資工3B 0616026 游騰德

Introduction to AI

Programming Assignment #2

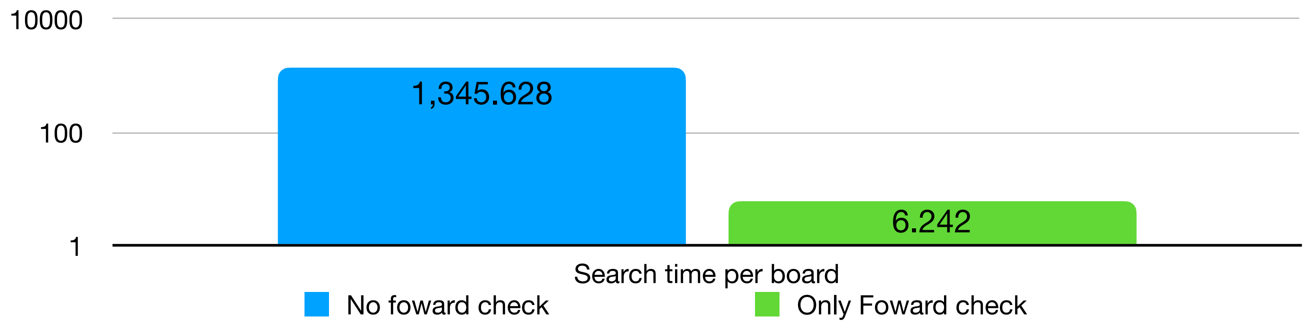
實驗與結果

演算法種類比較

表 1 為在一個 6*6 的盤面上，「完全不使用 Forward checking 和任何 Heuristics 」以及「只使用 Forward checking 」的時間比較。實驗隨機產生 100 組盤面進行測試，其中每個盤面均有 10 個地雷以及 16 個提示 (hints) ，實驗取 100 組測試結果之平均時間。

表 1 Y 軸為時間，使用對數刻度，單位為 10 毫秒 (10 ms)。

表 1



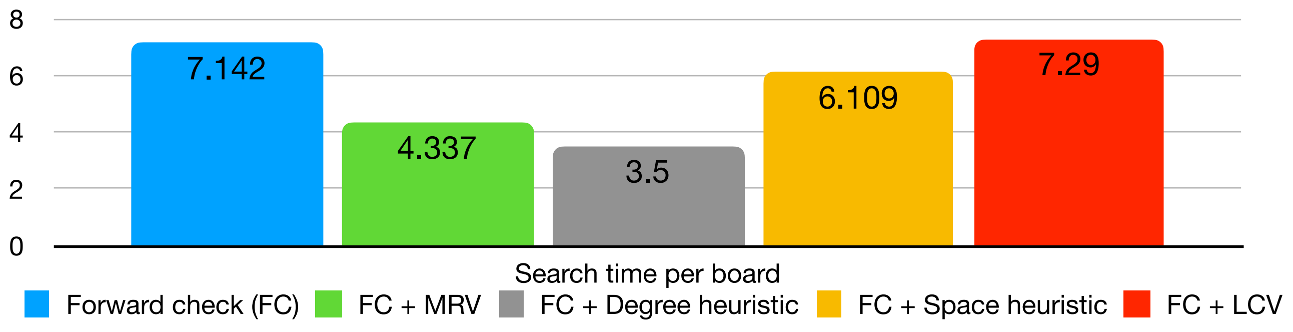
由表 1 可以看出，沒有 Forward checking 時，需要多出約 215 倍的搜尋時間，推測是由於 Forward checking 透過移除 Unassigned varaible 的 Domain 中的值而有效減少需展開的節點。另外，在實驗中我發現，不使用 Forward checking 時，搜尋每個盤面所花費的時間非常不定，少則數秒，多則數十分鐘，其變異相當巨大。

由於不使用 Forward checking 時，搜索所費時間太久，因此以下其他實驗若無特別標示，則皆預設有使用 Forward checking 。

表 2 為使用 Forward checking 後再加上「 MRV 」、「 LCV 」及各項 Heuristics 的時間比較，所測試的 Heuristics 有「 Degree heuristic 」以及「 Space heuristic 」。實驗同樣取 100 組測試結果之平均時間，隨機產生之盤面其條件與上組實驗相同。

表 2 Y 軸為時間，使用線性刻度，單位為 10 毫秒 (10ms)。 FC 表示 Forward checking 。

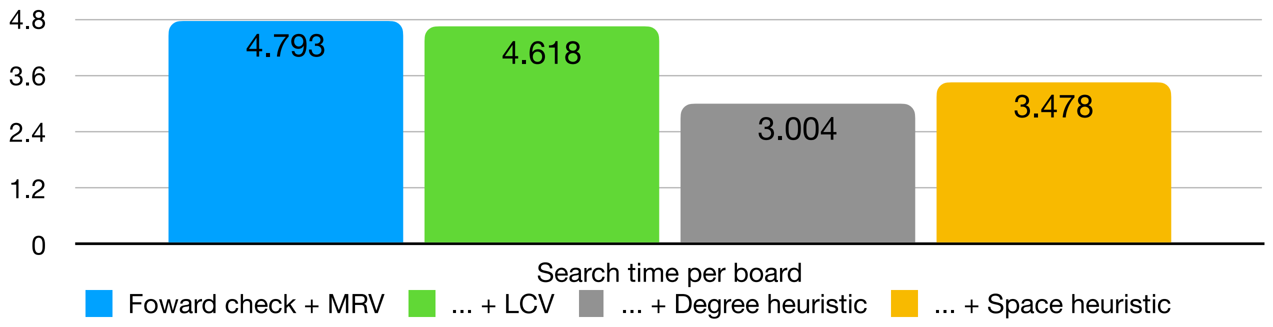
表 2



以表 2 最左方長條只使用 Forward checking 的時間當做對照組，可以發現只有在再加上 MRV 或 Degree heuristic 時，時間有較為明顯的減少，可推斷是因為 Space heuristic 和 LCV 為了減少節點而遞迴檢查所耗費的代價過大。為了解決這個問題，我認為應該先進行代價低的 Heuristic，再進行代價高的 Heuristic，以避免過多的遞迴檢查。

因此，經由表 2 的實驗的發現，接下來測試依序使用 Forward checking 加 MRV 後，再加上加「Degree heuristic」、「Space heuristic」或「LCV」，實驗由隨機產生之盤面 100 組，取測試結果之平均時間，結果如表 3 ，Y 軸為時間，單位為 10 毫秒 (10ms)。「…」表示「Forward check + MRV」。

表 3

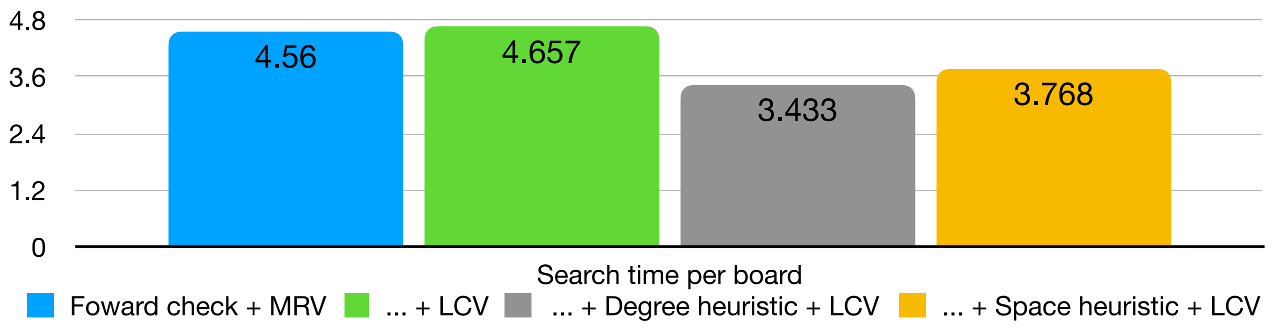


以表 3 左方長條使用 Forward checking 和 MRV 的時間當做對照組，可以發現依然只有在再加上 Degree heuristic 和 Space heuristic 時，時間有較為明顯的減少，表示 LCV 為了減少節點而遞迴檢查的耗費可能仍然過大。

Space heuristic 是我自己加入的 Heuristic，其和 Degree heuristic 的不同之處在於，Degree heuristic 選擇鄰近限制多的節點展開，Space heuristic 則是選擇鄰近「空的位置數量」少的節點優先展開。Space heuristic 的效能之所以稍微不如 Degree heuristic，我認為是因其需要在每次指定一個 Variable 之後，更新周圍「空的位置數量」，而 Degree heuristic 並不需要更新。然而，少部分時候 Space heuristic 會優於 Degree heuristic，例如在解 Spec 中的第一個 Sample board 時。

由於在上述實驗中使用 LCV 並無顯著改變，因此接下來嘗試在使用 Forward checking 、MRV 的基礎上，再單獨使用 LCV 或是同時加上「Degree heuristic」或「Space heuristic」，觀察結果差異。實驗中順序為先 Degree 或 Space heuristic 後再 LCV ，同樣取 100 組測試結果之平均時間，隨機產生之盤面其條件與第一組實驗相同，結果如表 4 ，Y 軸為時間，單位為 10 毫秒 (10ms)。「…」表示「Forward check + MRV」。

表 4



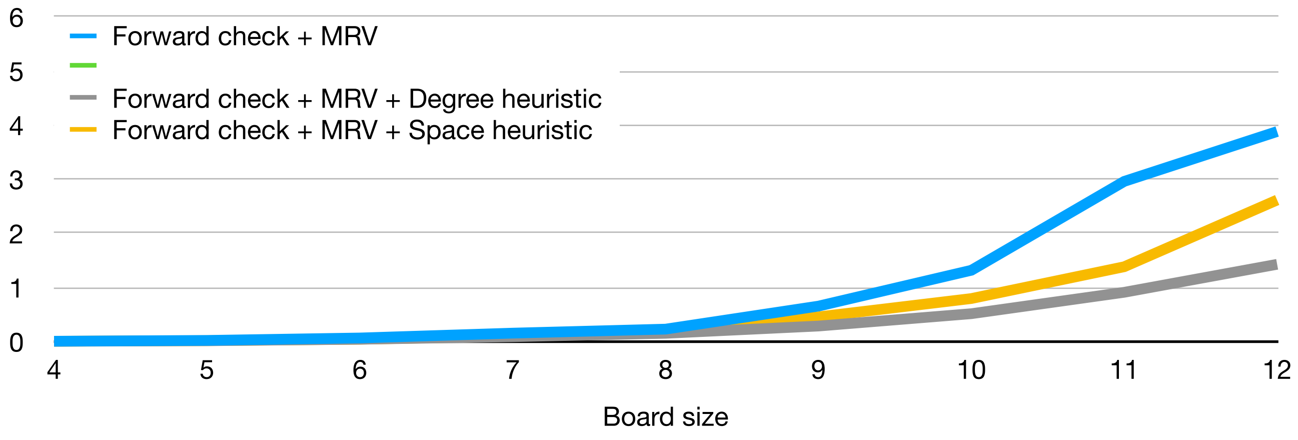
以表 4 左方長條使用 Forward checking 和 MRV 的時間當做對照組，單獨加上 LCV 確實沒有太大改變，甚至是更久，但是在若同時使用「 Degree heuristic 」或「Space heuristic 」都能明顯減少時間。然而，值得注意的是，如果去除表 4 中實驗組的 LCV ，結果都將會更快（可對照表 3 ）。

盤面大小比較

在盤面大小的實驗中，皆使用正方形盤面，每個盤面上的地雷總數佔全部位置的比例為 0.28 ，提示 (hints) 數量佔全部位置的比例為 0.44，測試每種盤面大小時，均隨機產生 100 組盤面進行測試，取 100 組測試結果之平均時間紀錄。

表 5 中，所有測試均以「 Forward checking + MRV 」為標準，加上兩種不同的 Heuristics ，觀察不同的變化。表 5 X 軸為盤面邊長，測試的盤面從 4*4 到 12* 12。 Y 軸為時間，使用線性刻度，單位為秒 (s)。

表 5



可以在表 5 中看到在盤面大小為 8*8 之後，有使用 Heuristic 的測試其實驗效果都較佳，與不使用 Heuristic 相比，使用 Heuristic 後所需時間大約為一半，此外在盤面大小越來越大時，使用 Degree heuristic 的結果又比使用 Space heuristic 的效果來的佳。接下來以以上測試條件加上「 LCV 」 ，觀察各組解開各個盤面大小所需的時間。實驗結果如表 6 。

表 6 X 軸為盤面邊長，測試的盤面從 4*4 到 12* 12。 Y 軸為時間，使用線性刻度，單位為秒 (s)。

表 6

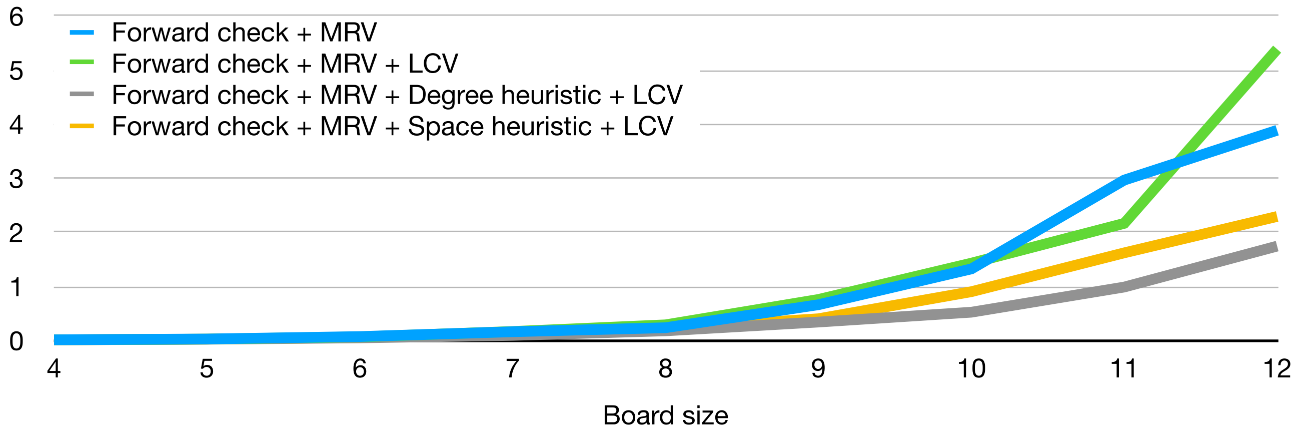
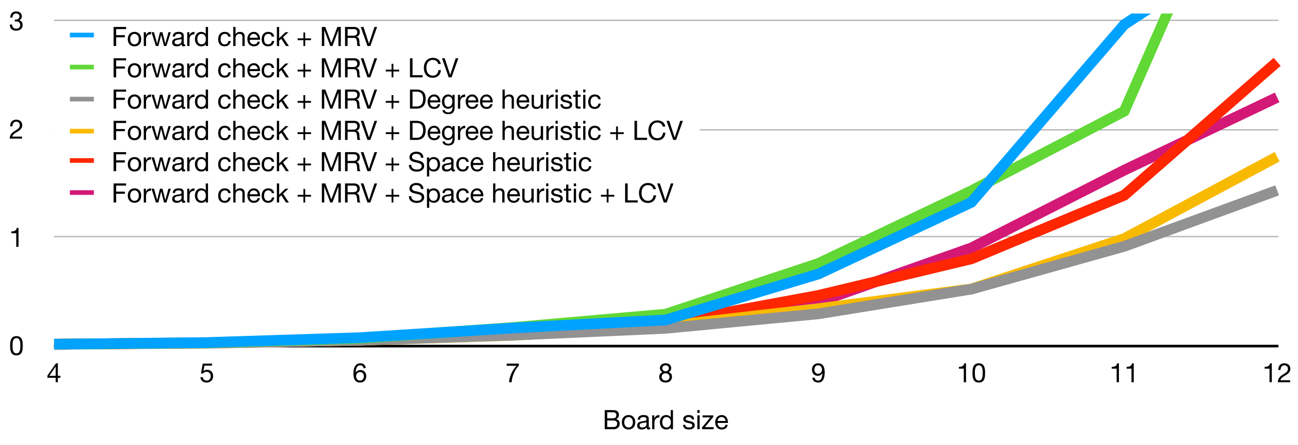


表 6 中可以看到，若不使用 Heuristics 而直接使用 LCV ，其測試結果要比不使用 LCV 時明顯來得差。若是同時使用 Heuristics 和 LCV ，所需時間與不使用 Heuristic 和 LCV 相比，大約為一半。

將表 5 與表 6 的資料疊合，並放大 Y 座標後製成表 7 ，觀察各組實驗使用 LCV 前後，所需時間的差異。

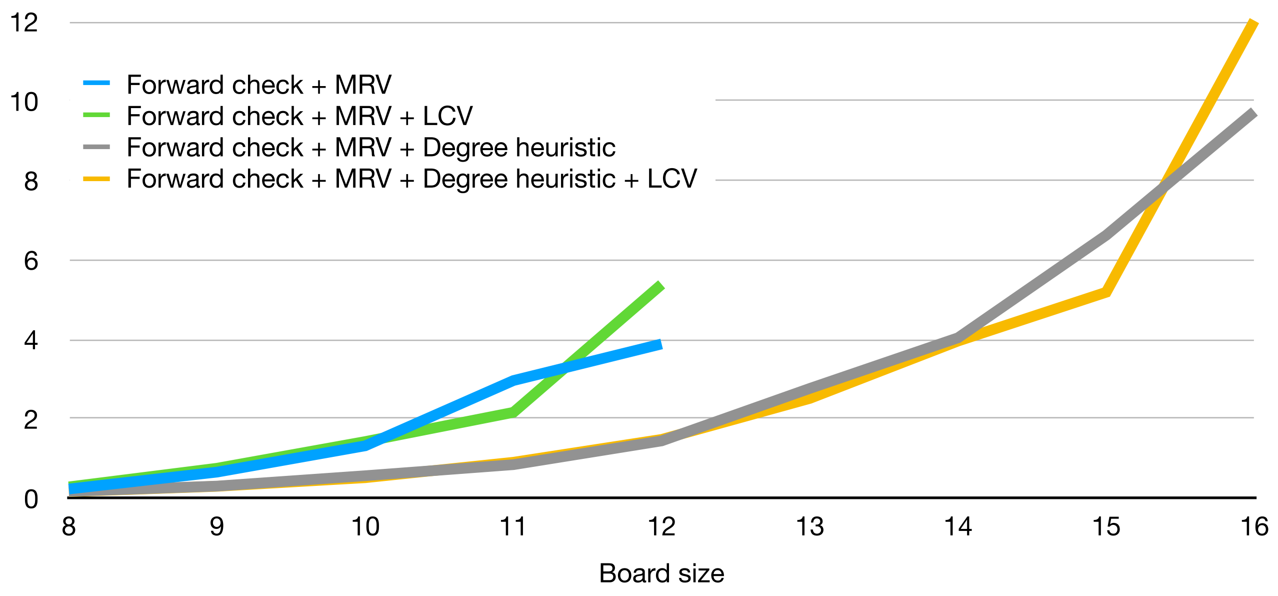
表 7



從表 7 中可以發現，六組測試中，大致兩兩互相疊合，表示使用 LCV 前後，時間差異並不明顯，因此我認為 LCV 對整體效能的影響並不顯著。

經由表 7 ，發現「Forward check + MRV + Degree heuristic」和「Forward check + MRV + Degree heuristic + LCV 」是效果最佳的兩組，因此以這兩種條件繼續測試更大的盤面。將盤面大小進一步跨大至 16*16 ，實驗結果如表 8 ， Y 軸為時間，單位為秒 (s)。

表 8



由表 8 可以看出，當盤面進一步擴大，兩者所需時間呈現明顯的指數級成長，但解出一個盤面所需的時間差大都落在 1 至 2 秒之間，可見其效能依然沒有巨大的區別。

另外，之所以只測試這兩種條件至盤面 16*16 ，是因為其他條件下，測試時間實在太長，我放了一整個晚上跑也沒有結果，因此只列出有結果的兩組。

所學

透過本次實驗我所學以下幾點：

1. 利用物件導向程式設計，以 Python 實作棋盤之 Framework。
2. 了解 Forward checking 、 LCV 、 MRV 以及各種 Heuristics 的運作原理、特性以及優缺點。
3. 透過比較不同的盤面大小來體會時間及空間複雜度對於演算法效能的影響。
4. 將實驗結果整理、列表並使人易於閱讀。

疑問與探討

1. 在 Spec 中有提到 「 … compare the performance (number of expanded nodes until solution)… 」 ，但我想說明一下，在我的報告中，之所以只比較時間而沒有比較展開的節點個數，是由於我認為如此一來，容易忽略演算法「為了減少節點所需付出的代價 (cost) 」 ，因此只單純考慮展開節點的個數是否變少，並用來衡量一個演算法的優劣，個人認為不夠全面。此外，為了減少時間上統計的變異，每組實驗我都隨機產生 100 組盤面（可參考 test.py 中的 gen_board() 函式）進行測試，取其平均，盡可能減少各種因素對時間結果的影響。
2. LCV 在整個實驗的過程中，其表現均不是相當好，可以說是可有可無。我試著去思考其原因，認為可能是因為當 LCV 在檢查受影響的 Value 時，其實也是類似重複的在做 Forward checking 中的 Consistency check。因此若是能更加清楚地去理解 LCV 和 Forward checking 之間的關係，嘗試把重複的步驟省去，我認為有機會增進 LCV 的效果。

未來發想

在盤面大小的實驗中，我發現各種演算法可能有其擅長的不同領域，也有各自不同地方的弱點，因此不一定特定一組演算法的搭配，在任何盤面大小皆有最佳的表現。此外，不同的演算法可能也僅適用於解開一個盤面的不同時期，例如：在前期只需要使用 Degree heuristic 來將限制多的位置優先展開，中期搭配 MRV 和 LCV 來選擇可以放置地雷的位置進行展延，末期使用 Space heuristic 來在自由度較少的位置進行最後的配置。以上只是一個舉例，主要是想表達：若是未來能做更多的實驗，仔細了解各種演算法適合的使用時機，再進行搭配運用，或許會有更佳的結果。

Appendix

Structure

- agent.py
- node.py
- board.py
- variable.py
- test.py

agent.py

```python
import time, copy
from variable import Assigned_Variable
from variable import Unassigned_Variable
from board import Board
from node import Node


class Agent():
    def __init__(self, forward_checking = False, mrv = False, heuristic = '',
lcv = False):
        self.fc = forward_checking
        self.mrv = mrv
        self.heuristic = heuristic
        self.lcv = lcv


    def search(self, b):
        # Initial unassigned variables
        unas_vrbls = []
        for j in range(b.size_y):
            for i in range(b.size_x):
                if b.hints[i][j] == -1:
                    variable = Unassigned_Variable((i, j), b, self.heuristic)
                    unas_vrbls.append(variable)

        # Create the root node
        root = Node([], unas_vrbls, None)

        # Initial frontier
        # Frontier: stack
        frontier = [root]
        while len(frontier):
            # Expand the deepest (most recent) unexpanded node
            cur_node = frontier.pop()

            # Return assigned variables when solution is found
            # or skip this node if not consistent
            consistency = cur_node.consistency_check(b)
            if consistency == 0 and len(cur_node.unas_vrbls) == 0:
                return cur_node.asgn_vrbls
            elif consistency < 0 or len(cur_node.unas_vrbls) == 0:
                continue

            # Forward checking (Optional)
            if self.fc:
                if cur_node.last_sltd_vrbl is not None:
                    if cur_node.forward_checking(b, 
cur_node.last_sltd_vrbl.position) != 0:
                        continue

            # Heuristics (Optional)
            sort_count = len(cur_node.unas_vrbls)
            # MRV
            if self.mrv:
                sort_count = cur_node.mrv(sort_count)
            # Degree heuristic or Space heuristic
            if self.heuristic == 'degree':
                sort_count = cur_node.degree_hrs(b, cur_node.last_sltd_vrbl, 
sort_count)
            elif self.heuristic == 'space':
                sort_count = cur_node.space_hrs(b, cur_node.last_sltd_vrbl, 
sort_count)
            # LCV
            if self.lcv:
                cur_node.lcv(b)

            # Choose the selected variable to expand
            sltd_vrbl = cur_node.unas_vrbls.pop()

            for value in sltd_vrbl.domain:
                # Create child node and append to parent
                child_asgn_vrbls = copy.deepcopy(cur_node.asgn_vrbls)
                child_asgn_vrbls.append(Assigned_Variable(sltd_vrbl.position, 
value))
```

```
                child_unas_vrbls = copy.deepcopy(cur_node.unas_vrbls)
                child = Node(child_asgn_vrbls, child_unas_vrbls, sltd_vrbl)
                cur_node.add_child(child)

                # Set frontier
                frontier.append(child)

        # Return empty list if no solution is found
        return []


if __name__ == '__main__':
    # Some examples and tests
    inputs_list = [
                    '6 6 10 -1 -1 -1 1 1 -1 -1 3 -1 -1 -1 0 2 3 -1 3 3 2 -1 -1
2 -1 -1 -1 -1 2 2 3 -1 3 -1 1 -1 -1 -1 1',
                    '6 6 10 -1 -1 -1 1 1 1 3 4 -1 2 -1 -1 2 -1 -1 -1 -1 -1 -1
-1 2 2 -1 2 1 2 -1 -1 1 -1 -1 1 -1 1 0 -1',
                    '6 6 10 -1 -1 -1 -1 -1 -1 -1 2 2 2 3 -1 -1 2 0 0 2 -1 -1 2
0 0 2 -1 -1 3 2 2 2 -1 -1 -1 -1 -1 -1 -1',
                    '6 6 10 -1 1 -1 1 1 -1 2 2 3 -1 -1 1 -1 -1 5 -1 5 -1 2 -1
5 -1 -1 -1 -1 2 -1 -1 3 -1 -1 -1 1 1 -1 0',
                  ]

    start_time = time.time()

    a = Agent(forward_checking = True, mrv = True, heuristic = 'space', lcv =
False)
    for inputs in inputs_list:
        b = Board(inputs)
        result = a.search(b)
        b.print_board(result)
        print()

    search_time = (time.time() - start_time) * 1000
    print(search_time)
```

node.py

```
import copy
from variable import Variable
from variable import Assigned_Variable
from variable import Unassigned_Variable
from board import Board


class Node():
    def __init__(self, asgn_vrbls, unas_vrbls, last_sltd_vrbl):
        self.asgn_vrbls = asgn_vrbls
        self.unas_vrbls = unas_vrbls
        self.last_sltd_vrbl = last_sltd_vrbl
        self.childs = []

    def add_child(self, child_node):
        self.childs.append(child_node)

    def board_status_string(self, b):
        # Return the string of board status of this node
        # Used for explored set
        status = []
        for i in range(b.size_x * b.size_y):
            status.append(9)
        for variable in self.asgn_vrbls:
            status[variable.position[1] * b.size_x + variable.position[0]] =
variable.assignment
        ret = ''
        for s in status:
            ret += str(s)
        return ret

    def all_arc_consistent_check(self, b):
        # Check all arc arc-consistent
        acc_count = 0
        for j in range(b.size_y):
            for i in range(b.size_x):
                position = (i, j)
                acc = b.arc_consistent_check(self.asgn_vrbls, position)
                if acc < 0:
                    return -1
                acc_count += acc
        return acc_count

    def consistency_check(self, b):
        # Check global constraint
        gcc_count = b.global_constraint_check(self.asgn_vrbls)
        if gcc_count < 0 or gcc_count > len(self.unas_vrbls):
            return -1

        # Check all arc arc-consistent
        all_acc_count = self.all_arc_consistent_check(b)
        if all_acc_count < 0:
            return -2

        return gcc_count + all_acc_count

    def forward_checking(self, b, position):
        fc_err = 0
        is_mine_pos = []
        no_mine_pos = []
        check_pos = b.around_position(position)

        # Find positions that can only be mine and positions that can not be 
mine
        for pos in check_pos:
            hint = b.hints[pos[0]][pos[1]]
            if hint > -1:
                lower_bound, upper_bound = 
b.forward_checking_limit(self.asgn_vrbls, pos)
                if lower_bound > hint or upper_bound < hint:
                    fc_err = -1
                    break
                elif lower_bound == hint:
                    no_mine_pos += b.around_position(pos)
                elif upper_bound == hint:
                    is_mine_pos += b.around_position(pos)

        # Remove values from the domain of varaibles whose position is in 
positions found above
        if not fc_err:
            for variable in self.unas_vrbls:
                pos = variable.position
                if (pos[0], pos[1]) in no_mine_pos:
                    try:
                        variable.domain.remove(1)
                    except:
                        pass
                if (pos[0], pos[1]) in is_mine_pos:
                    try:
                        variable.domain.remove(0)
                except:
                    pass
            if len(variable.domain) == 0:
                fc_err = -1
                break
        return fc_err

    def mrv(self, sort_bound):
        sort_count = 0

        # Categorize by remaining values
        groups = [[], []]
        for i in range(len(self.unas_vrbls) - sort_bound, 
len(self.unas_vrbls)):
            variable = self.unas_vrbls[i]
            if len(variable.domain) == 1:
                groups[0].append(variable)
                sort_count += 1
            else:
                groups[1].append(variable)

        # Sort by reverse order
        self.unas_vrbls = groups[1] + groups[0]
        sort_count = sort_bound if sort_count == 0 else sort_count
        return sort_count

    def degree_hrs(self, b, last_sltd_vrbl, sort_bound):
        sort_count = sort_bound
        if last_sltd_vrbl is not None:
            # Categorize by degree
            groups = [[], [], [], [], [], [], [], [], []]
            for i in range(len(self.unas_vrbls) - sort_bound, 
len(self.unas_vrbls)):
                variable = self.unas_vrbls[i]
                groups[variable.degree].append(variable)

            # Sort
            sort_vrbls = []
            for group in groups:
                sort_vrbls += group
                sort_count = len(group) if len(group) != 0 else sort_count
            self.unas_vrbls = self.unas_vrbls[0:len(self.unas_vrbls) - 
sort_bound] + sort_vrbls
```

```
        return sort_count

    def space_hrs(self, b, last_sltd_vrbl, sort_bound):
        sort_count = sort_bound
        if last_sltd_vrbl is not None:
            # Update space degree
            around = b.around_position(last_sltd_vrbl.position)
            for variable in self.unas_vrbls:
                if variable.position in around:
                    variable.degree -= 1

            # Categorize by space degree
            groups = [[], [], [], [], [], [], [], [], []]
            for i in range(len(self.unas_vrbls) - sort_bound,
len(self.unas_vrbls)):
                variable = self.unas_vrbls[i]
                groups[variable.degree].append(variable)

            # Sort by reverse order
            sort_vrbls = []
            for group in groups[::-1]:
                sort_vrbls += group
                sort_count = len(group) if len(group) != 0 else sort_count
            self.unas_vrbls = self.unas_vrbls[0:len(self.unas_vrbls) -
sort_bound] + sort_vrbls
        return sort_count

    def lcv(self, b):
        variable = self.unas_vrbls[-1]

        # Only for variables whose domains are still [0, 1]
        if len(variable.domain) == 2:
            limit_pos_counts = []
            domain = [0, 1]

            # Try both values
            for value in domain:
                new_asgn_vrbls = copy.deepcopy(self.asgn_vrbls)
                new_asgn_vrbls.append(Assigned_Variable(variable.position,
value))
                new_board = b.current_board(new_asgn_vrbls)

                # Foward check to calculate count of ruled out values
                fc_err = 0
                limit_pos = []
                check_pos = b.around_position(variable.position)
                for pos in check_pos:
                    hint = b.hints[pos[0]][pos[1]]
                    if hint > -1:
                        lower_bound, upper_bound = 
b.forward_checking_limit(new_asgn_vrbls, pos)
                        if lower_bound > hint or upper_bound < hint:
                            fc_err = -1
                            break
                        elif lower_bound == hint or upper_bound == hint:
                            around = b.around_position(pos)
                            for a in around:
                                if a not in limit_pos and new_board[a[0]]
[a[1]] == -1:
                                    limit_pos.append(a)
                if fc_err != -1:
                    limit_pos_counts.append(len(limit_pos))
                else:
                    limit_pos_counts.append(-1)

            # Switch domain if need
            if limit_pos_counts != [-1, -1]:
                if limit_pos_counts[1] < limit_pos_counts[0]:
                    variable.domain = [1, 0]

            # Remove value from domain if need
            if limit_pos_counts[1] == -1:
                try:
                    variable.domain.remove(1)
                except:
                    pass
            if limit_pos_counts[0] == -1:
                try:
                    variable.domain.remove(0)
                except:
                    pass
```

board.py

```
import copy
from variable import Variable
```

```
class Board():
    def __init__(self, inputs):
        args = inputs.split()
        self.size_x = int(args[0])
        self.size_y = int(args[1])
        self.mines_count = int(args[2])
        self.hints = []

        idx = 3
        for i in range(self.size_y):
            for j in range(self.size_x):
                if i == 0:
                    self.hints.append([])
                self.hints[j].append(int(args[idx]))
                idx += 1

    def available_position(self, position):
        # Returns true if the given position is available on this board
        return 0 <= position[0] < self.size_x and 0 <= position[1] <
self.size_y

    def around_position(self, position):
        # Returns a list of available postions around the given position
        x = position[0]
        y = position[1]
        psb_pos = [(x-1, y-1), (x, y-1), (x+1, y-1),
                   (x-1, y),             (x+1, y),
                   (x-1, y+1), (x, y+1), (x+1, y+1)]
        around = []
        for pos in psb_pos:
            if self.available_position(pos):
                around.append(pos)
        return around

    def current_board(self, asgn_vrbls = []):
        # Return a list of current board status
        current = copy.deepcopy(self.hints)
        for variable in asgn_vrbls:
            x = variable.position[0]
            y = variable.position[1]
            if variable.assignment == 0:
                current[x][y] = -2
            elif variable.assignment == 1:
                current[x][y] = -3
        return current

    def print_board(self, asgn_vrbls = []):
        # Print the current board status
        # _     : Unassigned
        # |     : Assigned no mine
        # *     : Assigned mine
        # [0-8] : Hint
        current = self.current_board(asgn_vrbls)
        for j in range(self.size_y):
            for i in range(self.size_x):
                current[i][j] = '_' if current[i][j] == -1 else current[i][j]
                current[i][j] = '|' if current[i][j] == -2 else current[i][j]
                current[i][j] = '*' if current[i][j] == -3 else current[i][j]
                print(current[i][j], end=" ")
            print()

    def forward_checking_limit(self, asgn_vrbls, position):
        # Returns lower and upper bounds of the sum of the given position
        current = self.current_board(asgn_vrbls)
        lower_bound = 0
        upper_bound = 0
        around = self.around_position(position)
        for a in around:
            if current[a[0]][a[1]] == -3:
                lower_bound += 1
                upper_bound += 1
            elif current[a[0]][a[1]] == -1:
                upper_bound += 1
        return lower_bound, upper_bound

    def arc_consistent_check(self, asgn_vrbls, position):
        # Returns difference of hint and mines count
        x = position[0]
        y = position[1]
        if not self.available_position(position):
            return 0
        if self.hints[x][y] == -1:
            return 0
        current = self.current_board(asgn_vrbls)

        bombs_count = 0
        around = self.around_position(position)
        for a in around:
            if current[a[0]][a[1]] == -3:
                bombs_count += 1

        return self.hints[x][y] - bombs_count

    def global_constraint_check(self, asgn_vrbls):
        # Returns difference of tolal mines and current assigned mines count
        mines_count = 0
        for variable in asgn_vrbls:
            if variable.assignment == 1:
                mines_count += 1

        return self.mines_count - mines_count
```

variable.py

```
class Variable():
    def __init__(self, position):
        self.position = position

class Assigned_Variable(Variable):
    def __init__(self, position, assignment):
        super(Assigned_Variable, self).__init__(position)
        self.assignment = assignment

class Unassigned_Variable(Variable):
    def __init__(self, position, b, heuristic):
        super(Unassigned_Variable, self).__init__(position)
        self.domain = [0, 1]
        self.degree = -1

        # Initial value of degree if need
        if heuristic != '':
            current = b.current_board()
            around = b.around_position(self.position)
            degree = 0
            for a in around:
                if heuristic == 'degree' and current[a[0]][a[1]] > -1:
                    degree += 1
                elif heuristic == 'space' and current[a[0]][a[1]] == -1:
                    degree += 1
            self.degree = degree
```

test.py

```
import time, random
from board import Board
from agent import Agent

FFNF = Agent(forward_checking = False, mrv = False, heuristic = '', lcv = 
False)
TFNF = Agent(forward_checking = True, mrv = False, heuristic = '', lcv = 
False)

TTNF = Agent(forward_checking = True, mrv = True, heuristic = '', lcv = False)
TFDF = Agent(forward_checking = True, mrv = False, heuristic = 'degree', lcv = 
False)
TFSF = Agent(forward_checking = True, mrv = False, heuristic = 'space', lcv = 
False)
TFNT = Agent(forward_checking = True, mrv = False, heuristic = '', lcv = True)

TTDF = Agent(forward_checking = True, mrv = True, heuristic = 'degree', lcv = 
False)
TTSF = Agent(forward_checking = True, mrv = True, heuristic = 'space', lcv = 
False)
TTNT = Agent(forward_checking = True, mrv = True, heuristic = '', lcv = True)

TTDT = Agent(forward_checking = True, mrv = True, heuristic = 'degree', lcv = 
True)
TTST = Agent(forward_checking = True, mrv = True, heuristic = 'space', lcv = 
True)


def around_position(position, board_x, board_y):
    around = []
    if position == 0:
        around = [
                                                        position+1,
                                        position+board_x, position+board_x+1]
    elif position+1 == board_x:
        around = [
                   position-1,
                   position+board_x-1, position+board_x]
    elif position == board_x*(board_y-1):
        around = [                    position-board_x, position-board_x+1,
                                                         position+1,
                  ]
    elif position+1 == board_x*board_y:
        around = [ position-board_x-1, position-board_x,
                   position-1,
                  ]
    elif 0 < position < board_y:
        around = [
                   position-1,                           position+1,
                   position+board_x-1, position+board_x, position+board_x+1]
    elif position % board_x == 0:
        around = [                    position-board_x, position-board_x+1,
                                                         position+1,
                                      position+board_x, position+board_x+1]
    elif (position+1) % board_x == 0:
        around = [ position-board_x-1, position-board_x,
                   position-1,
                   position+board_x-1, position+board_x]
    elif board_x*(board_y-1) < position < board_x*board_y:
        around = [ position-board_x-1, position-board_x, position-board_x+1,
                   position-1,                           position+1,
                  ]
    else:
        around = [ position-board_x-1, position-board_x, position-board_x+1,
                   position-1,                           position+1,
                   position+board_x-1, position+board_x, position+board_x+1]
    return around

def gen_board(board_x, board_y, mines, hints):
    # Randomly generate a new board
    board_inputs = []
    positions = []
    for i in range(board_x * board_y):
        board_inputs.append(-1)
        positions.append(i)

    # Select positions
    sltd_pos = random.sample(positions, mines + hints)
    mine_pos = sltd_pos[0:mines]
    hint_pos = sltd_pos[mines:]

    for pos in hint_pos:
        around = around_position(pos, board_x, board_y)
        mines_count = 0
        for a in around:
            if a in mine_pos:
                mines_count += 1
        board_inputs[pos] = mines_count

    # Generate inputs string
    board_inputs_string = str(board_x) + ' ' + str(board_y) + ' ' + str(mines)
    for i in board_inputs:
        board_inputs_string += ' ' + str(i)
    return board_inputs_string

def algorithm_test(board_counts, board_x, board_y, mines, hints, agents):
    print('==========================================================')
    print('Algorithm Tests')
    print()
    print('Test with   :', board_counts, 'boards')
    print('Board size  :', board_x, '*', board_y)
    print('Mines       :', mines)
    print('Hints       :', hints)
    print()
    print('FwCheck\tMRV\tHeurs\tLCV\tTime per board (ms)')
    print('----------------------------------------------------------')

    inputs_list = []
    for i in range(board_counts):
        inputs_list.append(gen_board(board_x, board_y, mines, hints))

    for a in agents:
        start_time = time.time()
        for inputs in inputs_list:
            b = Board(inputs)
            a.search(b)
        search_time = (time.time() - start_time) * 1000
        print('{}\t{}\t{}\t{}\t{}'.format(a.fc, a.mrv, a.heuristic if
a.heuristic != '' else 'None', a.lcv, search_time / board_counts))
    print()

def board_size_test(board_counts, min_board_size, max_board_size, agents):
    print('===========================================================')
    print('Board Size Tests')
    for a in agents:
        print('========')
        print('Test with       :', board_counts, 'boards per board size')
        print('Mines ratio     :', mines_ratio)
        print('Hints ratio     :', hints_ratio)
        print('Using algorithm : FwCheck\tMRV\tHeurs\tLCV')
        print('                  {}\t\t{}\t{}\t{}'.format(a.fc, a.mrv, 
a.heuristic if a.heuristic != '' else 'None', a.lcv))
        print()
        print('Board size\tTime per board (ms)')
        print('---------------------------------------------------------')

        for board_size in range(min_board_size, max_board_size+1):
            inputs_list = []
            mines = int(board_size * board_size * mines_ratio)
            hints = int(board_size * board_size * hints_ratio)

            for i in range(board_counts):
                inputs_list.append(gen_board(board_size, board_size, mines, 
hints))

            start_time = time.time()
            for inputs in inputs_list:
                b = Board(inputs)
                a.search(b)
            search_time = (time.time() - start_time) * 1000

            print('{: 3} *{: 3}\t{}'.format(board_size, board_size, 
search_time / board_counts))

        print()

if __name__ == '__main__':
    # Settings
    board_counts = 100

    #### For algorithm test
    board_x = 6
    board_y = 6
    mines = 10
    hints = 16
```

```
agents = [TTDF]
# agents = [FFNF, TFNF]
# agents = [TFNF, TTNF, TFDF, TFSF, TFNT]
# agents = [TTNF, TTDF, TTSF, TTNT]
# agents = [TTNF, TTDT, TTST, TTNT]

#### For board size test
min_board_size = 4
max_board_size = 12
mines_ratio = 0.28
hints_ratio = 0.44
agents = [TTNF, TTNT, TTDF, TTSF, TTDT, TTST]


# Tests
algorithm_test(board_counts, board_x, board_y, mines, hints, agents)
board_size_test(board_counts, min_board_size, max_board_size, agents)
```