Introduction to Artificial Intelligence Prog. 2 Report

0516003 李智嘉

1. Introduction

- In this project, we will write searching algorithms to find solutions of Minesweeper problem with three heuristic function and forwarding checking in DFS.
- There will be four type of implementation in this project.
  i. None (just simply iterate every node and domain values)
  ii. MRV
  iii. Degree
  iv. LCV (here I randomly choose one variable to expand, and choose the domain values by LCV)
- In the following experiment, I'll set **"base setting"** as **6 by 6 board**, with **10 mines** and **16 hints** (just like the examples spec provided). And divide experiment into five parts
  i. Small setting, with/without forwarding checking
  ii. Base setting, changing mines number with forwarding checking
  iii. Base setting, changing hints number with forwarding checking
  iv. Base setting, changing board size, with forwarding checking and same ratio from base setting (mine ratio: 0.277, hint ratio: 0.444)
  v. The max board size with same mine and hint ratio for four method in 3 second.
- For every sub-experiment, I'll sample **500** random generated board, and compare the **average expanded nodes number** and **average running time per board**.

2. Experiment

I. Small Setting (4, 4, 4, 7) (board size, #mines, #hints)

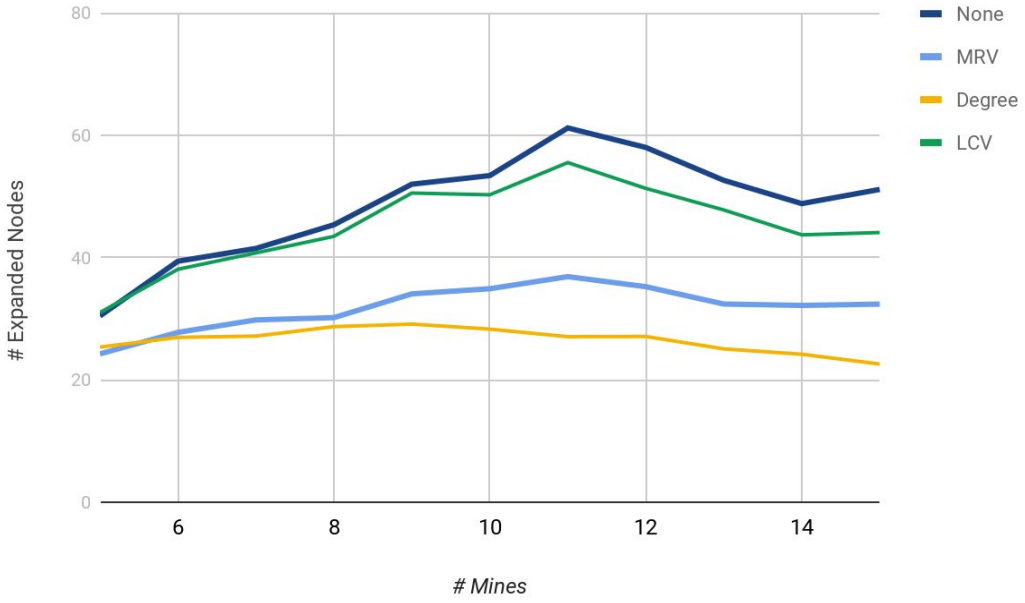
- Here I want to see the power of forwarding checking. The number means expanded nodes.

Version	w/o forward	w/ forward
None	208.618 (0.0015s)	14.5 (0.00022s)
MRV	208.618 (0.0017s)	12.49 (0.00020s)
Degree	223.78 (0.0018s)	11.342 (0.00018s)
LCV	14.092 (0.0003s)	13.988 (0.00026s)

- We can see that, forwarding checking can largely accelerate the speed of 'None', 'MRV' and 'Degree' version.

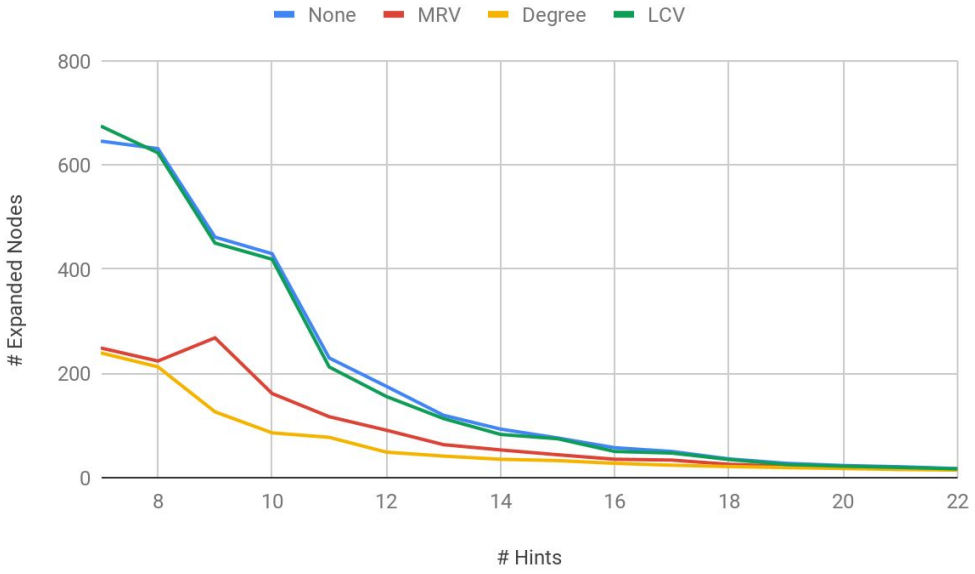
- For LCV, since I discard the value which would make board illegal, just like a pre-forward checking. So the expanded node and time consuming is very similar in both with and without forwarding checking.
- For MRV, since we don't do forwarding checking, the remaining domain would not change. That's why the expanded nodes number of MRV and None is equal.

II. Base setting, changing mines number



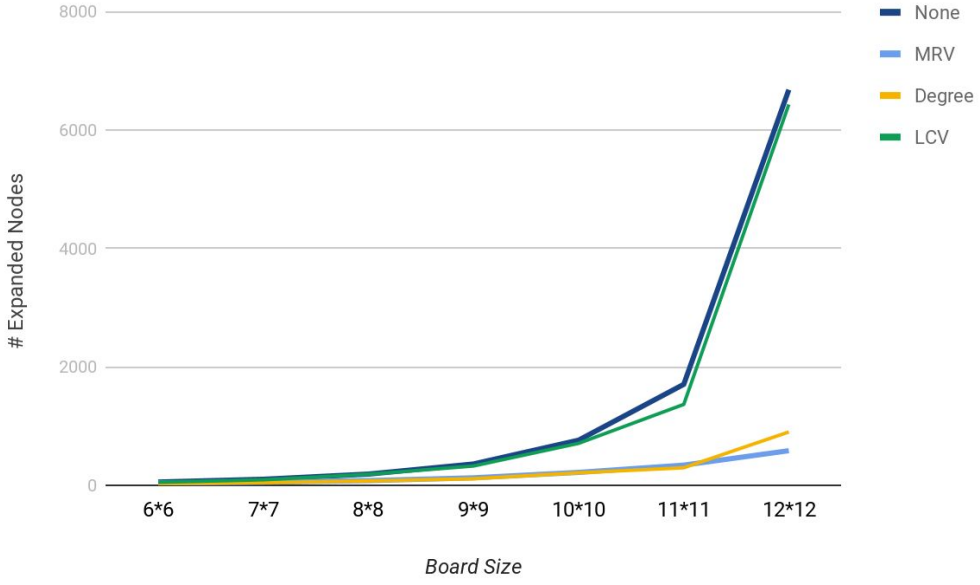
- In this part, I change the mines number and see how much the expanded nodes every algorithm should take.
- We can see that, for None and LCV and MRV, when the number of mines is small, the expanded nodes is also small. The reason may be that it will default assign 0 firstly, so when the number of mines is small, the chance it will has error is also small.
- As the number of mines increasing, the advantage of LCV is getting stronger. We can see that when the number of mines is small, LCV is very close to None. But when the number of mines increasing, the difference between them is getting larger.
- For Degree, it is relatively stable when we change number of mines.

III. Base setting, changing hints number



- In this part, I change the hints number and see how much the expanded nodes every algorithm should take.
- As the number of hints increasing, the expanded nodes every algorithm need are also decreasing. And the slope of decreasing are getting smoother as the number of hints increasing.
- In the previous two experiment, the performance of Degree is the best, next is MRV. LCV and None are very similar, but for more mines, LCV is better than None.

IV. Changing board size, with same ratio of mines and hints

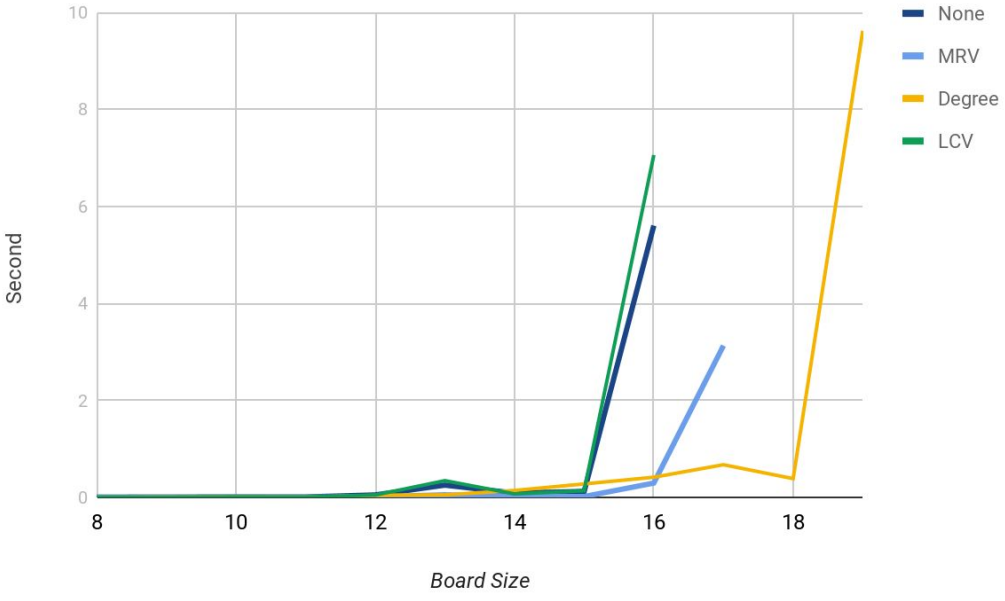


- In this part, I change the board size and see how much the expanded nodes every algorithm should take.
- We can see that, when it comes to large board (12 by 12 here), LCV and None would increase significantly, while Degree and MRV could still be less than 1000 nodes.
- Interestingly, at 12 by 12 board, the expanded nodes of Degree is larger than MRV. Maybe that comes from sample bias, since when the board size is getting larger, the more possible board setting could be, and we just sample the board which Degree would expand more nodes.
- So I sample more board (5000 board here) at 12 by 12 for MRV and Degree. Result shows that the performance of Degree is still better than MRV.

Version	# Expanded Nodes	Avg tims/board
MRV	983.1746	0.0539
Degree	565.294	0.0315

V. The max board size with same setting ratio in 3 second

- In this part, I'll start from 8 by 8 board with same ratio of mines and hints, to see that at which size of the board, the average running time would greater than 3 second.
- Since if the average time is greater than 3 second, the total time this experiment need would be very huge if the sample size is large. So here I only sample 5 board, which is also sensible since we hope to see that at most cases the algorithm would find a solution less than 3 second.



    - We can see that, the time LCV need is greater than None at 12 by 12 board. The reason may be that the time it need to compute affected domains is considerable.
    - Degree still perform the best, which can reach 18 by 18 size board with average time less than 1 second.

3. Conclusion

- As the above experiment shows, heuristic of Degree perform the best. The reason may be the type of this game.
- Since the domains is only 1 and 0, MRV could not perform very well when the domains of most variable remain one value (MRV can not give the priority when the remaining domain values are equal).
- And for Degree, the constraint number is not affected by this feature. Moreover, Degree is actually what human would do: pick up the variable with most constraint. So that's not surprising to see that it perform the best.
- For LCV, since I just choose the variable like None heuristic do, so LCV couldn't get much benefit from it's heuristic, which leads to bad performance just like None.

4. More things

- I think this program assignment is very interesting.
- If I have enough time and space, I think I can do more things like
    i. Comparing the time for every board. (previously, I compare the expanded nodes for different heuristic in different situation)
    ii. Apply LCV to different heuristic. Maybe I could use MRV or Degree to choose variable, and LCV to choose domains. To see that would it better than the original version or not.
    iii. Extend the domains. Previously in my conclusion, I said the reason MRV could not perform well may be the few values of domains. Maybe I can change the game rule to see that, in this new game setting, would MRV perform well or not.

```python
## main.py

import utils
import recur_heur

def solve(cur_board, cur_domains, constraints, constraint_list,
forward_check, heur):
    recur_heur.expanded_node_cnt = 0
    if heur == 'None':
        return recur_heur.simple(cur_board, cur_domains, constraints,
constraint_list, None, forward_check), recur_heur.expanded_node_cnt
    elif heur == 'MRV':
        return recur_heur.MRV(cur_board, cur_domains, constraints,
constraint_list, None, forward_check), recur_heur.expanded_node_cnt
    elif heur == 'Degree':
        return recur_heur.Degree(cur_board, cur_domains, constraints,
constraint_list, None, forward_check), recur_heur.expanded_node_cnt
    elif heur == 'LCV':
        return recur_heur.LCV(cur_board, cur_domains, constraints,
constraint_list, None, forward_check), recur_heur.expanded_node_cnt

if __name__ == "__main__":
    ## Read and process input
    inp = input()
    orig_map, board, domains, constraints, constraint_list =
utils.parse_input(list(map(lambda x: int(x), inp.split(' '))))

    ## Experiment Flags
    forward_check = True
    heur = 'Degree' ## could be: None, MRV, Degree, LCV

    ## Print board
    utils.print_board(orig_map, board)

    print(f'Forward Checking: {forward_check}, {heur} version')

    ## Start search
    board, node_num = solve(board, domains, constraints,
constraint_list, forward_check, heur)
```

```
    ## Print result
    if board is None:
        print('No solution')
    else:
        print(f'Number of expanded nodes: {node_num}')
        utils.print_board(orig_map, board)
```

```
## utils.py

import copy

def add_constraint(constraints, constraint_list, x_idx, y_idx, board,
x_size, y_size):
    node_list = []
    for x in [-1, 0, 1]:
        for y in [-1, 0, 1]:
            if y_idx+y < 0 or x_idx+x < 0 or x_idx+x >= x_size or
y_idx+y >= y_size:
                continue
            if board[y_idx+y][x_idx+x] == -1:
                node_list.append((x_idx+x, y_idx+y))
    for node in node_list:
        constraints[node].append((node_list, board[y_idx][x_idx]))
    constraint_list.append((node_list, board[y_idx][x_idx]))

    return constraints, constraint_list

def parse_input(inp):
    x_size, y_size, mines_num = inp[0], inp[1], inp[2]
    board = []

    ## key: (x, y)
    ## value: [[True, True], True], it's domain and it's unassigned or
not
    domains = dict()

    ## every elements in constraints would be dict
    ## key: (x, y)
    ## value: [constraint, constraint]
```

```
    ## the constraint would be ([(node1), (node2), (node3), ...],
constraint value)
    constraints = dict()
    constraint_list = []

    for y_idx in range(y_size):
        board.append([])
        for x_idx in range(x_size):
            val = inp[3 + y_idx*x_size + x_idx]
            board[-1].append(val)
            if val == -1:   ## variable
                # domains[(x_idx, y_idx)] = [True, True]
                domains[(x_idx, y_idx)] = [[True, True], True]
                constraints[(x_idx, y_idx)] = list()

    for y_idx in range(y_size):
        for x_idx in range(x_size):
            if board[y_idx][x_idx] != -1:
                ## add constraints
                add_constraint(constraints, constraint_list, x_idx,
y_idx, board, x_size, y_size)

    ## add global constraint
    node_list = [node for node in domains.keys()]
    for node in node_list:
        constraints[node].append((node_list, mines_num))
    constraint_list.append((node_list, mines_num))

    orig_map = copy.deepcopy(board)

    for y_idx in range(y_size):
        for x_idx in range(x_size):
            board[y_idx][x_idx] = 0

    return orig_map, board, domains, constraints, constraint_list

def check_accept(board, constraint_list):
    for constraint in constraint_list:
        node_list, tar_value = constraint
        value = sum([board[y][x] for x, y in node_list])
```

```
        if value != tar_value:
            return False
    return True

def check_avail(board, domains, constraint_list, just_calc=False):
    history = []    ## changed history, element would be (node, idx)
    out_cnt = 0
    for constraint in constraint_list:
        node_list, tar_value = constraint
        value = sum([board[y][x] for x, y in node_list])
        upper_bound, lower_bound = value, value

        for node in node_list:
            # if node in domains:
            if domains[node][1]:
                if domains[node][0][1]:
                    upper_bound += 1
                elif not domains[node][0][0] and domains[node][0][1]:
                    lower_bound += 1

        if upper_bound < tar_value or lower_bound > tar_value:
            if not just_calc:
                return False, history
            else:
                reverse_domain_hist(domains, history)
                return False, out_cnt

        if upper_bound == tar_value:    ## set all domains to upper
bound value
            for node in node_list:
                # if node in domains:
                if domains[node][1]:
                    if not domains[node][1]:
                        continue
                    if domains[node][0][1] and domains[node][0][0]:
                        history.append((node, 0))
                        domains[node][0][0] = False
                        if just_calc:
                            out_cnt += 1
```

```
        if lower_bound == tar_value:    ## set all domains to lower
bound value
            for node in node_list:
                # if node in domains:
                if domains[node][1]:
                    if not domains[node][1]:
                        continue
                    if domains[node][0][0] and domains[node][0][1]:
                        history.append((node, 1))
                        domains[node][0][1] = False
                        if just_calc:
                            out_cnt += 1
    if not just_calc:
        return True, history
    else:
        reverse_domain_hist(domains, history)
        return True, out_cnt

def reverse_domain_hist(domains, history):
    for node, idx in history:
        domains[node][0][idx] = True

def print_board(orig_map, board):
    for y_idx in range(len(orig_map)):
        for x_idx in range(len(orig_map[0])):
            if orig_map[y_idx][x_idx] != -1:
                print(orig_map[y_idx][x_idx], end=' ')
            elif board[y_idx][x_idx] == 0:
                print('O', end=' ')
            else:
                print('X', end=' ')
        print()
```

```
## recur_heur.py

import utils

expanded_node_cnt = 0

## Without heuristic version
```

```
def simple(cur_board, cur_domains, constraints, constraint_list,
changed_node, forward_check):
    global expanded_node_cnt
    ## Do forward checking, only check the constraints which affect
changed node
    if changed_node is not None and forward_check:
        check_res, history = utils.check_avail(cur_board, cur_domains,
constraints[changed_node])
        ## If checking fail, reverse the changed domain, and back to
previous node
        if not check_res:
            utils.reverse_domain_hist(cur_domains, history)
            return None

    ## If all domains has assigned (reach the leaf), check whether it's
acceptable
    ## If failed, reverse and go back to previous node
    ## If succeed, return the solution
    if sum([cur_domains[key][1] for key in cur_domains]) == 0:
        if not utils.check_accept(cur_board, constraint_list):
            if forward_check:
                utils.reverse_domain_hist(cur_domains, history)
            return None
        else:
            return cur_board

    ## Find the node which hasn't assigned
    for (x, y) in cur_domains.keys():
        if not cur_domains[(x, y)][1]:
            continue
        new_x, new_y = x, y
        break

    ## Expand node
    expanded_node_cnt += 1
    domain, _ = cur_domains[(new_x, new_y)]
    for new_val, available in enumerate(domain):
        ## If the domain value is available
        if available:
            cur_domains[(new_x, new_y)][1] = False
```

```
            cur_board[new_y][new_x] = new_val
            res = simple(cur_board, cur_domains, constraints,
constraint_list, (new_x, new_y), forward_check)
            ## If it has solution, return it
            if res is not None:
                return res
            ## Go back to previous status
            cur_domains[(new_x, new_y)][1] = True
            cur_board[new_y][new_x] = 0

    ## Reverse changed by forward checking
    if changed_node is not None and forward_check:
        utils.reverse_domain_hist(cur_domains, history)

    return None

def MRV(cur_board, cur_domains, constraints, constraint_list,
changed_node, forward_check):
    global expanded_node_cnt
    ## Do forward checking, only check the constraints which affect
changed node
    if changed_node is not None and forward_check:
        check_res, history = utils.check_avail(cur_board, cur_domains,
constraints[changed_node])

        ## If checking fail, reverse the changed domain, and back to
previous node
        if not check_res:
            utils.reverse_domain_hist(cur_domains, history)
            return None

    ## If all domains has assigned (reach the leaf), check whether it's
acceptable
    ## If failed, reverse and go back to previous node
    ## If succeed, return the solution
    if sum([cur_domains[key][1] for key in cur_domains]) == 0:
        if not utils.check_accept(cur_board, constraint_list):
            if forward_check:
                utils.reverse_domain_hist(cur_domains, history)
            return None
```

```
        else:
            return cur_board

    ## Choose the node to expand which has minimum available domain
    new_node, min_avail_num = None, -1
    for (new_x, new_y) in cur_domains.keys():
        if not cur_domains[(new_x, new_y)][1]:
            continue
        avail_num = sum(cur_domains[(new_x, new_y)][0])
        if avail_num < min_avail_num or min_avail_num == -1:
            new_node, min_avail_num = (new_x, new_y), avail_num

    ## Expand nodes in priority sequence
    expanded_node_cnt += 1
    new_x, new_y = new_node
    domain, _ = cur_domains[(new_x, new_y)]
    for new_val, available in enumerate(domain):
        ## If the domain value is available
        if available:
            cur_domains[(new_x, new_y)][1] = False
            cur_board[new_y][new_x] = new_val
            res = MRV(cur_board, cur_domains, constraints,
constraint_list, (new_x, new_y), forward_check)

            ## If it has solution, return it
            if res is not None:
                return res

            ## Go back to previous status
            cur_domains[(new_x, new_y)][1] = True
            cur_board[new_y][new_x] = 0

    ## Reverse the changed by forward checking
    if changed_node is not None and forward_check:
        utils.reverse_domain_hist(cur_domains, history)

    return None

def Degree(cur_board, cur_domains, constraints, constraint_list,
changed_node, forward_check):
```

```
    global expanded_node_cnt

    ## Do forward checking, only check the constraints which affect
changed node
    if changed_node is not None and forward_check:
        check_res, history = utils.check_avail(cur_board, cur_domains,
constraints[changed_node])

        ## If checking fail, reverse the changed domain, and back to
previous node
        if not check_res:
            utils.reverse_domain_hist(cur_domains, history)
            return None

    ## If all domains has assigned (reach the leaf), check whether it's
acceptable
    ## If failed, reverse and go back to previous node
    ## If succeed, return the solution
    if sum([cur_domains[key][1] for key in cur_domains]) == 0:
        if not utils.check_accept(cur_board, constraint_list):
            if forward_check:
                utils.reverse_domain_hist(cur_domains, history)
            return None
        else:
            return cur_board

    ## Choose the node to expand which has maximum constraint number
    new_node, max_constraint_num = None, -1
    for (new_x, new_y) in cur_domains.keys():
        if not cur_domains[(new_x, new_y)][1]:
            continue
        constraint_num = len(constraints[(new_x, new_y)])
        if constraint_num > max_constraint_num:
            new_node, max_constraint_num = (new_x, new_y),
constraint_num

    ## Expand the node
    expanded_node_cnt += 1
    (new_x, new_y) = new_node
    domain, _ = cur_domains[(new_x, new_y)]
```

```
    for new_val, available in enumerate(domain):
        ## If the domain value is available
        if available:
            cur_domains[(new_x, new_y)][1] = False
            cur_board[new_y][new_x] = new_val
            res = Degree(cur_board, cur_domains, constraints,
constraint_list, (new_x, new_y), forward_check)

            ## If it has solution, return it
            if res is not None:
                return res

            ## Go back to previous status
            cur_domains[(new_x, new_y)][1] = True
            cur_board[new_y][new_x] = 0

    ## Reverse the changed by forward checking
    if changed_node is not None and forward_check:
        utils.reverse_domain_hist(cur_domains, history)

    return None

def LCV(cur_board, cur_domains, constraints, constraint_list,
changed_node, forward_check):
    global expanded_node_cnt
    ## Do forward checking, only check the constraints which affect
changed node
    if changed_node is not None and forward_check:
        check_res, history = utils.check_avail(cur_board, cur_domains,
constraints[changed_node])

        ## If checking fail, reverse the changed domain, and back to
previous node
        if not check_res:
            utils.reverse_domain_hist(cur_domains, history)
            return None

    ## If all domains has assigned (reach the leaf), check whether it's
acceptable
    ## If failed, reverse and go back to previous node
```

```
    ## If succeed, return the solution
    if sum([cur_domains[key][1] for key in cur_domains]) == 0:
        if not utils.check_accept(cur_board, constraint_list):
            if forward_check:
                utils.reverse_domain_hist(cur_domains, history)
            return None
        else:
            return cur_board

    ## Find the node which hasn't assigned
    for (x, y) in cur_domains.keys():
        if not cur_domains[(x, y)][1]:
            continue
        new_x, new_y = x, y
        break

    value_seq = []
    if cur_domains[(new_x, new_y)][0][0] and cur_domains[(new_x,
new_y)][0][1]:
        ## sort the value in increasing order
        cur_domains[(new_x, new_y)][1] = False
        cur_board[new_y][new_x] = 0
        zero_can_do, zero_out_cnt = utils.check_avail(
            cur_board, cur_domains, constraints[(new_x, new_y)], True
        )
        cur_board[new_y][new_x] = 1
        one_can_do, one_out_cnt = utils.check_avail(
            cur_board, cur_domains, constraints[(new_x, new_y)], True
        )
        cur_board[new_y][new_x] = 0
        cur_domains[(new_x, new_y)][1] = True
        if zero_can_do and one_can_do:
            if zero_out_cnt > one_out_cnt:
                value_seq = [1, 0]
            else:
                value_seq = [0, 1]
        elif zero_can_do:
            value_seq = [0]
        elif one_can_do:
            value_seq = [1]
```

```
    elif cur_domains[(new_x, new_y)][0][0]:
        value_seq = [0]
    elif cur_domains[(new_x, new_y)][0][1]:
        value_seq = [1]

    expanded_node_cnt += 1
    for new_val in value_seq:
        cur_domains[(new_x, new_y)][1] = False
        cur_board[new_y][new_x] = new_val
        res = LCV(cur_board, cur_domains, constraints, constraint_list,
(new_x, new_y), forward_check)

        ## If it has solution, return it
        if res is not None:
            return res

        ## Go back to previous status
        cur_domains[(new_x, new_y)][1] = True
        cur_board[new_y][new_x] = 0

    ## Reverse the changed by forward checking
    if changed_node is not None and forward_check:
        utils.reverse_domain_hist(cur_domains, history)

    return None
```

```
## gen_case.py

import random

## Generate random board
def gen_case(x_size, y_size, mines_num, hint_num):
    board = []
    res = ''
    for _ in range(y_size):
        board.append([])
        for _ in range(x_size):
            board[-1].append('O')

    done_mine_num = 0
```

```
    while done_mine_num != mines_num:
        x, y = random.randint(0, x_size-1), random.randint(0, y_size-1)
        if board[y][x] == 'O':
            board[y][x] = 'X'
            done_mine_num += 1

    done_hint_num = 0
    while done_hint_num != hint_num:
        x, y = random.randint(0, x_size-1), random.randint(0, y_size-1)
        if board[y][x] == 'O':
            cnt = 0
            for y_d in [-1, 0, 1]:
                for x_d in [-1, 0, 1]:
                    if x+x_d<0 or x+x_d>=x_size or y+y_d<0 or
y+y_d>=y_size:
                        continue
                    if board[y+y_d][x+x_d] == 'X':
                        cnt += 1
            board[y][x] = cnt
            done_hint_num += 1

    # for y in range(y_size):
    #     for x in range(x_size):
    #         print(board[y][x], end=' ')
    #     print()

    print(f'{x_size} {y_size} {mines_num}',end=' ')
    res = f'{x_size} {y_size} {mines_num} '
    for y in range(y_size):
        for x in range(x_size):
            if type(board[y][x]) == int:
                print(f'{board[y][x]}',end='')
                res += f'{board[y][x]}'
            else:
                print('-1',end='')
                res += '-1'
            if x != x_size-1 or y != y_size-1:
                print(' ',end='')
                res += ' '
    print()
```

```
    return res

if __name__ == '__main__':
    gen_case(4, 4, 5, 4)
```

```
## experiment.py

import main
import gen_case
import utils
import copy
import time

def display_capacity(version_list, time_list):
    print('----------------------')
    for idx in range(len(version_list)):
        print(f'{version_list[idx]}: {time_list[idx]}')

def capacity_testing(version_list, size=8, round_num=5):
    out_cnt = 0
    time_list = [[], [], [], []] ## for capacity testing, element would
be(size, time)
    orig_version = copy.deepcopy(version_list)
    while out_cnt < 4:
        mines_num, hint_num = round(size*size*0.277777),
round(size*size*0.444444)
        time_sum_list = [0, 0, 0, 0]
        display_capacity(orig_version, time_list)

        for idx in range(round_num):
            inp = gen_case.gen_case(size, size, mines_num, hint_num)
            _, board, domains, constraints, constraint_list =
utils.parse_input(list(map(lambda x: int(x), inp.split(' '))))
            for ver_idx, ver in enumerate(version_list):
                if ver == 'Out':
                    continue

                sub_board, sub_domains = copy.deepcopy(board),
copy.deepcopy(domains)
                ## Start search
```

```
                st = time.process_time()
                sub_board, _ = main.solve(sub_board, sub_domains,
constraints, constraint_list, forward_check, ver)
                et = time.process_time()
                time_sum_list[ver_idx] += et-st

                if sub_board is None:
                    print(f"ERROR! {ver}")
                    exit(1)

        for idx in range(4):
            avg_time = time_sum_list[idx]/round_num
            if avg_time != 0:
                time_list[idx].append((size, avg_time))
                if avg_time > 1:
                    out_cnt += 1
                    version_list[idx] = 'Out'

        size += 1
    display_capacity(orig_version, time_list)

if __name__ == "__main__":
    ## Setting
    x_size, y_size = 6, 6
    mines_num, hint_num = 10, 16
    round_num = 500
    forward_check = True
    version_list = ['None', 'MRV', 'Degree', 'LCV']

    node_sum_list = [0, 0, 0, 0]
    time_sum_list = [0, 0, 0, 0]

    for idx in range(round_num):
        print('==================')
        print(f'Round {idx}:')
        inp = gen_case.gen_case(x_size, y_size, mines_num, hint_num)
        orig_map, board, domains, constraints, constraint_list =
utils.parse_input(list(map(lambda x: int(x), inp.split(' '))))
        utils.print_board(orig_map, board)
        for ver_idx, ver in enumerate(version_list):
```

```
            print(ver, end=' ')
            sub_board, sub_domains = copy.deepcopy(board),
copy.deepcopy(domains)
            ## Start search
            st = time.process_time()
            sub_board, node_num = main.solve(sub_board, sub_domains,
constraints, constraint_list, forward_check, ver)
            et = time.process_time()
            time_sum_list[ver_idx] += et-st

            if sub_board is None:
                print(f"ERROR! {ver}")
                exit(1)

            node_sum_list[ver_idx] += node_num
        print()

    print('======================')
    print(f'Board Size: ({x_size}, {y_size})')
    print(f'Mines Number: {mines_num}')
    print(f'Hints Number: {hint_num}')
    print(f'Sample Number: {round_num}')
    print()
    for ver_idx, ver in enumerate(version_list):
        print(f'{ver}: {node_sum_list[ver_idx]/round_num},
{time_sum_list[ver_idx]/round_num}s')
```