Homework 2 Report

0516034 楊翔鈞

作業題目為給定一個 6 * 6 大小的踩地雷，還有 16 個標有提示的格子，並找出 10 個地雷所在的位置。這次的作業要用 Backtrack Search 來求解，同時比較 forward checking 與 heuristic 對於 Backtrack Search 的影響。

- forward checking 主要檢查下列幾項：
    - 對於每個標有提示的格子，將格子周圍至多八個 variable 的 domain 加總，upper bound 為總和的最大值，lower bound 為總和的最小值
    - 如果 lower bound 比提示的數字還要大，則目前這個狀態不可能會有解
    - 如果 upper bound 比提示的數字還要小，則目前這個狀態不可能會有解
    - 如果 lower bound 跟提示的數字相等，則周圍至多八個 variable 的 domain 只能剩下 domain 中的最小值
    - 如果 upper bound 跟提示的數字相等，則周圍至多八個 variable 的 domain 只能剩下 domain 中的最大值
    - 不符合上述情況，則 domain 維持現狀
- 而 heuristic 則有以下三種：
    - MRV：domain 越少的 variable 越先 assign
    - Degree heuristic：跟此 variable 有關的 constraint 數量
    - LCV：針對 domain 不只一個的 variable，對 domain 中的值依序做 forward checking，比較 assign 該值後對其他 variable 造成的影響

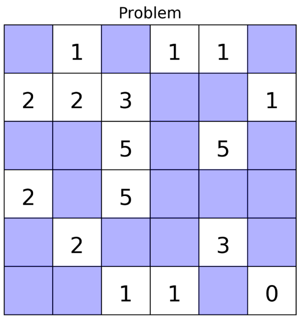
Backtrack Search 的流程為從 stack pop 出一個 node，做 forward checking 更新 domain，然後利用 heuristic 決定 expand 之後要 push 進 stack 的順序，以下為 pseudo-code：

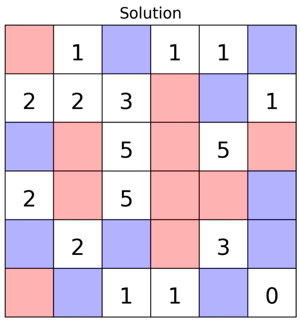
```
stack.push(初始狀態)
while stack 不為空
    node = stack.pop()
    if 對 node 做 forward checking 後沒有不合法的情況
        if 沒有尚未 assign 的 variable
            if 是合法的解
                return 解
            else
                continue
        expand 這個 node
        依照 MRV 與 degree heuristic 去決定 push 進 stack 的順序
```

可以發現決定 push 進 stack 的順序的那部分沒有提到 LCV，因為 LCV 是對 domain 裡面的每個值做 forward checking，所以我把這個過程跟一開始的 forward checking 併在一起了，expand 的時候就單純把 child node 都先 push 進去 stack，反正 pop 出來後如果 forward checking 發現錯誤也不會繼續下去。

關於實驗的部分，我每次都會隨機生成一個 6 * 6 的盤面，包含 10 顆地雷與 16 個提示，然後比較有無 forward checking、MRV、degree heuristic 的 node expansion 數量，下面會先附上 pdf 裡面的那四個盤面，然後再列舉幾個隨機生成的盤面，並比較程式的執行結果。

#盤面 1

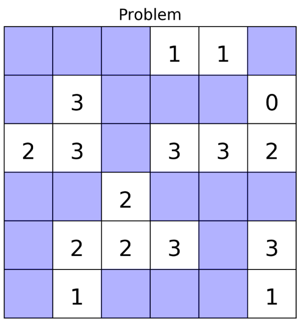


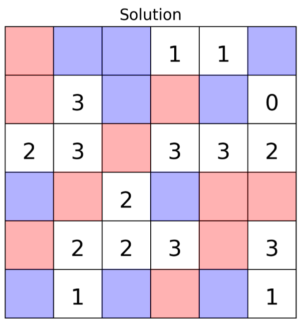


有無 forward checking 或 heuristic	node expansion 數量
forward checking, MRV, degree heuristic	20
forward checking, MRV	20
forward checking, degree heuristic	20
forward checking	20
MRV, degree heuristic	20
MRV	20
degree heuristic	20
nothing	20

- 所有情況下的表現都一樣。

#盤面 2





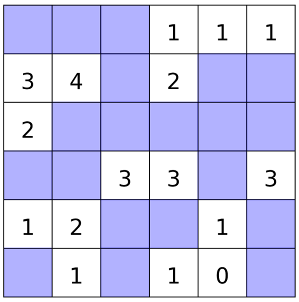
有無 forward checking 或 heuristic	node expansion 數量
forward checking, MRV, degree heuristic	20
forward checking, MRV	346
forward checking, degree heuristic	20
forward checking	跑不出來
MRV, degree heuristic	20

MRV	346
degree heuristic	20
nothing	跑不出來

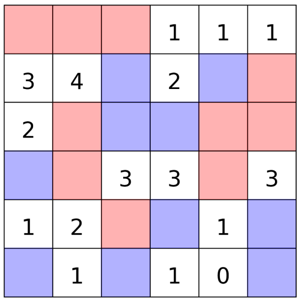
- 有 degree heuristic 的組別表現明顯優於其它者。

#盤面 3

Problem



Solution

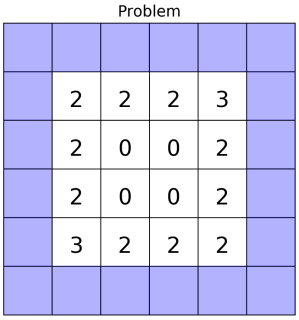


有無 forward checking 或 heuristic	node expansion 數量
forward checking, MRV, degree heuristic	2499
forward checking, MRV	20
forward checking, degree heuristic	跑不出來
forward checking	20
MRV, degree heuristic	2499
MRV	20
degree heuristic	跑不出來
nothing	20

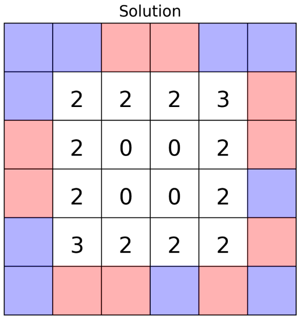
- 只有 degree heuristic 的組別會跑很久算不出解。
- 有 degree heuristic 與 MRV 的組別比只有 degree heuristic 的好。

#盤面 4

Problem



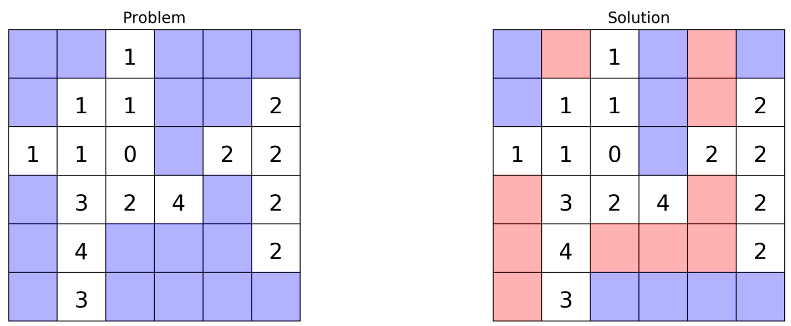
Solution



有無 forward checking 或 heuristic	node expansion 數量
forward checking, MRV, degree heuristic	20
forward checking, MRV	20
forward checking, degree heuristic	20
forward checking	20
MRV, degree heuristic	20
MRV	20
degree heuristic	20
nothing	20

- 所有情況下的表現都一樣。

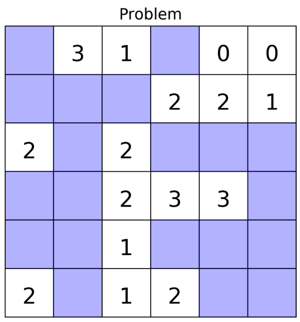
#盤面 5（隨機生成）

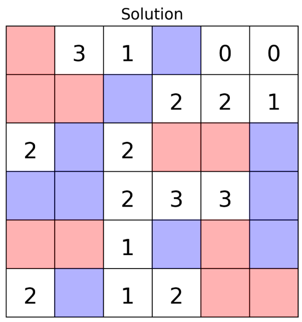


有無 forward checking 或 heuristic	node expansion 數量
forward checking, MRV, degree heuristic	1287
forward checking, MRV	20
forward checking, degree heuristic	跑不出來
forward checking	20
MRV, degree heuristic	1287
MRV	20
degree heuristic	跑不出來
nothing	20

- 只有 degree heuristic 的組別會跑很久算不出解。
- 有 degree heuristic 與 MRV 的組別比只有 degree heuristic 的好。

#盤面 6（隨機生成）





有無 forward checking 或 heuristic	node expansion 數量
forward checking, MRV, degree heuristic	20
forward checking, MRV	跑不出來
forward checking, degree heuristic	20
forward checking	跑不出來
MRV, degree heuristic	20
MRV	跑不出來
degree heuristic	20
nothing	跑不出來

- 沒有 degree heuristic 的那些組會跑很久算不出解。

#觀察

- Forward checking 可以減少被 push 進 stack 的 node 數量，因為對於每個 node, forward checking 都會檢查並跟新其 domain，domain 變小了可能的 child node 也就變少了。
- MRV 的特點就是可以先把 domain 越小的 variable 的狀態定下來，但必須在有 forward checking 的情況下才能發揮這個優勢，因為只有 forward checking 會去修改 variable 的 domain。
- 利用 degree heuristic 能夠很快滿足給定的 16 個提示的 constraint，但無法確保找到的解符合第 17 個 constraint（地雷總數）也就是 global constraint，導致 Backtrack Search 一直把重心放在那 16 個 constraint，直到找到一個可能的解才去檢查第 17 個 constraint，這也是為什麼上面有些盤面在有 degree heuristic 的情況下反而耗費更多時間。
- 從以上這六種盤面看來，沒辦法肯定有 heuristic 的就會比較有效率（node expansion 比較少），也沒辦法說 MRV 跟 degree heuristic 誰的效果好，我想這兩者是需要互相搭配的。

```python
import numpy as np
from datetime import datetime
import matplotlib.pyplot as plt
from collections import OrderedDict

MINE = 100
SAFE = 200
ASSIGN = 0
UNASSIGN = 1


# plot the board
# variables and safe places will be marked as blue cell
# mines will be marked as red cell
def plot_board(board, fig, subplot):
    ax = fig.add_subplot(subplot)
    board_size = board.shape[0]
    args = OrderedDict(
        fontsize=20,
        horizontalalignment='center',
        verticalalignment='center')

    for i in range(board.shape[0]):
        for j in range(board.shape[1]):
            if board[i, j]==-1 or board[i, j]==SAFE:
                ax.fill([j, j, j+1, j+1], [board_size-i-1, board_size-i,
board_size-i, board_size-i-1], c='b', alpha=0.3)
            elif board[i, j] == MINE:
                ax.fill([j, j, j+1, j+1], [board_size-i-1, board_size-i,
board_size-i, board_size-i-1], c='r', alpha=0.3)
            else:
                ax.annotate(board[i, j], (j+0.5, board_size-i-0.6),
**args)

    ax.set_xticks(np.arange(board_size+1))
    ax.set_yticks(np.arange(board_size+1))
    ax.set_xlim([0, board_size])
    ax.set_ylim([0, board_size])
    ax.tick_params(bottom=False, top=False, left=False, right=False)
    ax.tick_params(labelbottom=False, labeltop=False, labelleft=False,
labelright=False)
    ax.grid(b=True, which='major', color='k', linestyle='-')
    ax.set_aspect('equal')


# generate a board randomly with 10 mines and 16 local constraints
def generate_board(board_size=6, mine_num=10, cstr_num=16):
    mines = set()
    while len(mines) < mine_num:
        i, j = np.random.randint(0, board_size, 2)
```

```
            mines.add((i, j))
    board = np.zeros((board_size+2, board_size+2), dtype=np.int32)
    for mine in mines:
        board[mine[0]+1, mine[1]+1] = -1
        i, j = mine[0]+1, mine[1]+1
        for di in [-1, 0, 1]:
            for dj in [-1, 0, 1]:
                new_i = i + di
                new_j = j + dj
                if board[new_i, new_j] == -1:
                    continue
                board[new_i, new_j] += 1
    board = board[1:-1, 1:-1]
    i, j = np.where(board>=0)
    hints = np.asarray(list(zip(i, j)))
    safe_num = board_size**2 - mine_num
    mask_num = safe_num - cstr_num
    mask_idx = np.random.choice(np.arange(safe_num), mask_num,
replace=False)
    mask_hints = hints[mask_idx]
    for i, j in mask_hints:
        board[i, j] = -1
    board_str = ' '.join(board.ravel().astype(str))
    board_str = f'{board_size} {board_size} {mine_num} {board_str}'
    return board_str


# find the neighbor cells of given position, hint cells will be ignored
def get_neighbors(board, i, j):
    neighbors = []
    for di in [-1, 0, 1]:
        for dj in [-1, 0, 1]:
            if di==0 and dj==0:
                continue
            new_i = i + di
            new_j = j + dj
            if (new_i<0 or new_i>=board.shape[0] or
                    new_j<0 or new_j>=board.shape[1]):
                continue

            # ignore it if it is a hint cell
            if board[new_i, new_j] != -1:
                continue
            neighbors.append((i+di, j+dj))
    return neighbors


# do the forward checking on given status
def forward_checking(board, mine_num, cstr, assign, unassign):
    while True:
```

```
        cur_mine_num = sum([v for k, v in assign.items()])
        # if current number of mine cells exceeds total number of mines,
        # it is impossible to derive an answer from this status
        if cur_mine_num > mine_num:
            return False

        # compute the upper bound and lower bound of the sum of domains
of unassigned variables
        # if upper bound is smaller than number of mines remaining or
        # lower bound is larger than number of mines remaining,
        # it is impossible to derive an answer from this status
        remain_mine_num = mine_num - cur_mine_num
        upper_bound = sum([max(v) for k, v in unassign.items()])
        lower_bound = sum([min(v) for k, v in unassign.items()])
        if upper_bound < remain_mine_num:
            return False
        if lower_bound > remain_mine_num:
            return False

        prev_assign = dict(assign)
        prev_unassign = dict(unassign)
        for ((i, j), hint) in cstr:
            lower_bound = 0
            upper_bound = 0
            neighbors = get_neighbors(board, i, j)
            for pos in neighbors:
                if pos in assign:
                    hint -= assign[pos]
                else:
                    upper_bound += max(unassign[pos])
                    lower_bound += min(unassign[pos])

            # if the lower bound is larger than the hint, the constraint
cannot be satisfied.
            if lower_bound > hint:
                return False

            # if the upper bound is smaller than the hint, the
constraint cannot be satisfied.
            if upper_bound < hint:
                return False

            # if the lower bound equals the hint, the domains of all the
            # unassigned variables in the constraint should be limited
to
            # their respective minimal values.
            if lower_bound == hint:
                for pos in neighbors:
                    if pos not in assign:
                        unassign[pos] = [min(unassign[pos])]
```

```
            # if the upper bound equals the hint, the domains of all the
            # unassigned variables in the constraint should be limited
to
            # their respective maximal values
            if upper_bound == hint:
                for pos in neighbors:
                    if pos not in assign:
                        unassign[pos] = [max(unassign[pos])]

        # keep doing forward checking until the domains of all the
unassigned variables remain unchanged
        if prev_assign==assign and prev_unassign==unassign:
            break
    return True


# check whether the final assignment is valid or not
def is_valid_answer(board, mine_num, cstr, assign):
    total_mine = sum([v for k, v in assign.items()])
    # global constraint
    if total_mine != mine_num:
        return False

    # contraint given by each hint
    for ((i, j), hint) in cstr:
        neighbors = get_neighbors(board, i, j)
        for n in neighbors:
            hint -= assign[n]
        if hint != 0:
            return False
    return True


# compute the degree of a given variable
# degree = number of constrains regard to this variable
def get_degree(board, i, j):
    degree = 0
    for di in [-1, 0, 1]:
        for dj in [-1, 0, 1]:
            if di==0 and dj==0:
                continue
            new_i = i + di
            new_j = j + dj
            if (new_i<0 or new_i>=board.shape[0] or
                    new_j<0 or new_j>=board.shape[1]):
                continue
            if board[new_i, new_j] != -1:
                degree += 1
    return degree
```

```
# find the answer using backtrack search
# use forward_checking, MRV, degree_heuristic to control the detail of
backtrack search
def solve(
        board, mine_num, forword_checking=True,
        MRV=True, degree_heuristic=True):
    cstr = []

    # assign = {
    #     variable: assignment (is or is not a mine)
    # }
    assign = {}

    # unassign = {
    #     variable: domain
    # }
    unassign = {}
    for i in range(board.shape[0]):
        for j in range(board.shape[1]):
            if board[i, j] == -1:
                unassign[(i, j)] = [1, 0]
            else:
                cstr.append(((i, j), board[i, j]))
    cstr = sorted(cstr, key=lambda t: t[1])
    expand = 0
    stack = []
    stack.append((assign, unassign))
    while stack:
        node = stack.pop()

        # do the forward checking if forward_checking is True
        if ((not forward_checking) or
                forward_checking(board, mine_num, cstr, node[ASSIGN],
node[UNASSIGN])):

            # check the answer if there is no unassigned variables
            if not node[UNASSIGN].keys():
                if is_valid_answer(board, mine_num, cstr, node[ASSIGN]):
                    return (expand, node[ASSIGN])
                else:
                    continue

            # computer the order of child node to be pushed into the
stack
            t = {}
            for pos in node[UNASSIGN]:

                # MRV heuristic
```

```
                domain = 0
                if MRV:
                    domain = len(node[UNASSIGN][pos])

                # degree heuristic
                degree = 0
                if degree_heuristic:
                    degree = get_degree(board, pos[0], pos[1])
                t[pos] = (-domain, degree)
            order = [k for k, v in sorted(t.items(), key=lambda elem:
elem[1])]
            for pos in order:
                for value in node[UNASSIGN][pos]:
                    new_assign = dict(node[ASSIGN])
                    new_unassign = dict(node[UNASSIGN])

                    # assign a value to this variable
                    new_assign[pos] = value

                    # remove this variable from unassigned list
                    new_unassign.pop(pos, None)
                    stack.append((new_assign, new_unassign))
            expand += 1


if __name__ == '__main__':
    # generate a board randomly
    board = generate_board()
    board = np.asarray(board.strip().split(' ')).astype(np.int32)
    board_size = board[0:2]
    mine_num = board[2]
    board = board[3:].reshape(board_size)

    # plot the problem
    fig = plt.figure()
    plot_board(board, fig, 111)
    plt.title('Problem', fontsize=14)
    plt.savefig(f'problem', dpi=300, transparent=True)
    plt.close(fig)

    # solve the board with backtrack search
    fc, mrv, dh = True, True, True
    start = datetime.now()
    expand, assign = solve(
        board, mine_num, forword_checking=fc, MRV=mrv,
degree_heuristic=dh)
    end = datetime.now()
    delta = end - start

    print(f'size {board_size[0]}x{board_size[1]}')
```

```
print(f'{mine_num} mines\nexpand {expand} nodes')
print(f'forward checking: {fc}')
print(f'MRV: {mrv}')
print(f'degree heuristic: {dh}')
print(delta, end='\n\n')

# fill the board with the solution
for pos in assign:
    board[pos] = MINE if assign[pos] else SAFE

# plot the solution
fig = plt.figure()
plot_board(board, fig, 111)
plt.title('Solution', fontsize=14)
filename = f'solution'
if fc:
    filename = f'{filename}_fc'
if mrv:
    filename = f'{filename}_mrv'
if dh:
    filename = f'{filename}_dh'
filename = f'{filename}.png'
plt.savefig(f'{filename}', dpi=300, transparent=True)
plt.close(fig)
```