0  0]]

Step 3
[[ 1  0  3  2  2  0]
 [ 1  0  3  4  4  4]
 [12 12  0  0  0  6]
```

```
 [ 7  7  8  8  5  6]
 [ 9  0  0 11  5  0]
 [ 9 10 10 11  5  0]]

Step 4
[[ 1  0  3  2  2  0]
 [ 1  0  3  4  4  4]
 [12 12  0  0  0  0]
 [ 7  7  8  8  5  6]
 [ 9  0  0 11  5  6]
 [ 9 10 10 11  5  0]]
Map: 28 runtime: 0.18547
```

Not deleting repetitive nodes: Analysis(for map 11 to 30)

In [15]:

```python
import time
for MapTag in range (11,31):
    path_ids = []
    index = 1
    to_solve = read_input('/content/testcases/inp%s.txt' % MapTag)
    All_Blocks = to_solve[1]
    InitState = State(All_Blocks)
    start = time.time()
    IDS(InitState)
    end = time.time()
    print ('Map:', MapTag, '     steps:', len(path_ids)-1, '     time:', '{:.5f}'.format(end-start), 'sec')
```

```
Map: 11      steps: 2      time: 0.01005 sec
Map: 12      steps: 2      time: 0.01969 sec
Map: 13      steps: 2      time: 0.02897 sec
Map: 14      steps: 2      time: 0.00949 sec
Map: 15      steps: 2      time: 0.00457 sec
Map: 16      steps: 4      time: 0.36602 sec
Map: 17      steps: 4      time: 0.51024 sec
Map: 18      steps: 4      time: 0.05166 sec
Map: 19      steps: 4      time: 0.23305 sec
Map: 20      steps: 4      time: 0.11334 sec
Map: 21      steps: 4      time: 1.65663 sec
Map: 22      steps: 4      time: 1.26392 sec
Map: 23      steps: 4      time: 1.94893 sec
Map: 24      steps: 4      time: 0.04594 sec
Map: 25      steps: 4      time: 0.01568 sec
Map: 26      steps: 4      time: 0.04071 sec
Map: 27      steps: 6      time: 0.76730 sec
Map: 28      steps: 4      time: 0.17701 sec
Map: 29      steps: 6      time: 0.54275 sec
Map: 30      steps: 5      time: 1.36986 sec
```

Deleting repetitive nodes: Map solver

Firstly, we define some functions that could help us perform Iterative deepening search:

- Once again, `goal_state` function returns True **whenever the prison block can escape**.
- `depth_limited_search` performs depth-first-search with limited depth and returning step-by-step solution with a list named `solution` when it finds the goal state.
- `iterative_deepening_search` performs `depth_limited_search` with increasing depth limit starting from 0 till it finds the goal state.
- `print_result` calls the `iterative_deepening_search` and print the solution to users, makes it easier to use.

Note: `iterative_deepening_search` can't check whether the solution exists or not also the function only expands unvisited nodes so that it can save a lot of time but can't find the optimal solution.

In [16]:

```python
def goal_state(node):
    start = node.AllBlocks[-1].start_point_x
    if sum(node.GameMap.map[2, start: 6]) == node.AllBlocks[-1].indx * node.AllBlocks[-1].length:
        return True
    return False

def depth_limited_search(m, max_depth, current_depth):
    if current_depth >= max_depth: return #if the current depth >= the depth we set then return(stop)
    global solution
    global check
    global visited
    global count
    if check: return #if found the solution then return(stop)
    visited.add(m.GameMap.__str__()) #add this board so we don't encounter it again
    boards = [m]
    for board in boards:
        #make a list of possible moves
        for move in m.GetNextMoves():
            d = m.NextStates(move)
            # print(move)
            if d.GameMap.__str__() not in visited:
                visited.add(d.GameMap.__str__()) #add this board so we don't encounter it again
                steps[current_depth].append((board.GameMap.__str__(), d.GameMap.__str__())) #save the board for tracking
                if goal_state(d): #if solution found
                    check = 1
                    count += 1
                    if count == 1:
                        solution.append(d.GameMap.__str__())
                        solution.append(board.GameMap.__str__())
```

```
                        temp = board.GameMap.__str__()
                        k = len(steps)
                        while k > 0: # tracking back
                            for board_step in steps[k - 1]:
                                if temp == board_step[1]:
                                    solution.append(board_step[0])
                                    index = steps[k - 1].index(board_step)
                                    temp = board_step[0]
                            k -= 1
                        return
                depth_limited_search(d, max_depth, current_depth + 1) # solution not found, call the function with this m

def iterative_deepening_search(m):
    i = 0
    global steps
    global visited
    global check
    global solution
    global count
    solution = []
    check = 0
    count = 0
    while not check:
        # print('max_depth', i)
        visited = set()
        check = 0
        steps = [[] for j in range(i)]
        str_steps = [[] for j in range(i)]
        depth_limited_search(m, i, 0)
        i += 1
```

```
def print_result(m):
    global solution
    start = time.time()
    iterative_deepening_search(m)
    initial_board = solution.pop()
    solution = solution[::-1]
    run_time = time.time() - start
```

In [17]:

```
index = 1
MapTag = '11'
to_solve = read_input('/content/testcases/inp%s.txt' % MapTag)
```

In [18]:

```
import time
AllBlocks = to_solve[1]
InitState = State(AllBlocks)
print(InitState.GameMap)

start = time.time()
print_result(InitState)
end = time.time()

for step in range(len(solution)):
    print()
    print ('Step ', step+1)
    print(solution[step])
print('\nMap:',MapTag, ' time:','{:.5f}'.format(end-start))
```

```
[[0 1 0 0 0 0]
 [0 1 0 0 3 0]
 [6 6 2 0 3 0]
 [0 0 2 0 0 0]
 [4 0 0 5 5 5]
 [4 0 0 0 0 0]]

Step  1
[[0 1 0 0 3 0]
 [0 1 0 0 3 0]
 [6 6 2 0 0 0]
 [0 0 2 0 0 0]
 [4 0 0 5 5 5]
 [4 0 0 0 0 0]]

Step  2
[[0 1 2 0 3 0]
 [0 1 2 0 3 0]
 [6 6 0 0 0 0]
 [0 0 0 0 0 0]
 [4 0 0 5 5 5]
 [4 0 0 0 0 0]]

Map: 11  time: 0.01852
```

Deleting repetitive nodes: Analysis

In [19]:

```
for MapTag in range (11, 31):
    global index
    index = 1
    to_solve = read_input('/content/testcases/inp%i.txt' % MapTag)
    All_Blocks = to_solve[1]
    InitState = State(All_Blocks)\
```

```
    start = time.time()
    print_result(InitState)
    end = time.time()
    print('Map: %i   Steps: %02d   Number of nodes: %05d   Time: %s sec' %(MapTag, len(solution), len(visited), '{:.5f}'.
```

```
Map: 11   Steps: 02   Number of nodes: 00016   Time: 0.02394 sec
Map: 12   Steps: 02   Number of nodes: 00031   Time: 0.03412 sec
Map: 13   Steps: 02   Number of nodes: 00065   Time: 0.05212 sec
Map: 14   Steps: 02   Number of nodes: 00028   Time: 0.02210 sec
Map: 15   Steps: 02   Number of nodes: 00014   Time: 0.01364 sec
Map: 16   Steps: 04   Number of nodes: 00044   Time: 0.18544 sec
Map: 17   Steps: 04   Number of nodes: 00100   Time: 0.13451 sec
Map: 18   Steps: 05   Number of nodes: 00021   Time: 0.05378 sec
Map: 19   Steps: 08   Number of nodes: 00026   Time: 0.23723 sec
Map: 20   Steps: 11   Number of nodes: 00036   Time: 0.23751 sec
Map: 21   Steps: 05   Number of nodes: 00574   Time: 0.95596 sec
Map: 22   Steps: 05   Number of nodes: 00141   Time: 0.29653 sec
Map: 23   Steps: 06   Number of nodes: 00335   Time: 0.73277 sec
Map: 24   Steps: 06   Number of nodes: 00026   Time: 0.06882 sec
Map: 25   Steps: 04   Number of nodes: 00015   Time: 0.01991 sec
Map: 26   Steps: 06   Number of nodes: 00015   Time: 0.04210 sec
Map: 27   Steps: 09   Number of nodes: 00095   Time: 0.26225 sec
Map: 28   Steps: 05   Number of nodes: 00029   Time: 0.08633 sec
Map: 29   Steps: 08   Number of nodes: 00032   Time: 0.22117 sec
Map: 30   Steps: 15   Number of nodes: 00522   Time: 2.33544 sec
```

Informed search

In this part, we use the structure of priority queue to derive the nodes with the best heuristic.

Thus, some classes need adjustment.

For example, class `State` now include the evaluation function **f(state)=g(state)+h(state)**

The priority queue structure is implemented in class `PriorityQueue`

In [20]:

```
import numpy as np
import copy

class Block:
    index = 0
    """
    x range from 0 to 5
    y range from 0 to 5
    """
    def __init__(self, x, y, direction, length):
        global index
        self.start_point_x = x
        self.start_point_y = y
        self.direction = direction
        self.length = length
        Block.index += 1  #when you initialize next item, its index will raise
        self.indx = Block.index


    def __str__(self):
        a = 'start_point_x = %d\
        start_point_y = %d\
        direction = %s\
        length = %d\
        index  = %d'\
            % (int(self.start_point_x),\
            int(self.start_point_y),\
            self.direction,\
            int(self.length),\
            int(self.indx))
        return a

    def move(self, step, direct):
        """
        :param step: int
        :param direct: -1/+1
        :return: new block
        """
        if self.direction == "h":
            self.start_point_x += step * direct
        if self.direction == "v":
            self.start_point_y += step * direct

    def restartTag():
        Block.index = 0
    def __eq__(self, other):
        return self.start_point_x == other.start_point_x and self.start_point_y == other.start_point_y

class State():
    def __init__(self, AllBlocks, InitCost = 0, Func = None):
        self.AllBlocks = AllBlocks
        self.GameMap = Map()
        for block in self.AllBlocks:
            self.GameMap.add_block(block)
        self.InitCost = InitCost
        self.Eval = None
        self.Func = Func
        self.GetEvaluation()
```

```
        Block.restartTag()

    def __lt__(self, other):
        return self.Eval < other.Eval
    def __gt__(self, other):
        return self.Eval > other.Eval
    def __eq__(self, other):
        return self.Eval == other.Eval

    def GetEvaluation(self):
        if self.Func != None:
            self.Eval = self.InitCost + self.Func(self)
        else:
            self.Eval = self.InitCost

    def GetMap(self):
        return self.GameMap

    def GetNextMoves(self):
        All_Moves_list = []
        a = self.GetMap()
        for blk in self.AllBlocks:
            All_Moves_list += a.possible_move(blk)
        return All_Moves_list

    def Display(self):
        return self.GameMap.__str__()

    def NextState(self, move):
        New = []
        for block in self.AllBlocks:
            TempBlock = copy.copy(block)
            if TempBlock.indx == move[2]:
                TempBlock.move(move[0], move[1])
            New.append(TempBlock)
        return State(New, self.InitCost + 1 , self.Func)

class PriorityQueue():
    def __init__(self, Lst):
        self.Lst = Lst
        if self.GetSize() > 1:
            self.BuildHeap()

    def __str__(self):
        return str(self.Lst)

    def GetSize(self):
        return len(self.Lst)

    def BuildHeap(self):
        for i in range((self.GetSize() // 2) - 1, -1):
            self.Heapify(i)

    def Swap(self, i, j):
        self.Lst[i], self.Lst[j] = self.Lst[j], self.Lst[i]

    def Heapify(self, i):
        l = i * 2 + 1
        r = i * 2 + 2
        MinId = i
        if r <= self.GetSize() - 1:
            if self.Lst[MinId] > self.Lst[r]:
                MinId = r
        if l <= self.GetSize() - 1:
            if self.Lst[MinId] > self.Lst[l]:
                MinId = l

        if MinId != i:
            self.Swap(i, MinId)
            self.Heapify(MinId)

    def RevHeapify(self, i):
        parent = i // 2
        if parent != i and self.Lst[parent] > self.Lst[i]:
            self.Swap(parent, i)
            self.RevHeapify(parent)

    def Minimum(self):
        if self.GetSize != 0:
            return self.Lst[0]
        else:
            return float('inf')

    def Enqueue(self, a):
        self.Lst.append(a)
        self.RevHeapify(self.GetSize() - 1)

    def Dequeue(self):
        self.Swap(0, self.GetSize() - 1)
        Max = self.Lst.pop()
        self.Heapify(0)
        return Max
```

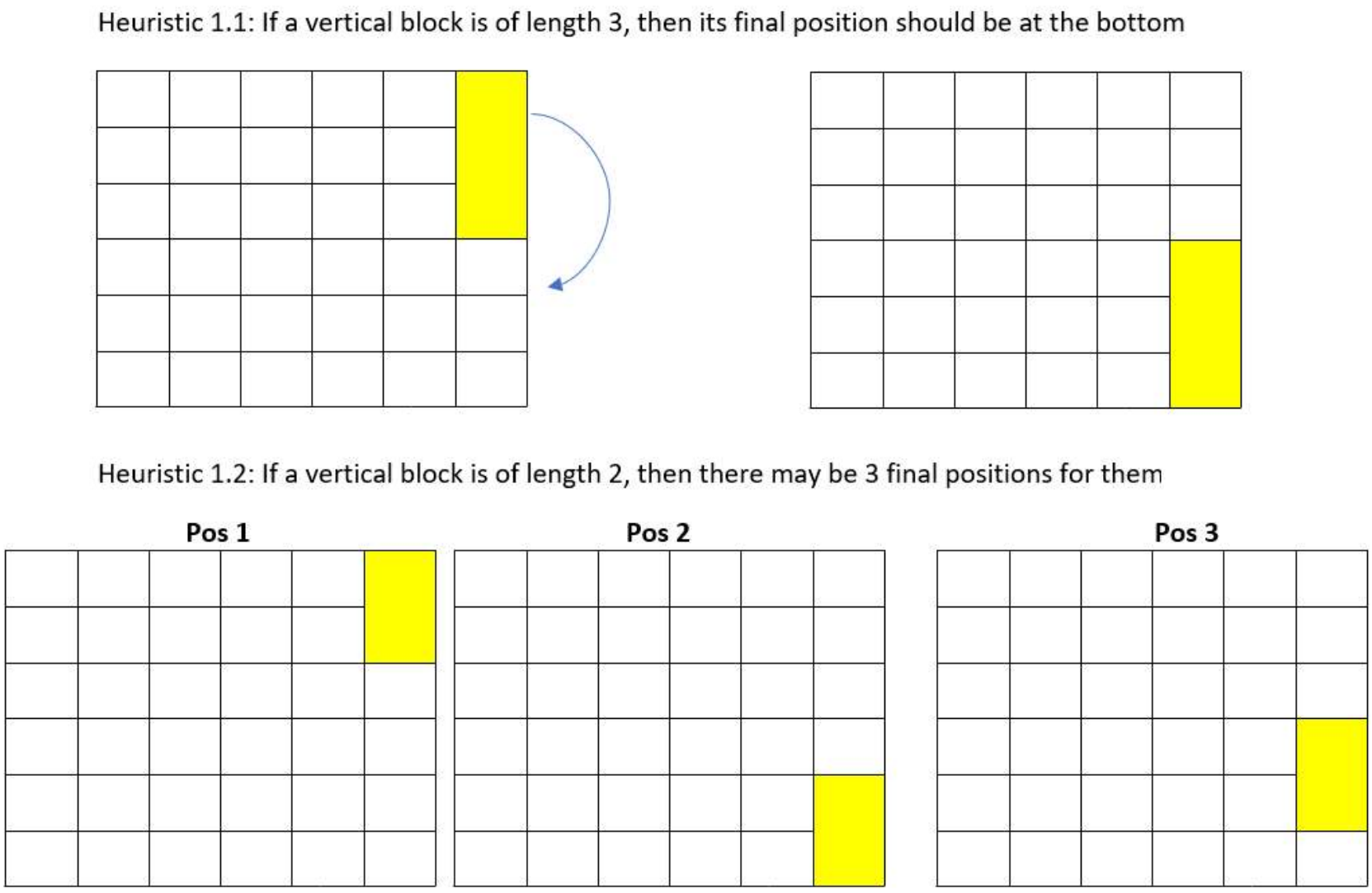
Heuristic functions

Heuristic 1

We call this heuristic 'blockage' heuristic. This heuristic is the one that most people who try to solve the board must have thought of.
It is based on the condition that vertical blocks should not block the way to the escape.

In [21]:
```
from google.colab.patches import cv2_imshow
import cv2
img = cv2.imread('/content/heuristic/heu1.PNG')
print ('This is the explanation of Heuristic 1')
cv2_imshow(img)
```

```
This is the explanation of Heuristic 1
```



Here's the implementation of Heuristic 1

In [22]:
```
def Heuristics1(State):
    count = 0
    for block in State.AllBlocks:
        if block.length == 3:
            if block.direction == 'v':
                if block.start_point_y != 3:
                    count += 1
        else:
            if block.direction == 'v':
                if block.start_point_y != 0 or block.start_point_y != 4 or block.start_point_y != 3:
                    count += 1
    return count
```

Heuristic 2

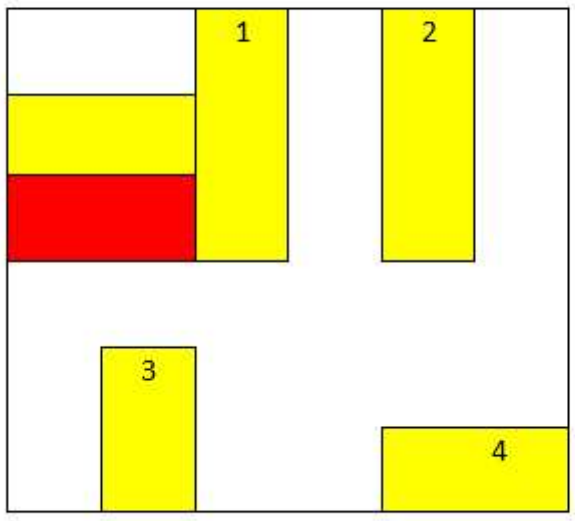
This heuristic is calculated based on the difference between the number of nodes from current state to goal state.
The goal state is generated from the result of breadth first search

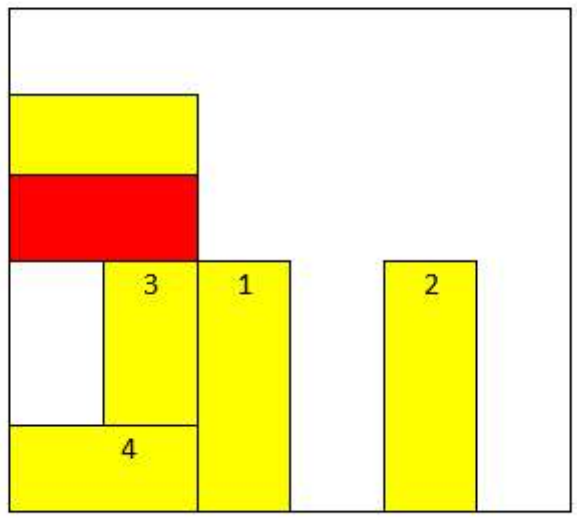
In [23]:
```
from google.colab.patches import cv2_imshow
import cv2
img = cv2.imread('/content/heuristic/heu2.PNG')
print ('This is the explanation of Heuristic 2')
cv2_imshow(img)
```

```
This is the explanation of Heuristic 2
```

Heuristic 2: Number of misplaced blocks, compared to goal state (generated by BFS)

Here f(state) = 4, as there are 4 blocks at the wrong position





Here's the implementation of Heuristic 2

In [24]:
```
def Heuristics2(State):
    global FinishAllBlocks
    count = 0
    for i in range(len(State.AllBlocks)):
        if not (State.AllBlocks[i] == FinishAllBlocks[i]):
            count += 1
    return count
```

Heuristic 3

This heuristic is calculated based on the Manhattan distance between the number of nodes from current state to goal state.
The goal state is also generated from the result of breadth first search.

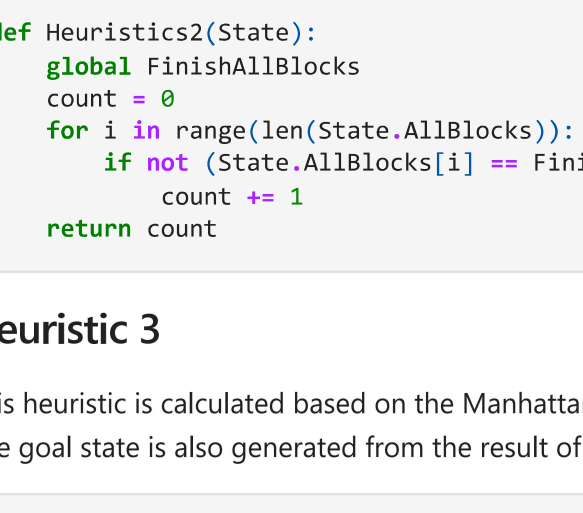
In [25]:
```
from google.colab.patches import cv2_imshow
import cv2
img = cv2.imread('/content/heuristic/heu3.PNG')
print ('This is the explanation of Heuristic 3')
cv2_imshow(img)
```

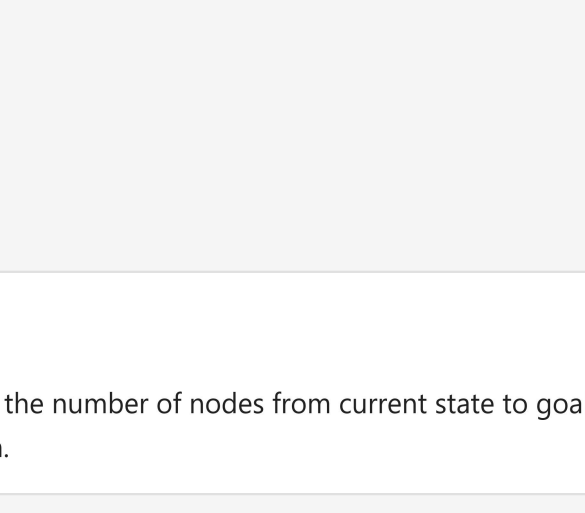
```
This is the explanation of Heuristic 3
```

Heuristic 3: Manhattan distance between current state and goal state (generated by BFS)

Here Manhattan (1) = 3, Manhattan (2) = 3, Manhattan (3) = 1, Manhattan (4) = 4

Therefore f = 3+3+1+4 = 11





Here's the implementation of Heuristic 3

In [26]:
```
def Heuristics3(State):
    global FinishAllBlocks
    sum = 0
    for i in range(len(State.AllBlocks)):
        sum += abs(State.AllBlocks[i].start_point_x - FinishAllBlocks[i].start_point_x)
        sum += abs(State.AllBlocks[i].start_point_y - FinishAllBlocks[i].start_point_y)
    return sum
```

A* search

Here we re-implement some functions to fit with the priority queue structure mentioned above

In [27]:
```
AdjacentStates = {}
def Goal(node):
    start = node.AllBlocks[-1].start_point_x
    if sum(node.GameMap.map[2, start: 6]) == InitState.AllBlocks[-1].indx * InitState.AllBlocks[-1].length:
        return True
    return False

def SuccGen(CurrState, PriQueue):
    for move in CurrState.GetNextMoves():
        NextState = CurrState.NextState(move)
        if NextState.GameMap.__str__() not in AdjacentStates:
            AdjacentStates[NextState.GameMap.__str__()] = CurrState.GameMap.__str__()
            PriQueue.Enqueue(NextState)

def Trace():
    CurrPos = FinishNode
    while CurrPos != InitState.GameMap.__str__():
        path.append(CurrPos)
        CurrPos = AdjacentStates[CurrPos]

def A_star_search(root):
    global Success
    PriQueue = PriorityQueue([])
    SuccGen(root, PriQueue)
    while PriQueue.GetSize() != 0:
        CurrNode = PriQueue.Dequeue()

        if Goal(CurrNode) == True:
            return CurrNode
        else:
            SuccGen(CurrNode, PriQueue)
    Success = False

def Trace():
    global path
    CurrPos = FinishNode.GameMap.__str__()
    while CurrPos != InitState.GameMap.__str__():
        path.append(CurrPos)
        CurrPos = AdjacentStates[CurrPos]
```

A* search: solver

A* search: solver - Heuristic 1

In [28]:
```
Tag = 40
_, All_Blocks = read_input('/content/testcases/inp%d.txt' % Tag)
_, FinishAllBlocks = read_input('/content/testcases/inp%d_2.txt' % Tag)
```

In [29]:
```
Func = Heuristics1
```

In [30]:
```
InitState = State(All_Blocks, 0, Func)
print(InitState.GameMap)
start = time.time()
path = []
AdjacentStates = {}
FinishNode = A_star_search(InitState)

Trace()
print('Number of steps: %d' %(len(path)))

for i in range(len(path) -1, -1, -1):
    print('Step %d' % (len(path) - i))
    print(path[i])
    print()

stop = time.time()
print('Time: ', stop - start)
print('Nodes: ',len(AdjacentStates))
```

```
[[ 0  1  2  2  2  3]
 [ 0  1  4  0  5  3]
 [12 12  4  6  5  7]
 [ 8  8  8  6  0  7]
 [ 9  0  0 10 10 10]
 [ 9 11 11  0  0  0]]
Number of steps: 17
Step 1
[[ 0  1  2  2  2  3]
 [ 0  1  4  0  5  3]
 [12 12  4  6  5  7]
 [ 8  8  8  6  0  7]
 [ 9 10 10 10  0  0]
 [ 9 11 11  0  0  0]]

Step 2
```

```
[[ 0  1  2  2  2  3]
 [ 0  1  4  0  5  3]
 [12 12  4  6  5  0]
 [ 8  8  8  6  0  0]
 [ 9 10 10 10  0  7]
 [ 9 11 11  0  0  7]]

Step 3
[[ 0  1  2  2  2  0]
 [ 0  1  4  0  5  3]
 [12 12  4  6  5  3]
 [ 8  8  8  6  0  0]
 [ 9 10 10 10  0  7]
 [ 9 11 11  0  0  7]]

Step 4
[[ 0  1  0  2  2  2]
 [ 0  1  4  0  5  3]
 [12 12  4  6  5  3]
 [ 8  8  8  6  0  0]
 [ 9 10 10 10  0  7]
 [ 9 11 11  0  0  7]]

Step 5
[[ 0  1  4  2  2  2]
 [ 0  1  4  0  5  3]
 [12 12  0  6  5  3]
 [ 8  8  8  6  0  0]
 [ 9 10 10 10  0  7]
 [ 9 11 11  0  0  7]]

Step 6
[[ 0  1  4  2  2  2]
 [ 0  1  4  6  5  3]
 [12 12  0  6  5  3]
 [ 8  8  8  0  0  0]
 [ 9 10 10 10  0  7]
 [ 9 11 11  0  0  7]]

Step 7
[[ 0  1  4  2  2  2]
 [ 0  1  4  6  5  3]
 [12 12  0  6  5  3]
 [ 0  8  8  8  0  0]
 [ 9 10 10 10  0  7]
 [ 9 11 11  0  0  7]]

Step 8
[[ 0  1  4  2  2  2]
 [ 0  1  4  6  5  3]
 [ 0 12 12  6  5  3]
 [ 0  8  8  8  0  0]
 [ 9 10 10 10  0  7]
 [ 9 11 11  0  0  7]]

Step 9
[[ 0  1  4  2  2  2]
 [ 9  1  4  6  5  3]
 [ 9 12 12  6  5  3]
 [ 0  8  8  8  0  0]
 [ 0 10 10 10  0  7]
 [ 0 11 11  0  0  7]]

Step 10
[[ 0  1  4  2  2  2]
 [ 9  1  4  6  5  3]
 [ 9 12 12  6  5  3]
 [ 8  8  8  0  0  0]
 [ 0 10 10 10  0  7]
 [ 0 11 11  0  0  7]]

Step 11
[[ 0  1  4  2  2  2]
 [ 9  1  4  6  0  3]
 [ 9 12 12  6  0  3]
 [ 8  8  8  0  5  0]
 [ 0 10 10 10  5  7]
 [ 0 11 11  0  0  7]]

Step 12
[[ 0  1  4  2  2  2]
 [ 9  1  4  6  0  3]
 [ 9 12 12  6  0  3]
 [ 8  8  8  0  5  0]
 [10 10 10  0  5  7]
 [ 0 11 11  0  0  7]]

Step 13
[[ 0  1  4  2  2  2]
 [ 9  1  4  0  0  3]
 [ 9 12 12  0  0  3]
 [ 8  8  8  6  5  0]
 [10 10 10  6  5  7]
 [ 0 11 11  0  0  7]]
```

```
Step 14
[[ 0  1  4  2  2  2]
 [ 9  1  4  0  0  3]
 [ 9  0  0 12 12  3]
 [ 8  8  8  6  5  0]
 [10 10 10  6  5  7]
 [ 0 11 11  0  0  7]]

Step 15
[[ 0  1  0  2  2  2]
 [ 9  1  4  0  0  3]
 [ 9  0  4 12 12  3]
 [ 8  8  8  6  5  0]
 [10 10 10  6  5  7]
 [ 0 11 11  0  0  7]]

Step 16
[[ 0  1  2  2  2  0]
 [ 9  1  4  0  0  3]
 [ 9  0  4 12 12  3]
 [ 8  8  8  6  5  0]
 [10 10 10  6  5  7]
 [ 0 11 11  0  0  7]]

Step 17
[[ 0  1  2  2  2  3]
 [ 9  1  4  0  0  3]
 [ 9  0  4 12 12  0]
 [ 8  8  8  6  5  0]
 [10 10 10  6  5  7]
 [ 0 11 11  0  0  7]]
Time:  17.35651183128357
Nodes:  6695
```

A* search: solver - Heuristic 2

In [31]:

```
Func = Heuristics2
```

In [32]:

```
InitState = State(All_Blocks, 0, Func)
print(InitState.GameMap)
start = time.time()
path = []
AdjacentStates = {}
FinishNode = A_star_search(InitState)

Trace()
print('Number of steps: %d' %(len(path)))

for i in range(len(path) -1, -1, -1):
    print('Step %d' % (len(path) - i))
    print(path[i])
    print()

stop = time.time()
print('Time: ', stop - start)
print('Nodes: ',len(AdjacentStates))
```

```
[[ 0  1  2  2  2  3]
 [ 0  1  4  0  5  3]
 [12 12  4  6  5  7]
 [ 8  8  8  6  0  7]
 [ 9  0  0 10 10 10]
 [ 9 11 11  0  0  0]]
Number of steps: 17
Step 1
[[ 0  1  2  2  2  3]
 [ 0  1  4  0  5  3]
 [12 12  4  6  5  7]
 [ 8  8  8  6  0  7]
 [ 9 10 10 10  0  0]
 [ 9 11 11  0  0  0]]

Step 2
[[ 0  1  2  2  2  3]
 [ 0  1  4  0  5  3]
 [12 12  4  6  5  0]
 [ 8  8  8  6  0  7]
 [ 9 10 10 10  0  7]
 [ 9 11 11  0  0  0]]

Step 3
[[ 0  1  2  2  2  3]
 [ 0  1  4  6  5  3]
 [12 12  4  6  5  0]
 [ 8  8  8  0  0  7]
 [ 9 10 10 10  0  7]
 [ 9 11 11  0  0  0]]

Step 4
[[ 0  1  2  2  2  0]
 [ 0  1  4  6  5  3]
 [12 12  4  6  5  3]
```

```
 [ 8  8  8  0  0  7]
 [ 9 10 10 10  0  7]
 [ 9 11 11  0  0  0]]

Step 5
[[ 0  1  0  2  2  2]
 [ 0  1  4  6  5  3]
 [12 12  4  6  5  3]
 [ 8  8  8  0  0  7]
 [ 9 10 10 10  0  7]
 [ 9 11 11  0  0  0]]

Step 6
[[ 0  1  4  2  2  2]
 [ 0  1  4  6  5  3]
 [12 12  0  6  5  3]
 [ 8  8  8  0  0  7]
 [ 9 10 10 10  0  7]
 [ 9 11 11  0  0  0]]

Step 7
[[ 0  1  4  2  2  2]
 [ 0  1  4  6  5  3]
 [ 0 12 12  6  5  3]
 [ 8  8  8  0  0  7]
 [ 9 10 10 10  0  7]
 [ 9 11 11  0  0  0]]

Step 8
[[ 0  1  4  2  2  2]
 [ 0  1  4  6  5  3]
 [ 0 12 12  6  5  3]
 [ 0  0  8  8  8  7]
 [ 9 10 10 10  0  7]
 [ 9 11 11  0  0  0]]

Step 9
[[ 0  1  4  2  2  2]
 [ 9  1  4  6  5  3]
 [ 9 12 12  6  5  3]
 [ 0  0  8  8  8  7]
 [ 0 10 10 10  0  7]
 [ 0 11 11  0  0  0]]

Step 10
[[ 0  1  4  2  2  2]
 [ 9  1  4  6  5  3]
 [ 9 12 12  6  5  3]
 [ 8  8  8  0  0  7]
 [ 0 10 10 10  0  7]
 [ 0 11 11  0  0  0]]

Step 11
[[ 0  1  4  2  2  2]
 [ 9  1  4  6  0  3]
 [ 9 12 12  6  0  3]
 [ 8  8  8  0  5  7]
 [ 0 10 10 10  5  7]
 [ 0 11 11  0  0  0]]

Step 12
[[ 0  1  4  2  2  2]
 [ 9  1  4  6  0  3]
 [ 9 12 12  6  0  3]
 [ 8  8  8  0  5  7]
 [10 10 10  0  5  7]
 [ 0 11 11  0  0  0]]

Step 13
[[ 0  1  4  2  2  2]
 [ 9  1  4  0  0  3]
 [ 9 12 12  0  0  3]
 [ 8  8  8  6  5  7]
 [10 10 10  6  5  7]
 [ 0 11 11  0  0  0]]

Step 14
[[ 0  1  4  2  2  2]
 [ 9  1  4  0  0  3]
 [ 9  0  0 12 12  3]
 [ 8  8  8  6  5  7]
 [10 10 10  6  5  7]
 [ 0 11 11  0  0  0]]

Step 15
[[ 0  1  0  2  2  2]
 [ 9  1  4  0  0  3]
 [ 9  0  4 12 12  3]
 [ 8  8  8  6  5  7]
 [10 10 10  6  5  7]
 [ 0 11 11  0  0  0]]

Step 16
[[ 0  1  2  2  2  0]
 [ 9  1  4  0  0  3]
```

```
 [ 9  0  4 12 12  3]
 [ 8  8  8  6  5  7]
 [10 10 10  6  5  7]
 [ 0 11 11  0  0  0]]

Step 17
[[ 0  1  2  2  2  3]
 [ 9  1  4  0  0  3]
 [ 9  0  4 12 12  0]
 [ 8  8  8  6  5  7]
 [10 10 10  6  5  7]
 [ 0 11 11  0  0  0]]

Time:  3.3291056156158447
Nodes:  1643
```

A* search: solver - Heuristic 3

In [33]:

```
Func = Heuristics3
```

In [34]:

```
InitState = State(All_Blocks, 0, Func)
print(InitState.GameMap)
start = time.time()
path = []
AdjacentStates = {}
FinishNode = A_star_search(InitState)

Trace()
print('Number of steps: %d' %(len(path)))

for i in range(len(path) -1, -1, -1):
    print('Step %d' % (len(path) - i))
    print(path[i])
    print()

stop = time.time()
print('Time: ', stop - start)
print('Nodes: ',len(AdjacentStates))
```

```
[[ 0  1  2  2  2  3]
 [ 0  1  4  0  5  3]
 [12 12  4  6  5  7]
 [ 8  8  8  6  0  7]
 [ 9  0  0 10 10 10]
 [ 9 11 11  0  0  0]]
Number of steps: 18
Step 1
[[ 0  1  2  2  2  3]
 [ 0  1  4  0  5  3]
 [12 12  4  6  5  7]
 [ 8  8  8  6  0  7]
 [ 9 10 10 10  0  0]
 [ 9 11 11  0  0  0]]

Step 2
[[ 0  1  2  2  2  3]
 [ 0  1  4  0  0  3]
 [12 12  4  6  0  7]
 [ 8  8  8  6  5  7]
 [ 9 10 10 10  5  0]
 [ 9 11 11  0  0  0]]

Step 3
[[ 0  1  2  2  2  3]
 [ 0  1  4  0  0  3]
 [12 12  4  6  0  0]
 [ 8  8  8  6  5  7]
 [ 9 10 10 10  5  7]
 [ 9 11 11  0  0  0]]

Step 4
[[ 0  1  2  2  2  3]
 [ 0  1  4  6  0  3]
 [12 12  4  6  0  0]
 [ 8  8  8  0  5  7]
 [ 9 10 10 10  5  7]
 [ 9 11 11  0  0  0]]

Step 5
[[ 0  1  2  2  2  3]
 [ 0  1  4  6  0  3]
 [12 12  4  6  0  0]
 [ 0  8  8  8  5  7]
 [ 9 10 10 10  5  7]
 [ 9 11 11  0  0  0]]

Step 6
[[ 0  1  2  2  2  3]
 [ 0  1  4  6  0  3]
 [12 12  4  6  0  0]
 [ 9  8  8  8  5  7]
 [ 9 10 10 10  5  7]
 [ 0 11 11  0  0  0]]
```

```
Step 7
[[ 0  1  2  2  2  0]
 [ 0  1  4  6  0  3]
 [12 12  4  6  0  3]
 [ 9  8  8  8  5  7]
 [ 9 10 10 10  5  7]
 [ 0 11 11  0  0  0]]

Step 8
[[ 0  1  0  2  2  2]
 [ 0  1  4  6  0  3]
 [12 12  4  6  0  3]
 [ 9  8  8  8  5  7]
 [ 9 10 10 10  5  7]
 [ 0 11 11  0  0  0]]

Step 9
[[ 0  1  4  2  2  2]
 [ 0  1  4  6  0  3]
 [12 12  0  6  0  3]
 [ 9  8  8  8  5  7]
 [ 9 10 10 10  5  7]
 [ 0 11 11  0  0  0]]

Step 10
[[ 0  1  4  2  2  2]
 [ 0  1  4  6  0  3]
 [ 0 12 12  6  0  3]
 [ 9  8  8  8  5  7]
 [ 9 10 10 10  5  7]
 [ 0 11 11  0  0  0]]

Step 11
[[ 0  1  4  2  2  2]
 [ 9  1  4  6  0  3]
 [ 9 12 12  6  0  3]
 [ 0  8  8  8  5  7]
 [ 0 10 10 10  5  7]
 [ 0 11 11  0  0  0]]

Step 12
[[ 0  1  4  2  2  2]
 [ 9  1  4  6  0  3]
 [ 9 12 12  6  0  3]
 [ 8  8  8  0  5  7]
 [ 0 10 10 10  5  7]
 [ 0 11 11  0  0  0]]

Step 13
[[ 0  1  4  2  2  2]
 [ 9  1  4  6  0  3]
 [ 9 12 12  6  0  3]
 [ 8  8  8  0  5  7]
 [10 10 10  0  5  7]
 [ 0 11 11  0  0  0]]

Step 14
[[ 0  1  4  2  2  2]
 [ 9  1  4  0  0  3]
 [ 9 12 12  0  0  3]
 [ 8  8  8  6  5  7]
 [10 10 10  6  5  7]
 [ 0 11 11  0  0  0]]

Step 15
[[ 0  1  4  2  2  2]
 [ 9  1  4  0  0  3]
 [ 9  0  0 12 12  3]
 [ 8  8  8  6  5  7]
 [10 10 10  6  5  7]
 [ 0 11 11  0  0  0]]

Step 16
[[ 0  1  0  2  2  2]
 [ 9  1  4  0  0  3]
 [ 9  0  4 12 12  3]
 [ 8  8  8  6  5  7]
 [10 10 10  6  5  7]
 [ 0 11 11  0  0  0]]

Step 17
[[ 0  1  2  2  2  0]
 [ 9  1  4  0  0  3]
 [ 9  0  4 12 12  3]
 [ 8  8  8  6  5  7]
 [10 10 10  6  5  7]
 [ 0 11 11  0  0  0]]

Step 18
[[ 0  1  2  2  2  3]
 [ 9  1  4  0  0  3]
 [ 9  0  4 12 12  0]
 [ 8  8  8  6  5  7]
 [10 10 10  6  5  7]
```

```
 [ 0 11 11  0  0  0]]

Time:  1.763026475906372
Nodes:  1100
```

A* search: Analysis

A* search: Analysis - Heuristic 1

In [35]:
```
Func = Heuristics1
```

In [36]:
```
import time
for Tag in range(11, 56):
    _, All_Blocks = read_input('testcases/inp%d.txt' % Tag)
    _, FinishAllBlocks = read_input('testcases/inp%d_2.txt' % Tag)
    AdjacentStates = {}
    path = []
    InitState = State(All_Blocks, 0, Func)

    #print(InitState.GameMap)
    start = time.time()

    Success = True
    FinishState = State(FinishAllBlocks)
    FinishNode = A_star_search(InitState)
    Trace()
    stop = time.time()
    print('Map: %i  Number of steps: %02d  Fringe nodes: %05d   Time: %s sec' %(Tag, len(path), len(AdjacentStates), '{:.
```

```
Map: 11  Number of steps: 02  Fringe nodes: 00092   Time: 0.05739 sec
Map: 12  Number of steps: 02  Fringe nodes: 00148   Time: 0.09138 sec
Map: 13  Number of steps: 02  Fringe nodes: 00091   Time: 0.05232 sec
Map: 14  Number of steps: 02  Fringe nodes: 00059   Time: 0.02714 sec
Map: 15  Number of steps: 02  Fringe nodes: 00149   Time: 0.10314 sec
Map: 16  Number of steps: 04  Fringe nodes: 00417   Time: 0.55906 sec
Map: 17  Number of steps: 04  Fringe nodes: 00202   Time: 0.27068 sec
Map: 18  Number of steps: 04  Fringe nodes: 00045   Time: 0.07426 sec
Map: 19  Number of steps: 05  Fringe nodes: 00104   Time: 0.10150 sec
Map: 20  Number of steps: 04  Fringe nodes: 00035   Time: 0.03102 sec
Map: 21  Number of steps: 04  Fringe nodes: 01089   Time: 1.12000 sec
Map: 22  Number of steps: 04  Fringe nodes: 00052   Time: 0.02746 sec
Map: 23  Number of steps: 04  Fringe nodes: 00340   Time: 0.28822 sec
Map: 24  Number of steps: 04  Fringe nodes: 00075   Time: 0.05554 sec
Map: 25  Number of steps: 04  Fringe nodes: 00049   Time: 0.04217 sec
Map: 26  Number of steps: 04  Fringe nodes: 00020   Time: 0.01715 sec
Map: 27  Number of steps: 06  Fringe nodes: 00064   Time: 0.05770 sec
Map: 28  Number of steps: 04  Fringe nodes: 00064   Time: 0.09197 sec
Map: 29  Number of steps: 06  Fringe nodes: 00158   Time: 0.17975 sec
Map: 30  Number of steps: 05  Fringe nodes: 00643   Time: 0.69569 sec
Map: 31  Number of steps: 15  Fringe nodes: 01282   Time: 2.24070 sec
Map: 32  Number of steps: 17  Fringe nodes: 00185   Time: 0.28264 sec
Map: 33  Number of steps: 14  Fringe nodes: 01400   Time: 2.36255 sec
Map: 34  Number of steps: 15  Fringe nodes: 01741   Time: 3.27891 sec
Map: 35  Number of steps: 17  Fringe nodes: 01586   Time: 3.48920 sec
Map: 36  Number of steps: 14  Fringe nodes: 00076   Time: 0.08813 sec
Map: 37  Number of steps: 14  Fringe nodes: 00791   Time: 1.31379 sec
Map: 38  Number of steps: 17  Fringe nodes: 04760   Time: 12.19895 sec
Map: 39  Number of steps: 18  Fringe nodes: 02335   Time: 4.89659 sec
Map: 40  Number of steps: 17  Fringe nodes: 06695   Time: 17.10553 sec
Map: 41  Number of steps: 20  Fringe nodes: 01897   Time: 3.78470 sec
Map: 42  Number of steps: 22  Fringe nodes: 01003   Time: 1.69284 sec
Map: 43  Number of steps: 21  Fringe nodes: 04770   Time: 11.04176 sec
Map: 44  Number of steps: 22  Fringe nodes: 40265   Time: 128.66335 sec
Map: 45  Number of steps: 20  Fringe nodes: 05911   Time: 14.76404 sec
Map: 46  Number of steps: 21  Fringe nodes: 01928   Time: 3.96455 sec
Map: 47  Number of steps: 08  Fringe nodes: 00975   Time: 1.12298 sec
Map: 48  Number of steps: 11  Fringe nodes: 00805   Time: 1.56603 sec
Map: 49  Number of steps: 18  Fringe nodes: 00556   Time: 0.82087 sec
Map: 50  Number of steps: 05  Fringe nodes: 01029   Time: 1.12812 sec
Map: 51  Number of steps: 05  Fringe nodes: 00503   Time: 0.44124 sec
Map: 52  Number of steps: 09  Fringe nodes: 01035   Time: 1.65629 sec
Map: 53  Number of steps: 09  Fringe nodes: 01826   Time: 4.08632 sec
Map: 54  Number of steps: 10  Fringe nodes: 00876   Time: 1.62109 sec
Map: 55  Number of steps: 10  Fringe nodes: 02240   Time: 4.44874 sec
```

A* search: Analysis - Heuristic 2

In [37]:
```
Func = Heuristics2
```

In [38]:
```
import time
for Tag in range(11, 56):
    _, All_Blocks = read_input('testcases/inp%d.txt' % Tag)
    _, FinishAllBlocks = read_input('testcases/inp%d_2.txt' % Tag)
    AdjacentStates = {}
    path = []
    InitState = State(All_Blocks, 0, Func)

    #print(InitState.GameMap)
    start = time.time()

    Success = True
```

```
    FinishState = State(FinishAllBlocks)
    FinishNode = A_star_search(InitState)
    Trace()
    stop = time.time()
    print('Map: %i  Number of steps: %02d  Fringe nodes: %05d   Time: %s sec' %(Tag, len(path), len(AdjacentStates), '{:.
```

```
Map: 11  Number of steps: 02  Fringe nodes: 00023   Time: 0.01196 sec
Map: 12  Number of steps: 02  Fringe nodes: 00036   Time: 0.01693 sec
Map: 13  Number of steps: 02  Fringe nodes: 00025   Time: 0.01173 sec
Map: 14  Number of steps: 02  Fringe nodes: 00036   Time: 0.01608 sec
Map: 15  Number of steps: 02  Fringe nodes: 00022   Time: 0.01006 sec
Map: 16  Number of steps: 04  Fringe nodes: 00108   Time: 0.06711 sec
Map: 17  Number of steps: 04  Fringe nodes: 00032   Time: 0.01589 sec
Map: 18  Number of steps: 04  Fringe nodes: 00015   Time: 0.00866 sec
Map: 19  Number of steps: 04  Fringe nodes: 00023   Time: 0.01486 sec
Map: 20  Number of steps: 04  Fringe nodes: 00019   Time: 0.01519 sec
Map: 21  Number of steps: 04  Fringe nodes: 00074   Time: 0.04677 sec
Map: 22  Number of steps: 04  Fringe nodes: 00061   Time: 0.03567 sec
Map: 23  Number of steps: 04  Fringe nodes: 00063   Time: 0.03254 sec
Map: 24  Number of steps: 04  Fringe nodes: 00031   Time: 0.01770 sec
Map: 25  Number of steps: 04  Fringe nodes: 00011   Time: 0.01012 sec
Map: 26  Number of steps: 04  Fringe nodes: 00014   Time: 0.00791 sec
Map: 27  Number of steps: 06  Fringe nodes: 00048   Time: 0.03041 sec
Map: 28  Number of steps: 04  Fringe nodes: 00036   Time: 0.03445 sec
Map: 29  Number of steps: 06  Fringe nodes: 00056   Time: 0.05283 sec
Map: 30  Number of steps: 05  Fringe nodes: 00051   Time: 0.02631 sec
Map: 31  Number of steps: 15  Fringe nodes: 00899   Time: 1.71691 sec
Map: 32  Number of steps: 17  Fringe nodes: 00160   Time: 0.22277 sec
Map: 33  Number of steps: 14  Fringe nodes: 00855   Time: 1.41484 sec
Map: 34  Number of steps: 15  Fringe nodes: 01012   Time: 2.07077 sec
Map: 35  Number of steps: 17  Fringe nodes: 00416   Time: 0.44537 sec
Map: 36  Number of steps: 15  Fringe nodes: 00061   Time: 0.05558 sec
Map: 37  Number of steps: 15  Fringe nodes: 00539   Time: 0.84109 sec
Map: 38  Number of steps: 17  Fringe nodes: 04097   Time: 9.80546 sec
Map: 39  Number of steps: 18  Fringe nodes: 01858   Time: 4.15413 sec
Map: 40  Number of steps: 17  Fringe nodes: 01643   Time: 3.22630 sec
Map: 41  Number of steps: 21  Fringe nodes: 00269   Time: 0.35626 sec
Map: 42  Number of steps: 21  Fringe nodes: 00712   Time: 1.16333 sec
Map: 43  Number of steps: 22  Fringe nodes: 03865   Time: 8.11796 sec
Map: 44  Number of steps: 22  Fringe nodes: 25597   Time: 58.62335 sec
Map: 45  Number of steps: 20  Fringe nodes: 04262   Time: 7.49878 sec
Map: 46  Number of steps: 20  Fringe nodes: 01237   Time: 2.28408 sec
Map: 47  Number of steps: 08  Fringe nodes: 00053   Time: 0.04259 sec
Map: 48  Number of steps: 11  Fringe nodes: 00242   Time: 0.17966 sec
Map: 49  Number of steps: 19  Fringe nodes: 00422   Time: 0.63410 sec
Map: 50  Number of steps: 05  Fringe nodes: 00087   Time: 0.05499 sec
Map: 51  Number of steps: 05  Fringe nodes: 00091   Time: 0.06621 sec
Map: 52  Number of steps: 08  Fringe nodes: 00115   Time: 0.09082 sec
Map: 53  Number of steps: 09  Fringe nodes: 00111   Time: 0.06481 sec
Map: 54  Number of steps: 10  Fringe nodes: 00193   Time: 0.14392 sec
Map: 55  Number of steps: 10  Fringe nodes: 01007   Time: 1.45873 sec
```

A* search: Analysis - Heuristic 3

In [39]:
```
Func = Heuristics3
```

In [40]:
```
import time
for Tag in range(11, 56):
    _, All_Blocks = read_input('testcases/inp%d.txt' % Tag)
    _, FinishAllBlocks = read_input('testcases/inp%d_2.txt' % Tag)
    AdjacentStates = {}
    path = []
    InitState = State(All_Blocks, 0, Func)

    #print(InitState.GameMap)
    start = time.time()

    Success = True
    FinishState = State(FinishAllBlocks)
    FinishNode = A_star_search(InitState)
    Trace()
    stop = time.time()
    print('Map: %i  Number of steps: %02d  Fringe nodes: %05d   Time: %s sec' %(Tag, len(path), len(AdjacentStates), '{:.
```

```
Map: 11  Number of steps: 02  Fringe nodes: 00017   Time: 0.01459 sec
Map: 12  Number of steps: 02  Fringe nodes: 00036   Time: 0.01642 sec
Map: 13  Number of steps: 02  Fringe nodes: 00025   Time: 0.01135 sec
Map: 14  Number of steps: 02  Fringe nodes: 00036   Time: 0.01780 sec
Map: 15  Number of steps: 02  Fringe nodes: 00022   Time: 0.01041 sec
Map: 16  Number of steps: 04  Fringe nodes: 00107   Time: 0.06661 sec
Map: 17  Number of steps: 04  Fringe nodes: 00032   Time: 0.01615 sec
Map: 18  Number of steps: 04  Fringe nodes: 00015   Time: 0.01130 sec
Map: 19  Number of steps: 04  Fringe nodes: 00023   Time: 0.01624 sec
Map: 20  Number of steps: 04  Fringe nodes: 00025   Time: 0.01944 sec
Map: 21  Number of steps: 05  Fringe nodes: 00101   Time: 0.06648 sec
Map: 22  Number of steps: 04  Fringe nodes: 00037   Time: 0.02832 sec
Map: 23  Number of steps: 04  Fringe nodes: 00057   Time: 0.04231 sec
Map: 24  Number of steps: 04  Fringe nodes: 00026   Time: 0.01601 sec
Map: 25  Number of steps: 04  Fringe nodes: 00011   Time: 0.00877 sec
Map: 26  Number of steps: 04  Fringe nodes: 00014   Time: 0.01352 sec
Map: 27  Number of steps: 07  Fringe nodes: 00035   Time: 0.02179 sec
Map: 28  Number of steps: 04  Fringe nodes: 00041   Time: 0.03175 sec
Map: 29  Number of steps: 06  Fringe nodes: 00056   Time: 0.04600 sec
```

```
Map: 30  Number of steps: 05  Fringe nodes: 00040   Time: 0.02402 sec
Map: 31  Number of steps: 18  Fringe nodes: 00963   Time: 1.85735 sec
Map: 32  Number of steps: 18  Fringe nodes: 00157   Time: 0.22366 sec
Map: 33  Number of steps: 15  Fringe nodes: 00620   Time: 0.79990 sec
Map: 34  Number of steps: 17  Fringe nodes: 00904   Time: 1.65638 sec
Map: 35  Number of steps: 21  Fringe nodes: 00285   Time: 0.35498 sec
Map: 36  Number of steps: 16  Fringe nodes: 00062   Time: 0.05535 sec
Map: 37  Number of steps: 16  Fringe nodes: 00458   Time: 0.60871 sec
Map: 38  Number of steps: 19  Fringe nodes: 01338   Time: 1.57712 sec
Map: 39  Number of steps: 22  Fringe nodes: 01563   Time: 2.48185 sec
Map: 40  Number of steps: 18  Fringe nodes: 01100   Time: 1.66645 sec
Map: 41  Number of steps: 23  Fringe nodes: 00271   Time: 0.35753 sec
Map: 42  Number of steps: 21  Fringe nodes: 00684   Time: 1.14811 sec
Map: 43  Number of steps: 24  Fringe nodes: 00619   Time: 0.61713 sec
Map: 44  Number of steps: 22  Fringe nodes: 07102   Time: 11.73359 sec
Map: 45  Number of steps: 23  Fringe nodes: 02400   Time: 4.08439 sec
Map: 46  Number of steps: 24  Fringe nodes: 01055   Time: 1.77217 sec
Map: 47  Number of steps: 08  Fringe nodes: 00046   Time: 0.03038 sec
Map: 48  Number of steps: 11  Fringe nodes: 00097   Time: 0.05436 sec
Map: 49  Number of steps: 21  Fringe nodes: 00389   Time: 0.59733 sec
Map: 50  Number of steps: 06  Fringe nodes: 00129   Time: 0.07102 sec
Map: 51  Number of steps: 05  Fringe nodes: 00063   Time: 0.03443 sec
Map: 52  Number of steps: 08  Fringe nodes: 00104   Time: 0.07669 sec
Map: 53  Number of steps: 10  Fringe nodes: 00167   Time: 0.10164 sec
Map: 54  Number of steps: 10  Fringe nodes: 00081   Time: 0.04318 sec
Map: 55  Number of steps: 11  Fringe nodes: 01307   Time: 1.71103 sec
```

A* search: Analysis - Heuristic 3.5 - Math form

Firstly we need to reinitialize `State` class

In [41]:

```python
class State():
    def __init__(self, AllBlocks, InitCost = 0, Func = None):
        self.AllBlocks = AllBlocks
        self.GameMap = Map()
        for block in self.AllBlocks:
            self.GameMap.add_block(block)
        self.InitCost = InitCost
        self.Eval = None
        self.Func = Func
        self.GetEvaluation()

        Block.restartTag()

    def __lt__(self, other):
        return self.Eval < other.Eval
    def __gt__(self, other):
        return self.Eval > other.Eval
    def __eq__(self, other):
        return self.Eval == other.Eval

    def GetEvaluation(self):
        if self.Func != None:
            self.Eval = self.InitCost + self.Func(self)
        else:
            self.Eval = self.InitCost

    def GetMap(self):
        return self.GameMap

    def GetNextMoves(self):
        All_Moves_list = []
        a = self.GetMap()
        for blk in self.AllBlocks:
            All_Moves_list += a.possible_move(blk)
        return All_Moves_list

    def Display(self):
        return self.GameMap.__str__()

    def NextState(self, move):
        New = []
        for block in self.AllBlocks:
            TempBlock = copy.copy(block)
            if TempBlock.indx == move[2]:
                TempBlock.move(move[0], move[1])
            New.append(TempBlock)
        return State(New, self.InitCost + move[0] , self.Func)
```

Now there's some analysis

In [42]:

```python
Func = Heuristics3
```

In [43]:

```python
import time
for Tag in range(11, 56):
    _, All_Blocks = read_input('testcases/inp%d.txt' % Tag)
    _, FinishAllBlocks = read_input('testcases/inp%d_2.txt' % Tag)
    AdjacentStates = {}
    path = []
    InitState = State(All_Blocks, 0, Func)

    #print(InitState.GameMap)
```

```
    start = time.time()

    Success = True
    FinishState = State(FinishAllBlocks)
    FinishNode = A_star_search(InitState)
    Trace()
    stop = time.time()
    print('Map: %i  Number of steps: %02d  Fringe nodes: %05d   Time: %s sec' %(Tag, len(path), len(AdjacentStates), '{:.
```

```
Map: 11  Number of steps: 02  Fringe nodes: 00030   Time: 0.02005 sec
Map: 12  Number of steps: 02  Fringe nodes: 00036   Time: 0.01723 sec
Map: 13  Number of steps: 02  Fringe nodes: 00025   Time: 0.01121 sec
Map: 14  Number of steps: 02  Fringe nodes: 00036   Time: 0.01623 sec
Map: 15  Number of steps: 02  Fringe nodes: 00033   Time: 0.01520 sec
Map: 16  Number of steps: 04  Fringe nodes: 00107   Time: 0.07154 sec
Map: 17  Number of steps: 04  Fringe nodes: 00032   Time: 0.01583 sec
Map: 18  Number of steps: 04  Fringe nodes: 00015   Time: 0.00865 sec
Map: 19  Number of steps: 04  Fringe nodes: 00028   Time: 0.01544 sec
Map: 20  Number of steps: 05  Fringe nodes: 00032   Time: 0.01863 sec
Map: 21  Number of steps: 04  Fringe nodes: 00275   Time: 0.22880 sec
Map: 22  Number of steps: 04  Fringe nodes: 00161   Time: 0.12307 sec
Map: 23  Number of steps: 04  Fringe nodes: 00084   Time: 0.04307 sec
Map: 24  Number of steps: 04  Fringe nodes: 00049   Time: 0.02896 sec
Map: 25  Number of steps: 04  Fringe nodes: 00011   Time: 0.01019 sec
Map: 26  Number of steps: 04  Fringe nodes: 00016   Time: 0.01222 sec
Map: 27  Number of steps: 06  Fringe nodes: 00105   Time: 0.10248 sec
Map: 28  Number of steps: 04  Fringe nodes: 00035   Time: 0.02671 sec
Map: 29  Number of steps: 06  Fringe nodes: 00063   Time: 0.04513 sec
Map: 30  Number of steps: 05  Fringe nodes: 00077   Time: 0.06470 sec
Map: 31  Number of steps: 19  Fringe nodes: 01108   Time: 1.96581 sec
Map: 32  Number of steps: 21  Fringe nodes: 00202   Time: 0.28767 sec
Map: 33  Number of steps: 17  Fringe nodes: 00968   Time: 1.69310 sec
Map: 34  Number of steps: 17  Fringe nodes: 00998   Time: 1.82763 sec
Map: 35  Number of steps: 21  Fringe nodes: 00894   Time: 0.88867 sec
Map: 36  Number of steps: 17  Fringe nodes: 00071   Time: 0.06693 sec
Map: 37  Number of steps: 18  Fringe nodes: 00507   Time: 0.78070 sec
Map: 38  Number of steps: 22  Fringe nodes: 03857   Time: 7.95316 sec
Map: 39  Number of steps: 26  Fringe nodes: 01978   Time: 4.35887 sec
Map: 40  Number of steps: 18  Fringe nodes: 01495   Time: 2.42070 sec
Map: 41  Number of steps: 24  Fringe nodes: 00419   Time: 0.52014 sec
Map: 42  Number of steps: 26  Fringe nodes: 00815   Time: 1.32013 sec
Map: 43  Number of steps: 26  Fringe nodes: 03202   Time: 5.18799 sec
Map: 44  Number of steps: 26  Fringe nodes: 14565   Time: 25.32084 sec
Map: 45  Number of steps: 25  Fringe nodes: 02598   Time: 4.58701 sec
Map: 46  Number of steps: 23  Fringe nodes: 01411   Time: 2.71577 sec
Map: 47  Number of steps: 10  Fringe nodes: 00078   Time: 0.05872 sec
Map: 48  Number of steps: 13  Fringe nodes: 00334   Time: 0.22896 sec
Map: 49  Number of steps: 21  Fringe nodes: 00486   Time: 0.70398 sec
Map: 50  Number of steps: 06  Fringe nodes: 00152   Time: 0.09006 sec
Map: 51  Number of steps: 05  Fringe nodes: 00108   Time: 0.05946 sec
Map: 52  Number of steps: 08  Fringe nodes: 00134   Time: 0.10940 sec
Map: 53  Number of steps: 09  Fringe nodes: 00463   Time: 0.39296 sec
Map: 54  Number of steps: 10  Fringe nodes: 00365   Time: 0.34074 sec
Map: 55  Number of steps: 13  Fringe nodes: 02648   Time: 4.96581 sec
```

Appendix A. Randomize Map

The method we intended to use is to initialize a map instance, and then move the blocks randomly for a number of times. We would compare the result when we successfully generated the map.

Re-Implement classes

In [44]:

```
class Block:
    """
    x range from 0 to 5
    y range from 0 to 5
    """
    def __init__(self, x, y, direction, length):
        global index
        self.start_point_x = x
        self.start_point_y = y
        self.direction = direction
        self.length = length
        self.indx = index
        index += 1          #when you initialize next item, its index will raise

    def __str__(self):
        a = 'start_point_x = %d\
        start_point_y = %d\
        direction = %s\
        length = %d\
        index  = %d'\
            % (int(self.start_point_x),\
            int(self.start_point_y),\
            self.direction,\
            int(self.length),\
            int(self.indx))
        return a

    def move(self, step, direct):
        """
```

```
        :param step: int
        :param direct: -1/+1
        :return: new block
        """
        if self.direction == "h":
            self.start_point_x += step * direct
        if self.direction == "v":
            self.start_point_y += step * direct
```

In [45]:

```
class Map:
    def __init__(self):
        self.map = np.array([[0 for i in range(6)] for i in range(6)])

    def __str__(self):
        return str(self.map)

    def add_block(self, blk):
        x, y, direction, length, index = (
            blk.start_point_x,
            blk.start_point_y,
            blk.direction,
            blk.length,
            blk.indx,
        )
        if direction == "h":
            for i in range(length):
                self.map[y][x + i] = index
        if direction == "v":
            for i in range(length):
                self.map[y + i][x] = index

    def possible_move(self, blk):
        x, y, direction, length, index = (
            blk.start_point_x,
            blk.start_point_y,
            blk.direction,
            blk.length,
            blk.indx
        )
        move_list = []
        if direction == "v":
            if y == 0:
                up = 0
            else:
                for up in range(1, y + 1):
                    if self.map[y - up][x] != 0:
                        up -= 1
                        break
                    else:
                        move_list.append((up, -1, index))

            if y + length == 6:
                down = 0
            else:
                for down in range(1, 7 - y - length):
                    if self.map[y + length + down - 1][x] != 0:
                        down -= 1
                        break
                    else:
                        move_list.append((down, +1, index))

        if direction == "h":
            if x == 0:
                left = 0
            else:
                for left in range(1, x + 1):
                    if self.map[y][x - left] != 0:
                        left -= 1
                        break
                    else:
                        move_list.append((left, -1, index))
            if x + length == 6:
                right = 0
            else:
                for right in range(1, 7 - x - length):
                    if self.map[y][x + length + right - 1] != 0:
                        right -= 1
                        break
                    else:
                        move_list.append((right, +1, index))
        return move_list
```

In [46]:

```
class State():
    def __init__(self, AllBlocks):
        self.AllBlocks = AllBlocks
        self.GameMap = Map()
        for block in self.AllBlocks:
            self.GameMap.add_block(block)
    def GetMap(self):
        return self.GameMap

    def GetNextMoves(self):
        All_Moves_list = []
```

```
        a = self.GetMap()
        for blk in self.AllBlocks:
            All_Moves_list += a.possible_move(blk)
        return All_Moves_list

    def Display(self):
        return self.GameMap.__str__()

    def NextStates(self, move):
        New = []
        for block in self.AllBlocks:
            TempBlock = copy.copy(block)
            if TempBlock.indx == move[2]:
                TempBlock.move(move[1], move[0])
            New.append(TempBlock)
        return State(New)
```

Randomizer and testing with Breadth-first search

In [47]:

```
def Goal_state(node):
    start = node.AllBlocks[-1].start_point_x
    if sum(node.GameMap.map[2, start: 6]) == node.AllBlocks[-1].indx * node.AllBlocks[-1].length:
        return True
    return False

def SuccGen(CurrState):
    ChildLst = []
    for move in CurrState.GetNextMoves():
        NextState = CurrState.NextStates(move)
        if NextState.GameMap.__str__() not in AdjacentStates:
            AdjacentStates[NextState.GameMap.__str__()] = CurrState.GameMap.__str__()
            ChildLst.append(NextState)
    return ChildLst

def Trace():
    CurrPos = FinishNode
    while CurrPos != InitState.GameMap.__str__():
        path.append(CurrPos)
        CurrPos = AdjacentStates[CurrPos]

def BFS(root):
    if Goal_state(root):
        return 'Already goal'
    else:
        Queue = SuccGen(root)
        while len(Queue) != 0:
            CurrNode = Queue.pop(0)
            if Goal_state(CurrNode):
                return CurrNode.GameMap.__str__()
            else:
                for ChildNode in SuccGen(CurrNode):
                    Queue.append(ChildNode)
        return 'Failure'
```

Simple randomizer

In [48]:

```
import random as rd
Tag = 29
```

In [49]:

```
def map_randomizer(Tag):
    global index
    index = 1
    _, All_Blocks = read_input('testcases/inp%02d.txt' % Tag)
    InitialState = State(All_Blocks)
    CurrentState = InitialState
    for i in range(50):
        move = rd.choice(CurrentState.GetNextMoves())
        CurrentState = CurrentState.NextStates(move)
    return CurrentState
```

In [50]:

```
index = 1
AdjacentStates = {}
path = []

InitState = map_randomizer(Tag)
print ('\nGenerated Map:')
print(InitState.Display())
```

```
Generated Map:
[[ 1  0  3  3  0  0]
 [ 1  0  0  2  4  5]
 [ 6 14 14  2  4  5]
 [ 6  0  8  7  9 10]
 [11 11  8  7  9 10]
 [12 12  8 13 13  0]]
```

Randomizer test with Breadth-first search: result

In [51]:
```
for x in range (1,10):
  InitState = map_randomizer(Tag)
  index = 1
  AdjacentStates = {}
  path = []
  FinishNode = BFS(InitState)
  if FinishNode == 'Already goal':
    print('Test: %d     Goal reached already' %x)

  elif FinishNode != 'Failure':
      Trace()
      print('Test: %d     Number of steps: %d' %(x, len(path)))
  else:
      print('Test: %d     No ways to escape!' %x)
```

```
Test: 1     Number of steps: 4
Test: 2     Goal reached already
Test: 3     Number of steps: 6
Test: 4     Number of steps: 6
Test: 5     Number of steps: 5
Test: 6     Number of steps: 1
Test: 7     Number of steps: 4
Test: 8     Goal reached already
Test: 9     Number of steps: 2
```