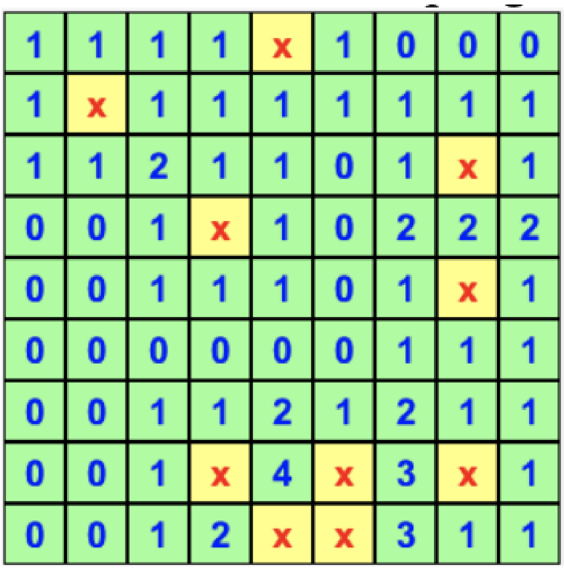
Intro to AI lab3 - MineSweeper with Logic Agent



Author Information

- National Chiao Tung University, 0712238
- Yan-Tong Lin
- E-mail: 0312fs3@gmail.com

Task description

- develop a Minesweeper AI based on (propositional) logic agent
- develop the environment for the agent

Proposed Algorithm

- as described in the assignment spec.

Implementation

General

- Programming Language: python
- IDE: jupyter notebook
- Version Control: git and github

Game control module

- Board(n,m,nmine) generate a random board corresponding to the parameter
- board.get_start(start=sqrt) return a list of starting safe positions
- LogicAgent.solve(board) will solve the problem for the board

OOP design (logic inference part)

- Variable/Literal(var)

  - a class
    - pos(position of variable, pair of (x,y)
    - T(mine or safe)
  - Clause is set/frozenset of variable $\Longrightarrow$ Var need to be hashable
    - hash with $x*1000*10+y*10+T$ to avoid collision
      - a slight improvement on performance than hash(repr)
    - use repr in python to print it
- Clause(cls)
  - not a class
  - direct use python set/frozenset
  - KB is set of clause $\Longrightarrow$ Clause need to be hashable
  - frozenset is hashable but immutable
  - set is mutable unhashable
  - store in KB as frozenset, and change as set
- Knowledge Base(KB)
  - treat as a ADT(abstract data type)
  - define insert(), match()… key function for it
  - used by agent to do inference

logic agent

- singleton clause meams ground truth
- do matching until it get a singleton clause (or fail with MATCHING LIMIT EXCEED)
- apply changes corresponding to the singleton clause
- algorithm detail is as described in assignment descrition
- some setting for **hyperparams**:
  - CHECK_SUB_WHEN_INSERT:
    - this can reduce the insert complex to O(|KB0|) from O(|KB|+|KB0|)
    - but may not be a good trade since max_kb/kb_inserted records indicate this can cause the kb to grow faster
    - however, checking subsumption before matching should relieve a little(since the most time spent correspodint to |KB| is matching)
  - MAX_ITER = 3:
    - max iteration of matching permmited
    - 3 is enough for solving most puzzles while keeping running time acceptable
  - MATCH_SIZE_LIMIT = 2:
    - max size for one side of matching
    - set to 2 can
    - extra experiment can be done on this parameter
  - GLOBAL_LIMIT = 1000:
    - when to add in global limit
    - was set to 5000 but cause heavy time comsumption
      - a small experiment showed time is influenced a lot by this factor
- No time for cross validation for limited time
  - may not be the optimal set of hyper params

Experiment Results and Discussion

Evaluation Criterion

- definition
  - n, m : board width and length
  - $nmine$: # of mine
  - T : testcases
- **grid success(win) rate**
  - $\sum|KB0|Tnm$
  - percentage of KB0(inferenced ground truth) to all grids
- **board success(win) rate**
  - # of success boardsT
  - percentage that the agent completes the whole game
- **max KB size(max_kb)**
  - how many clauses is in the KB of agent at the same time(count on insert)
    - the maximum of this value indicate the max memory usage during the process
  - a measurement for memory used
- **inserted clauses(inserted_cls)**
  - how many clauses is put into KB(count on insert)
  - a evaluation of time used
    - but since matching step takes $O(|KB|^2)$ should be the dominating term
    - this may be positive related to execution time but not linearly

Winrate / Board Size

- compare (grid) success rate on different board setting
- evaluate by 100 random generated board for each setting

	(9,9), 10	(16,16), 25	(32,16), 99
Grid Winrate	0.9888	0.9989	0.9779
Board Winrate	0.93	0.92	0.48

Discussion

- **observations**
  - board winrate drop dramatically in bigger map
  - grid winrate is similar for 3 cases
- about **ambiguity**
  - both winrate is associated with “ambiguity”(multiple solution of a board),
    - ambiguity $\Longrightarrow$ logic agent cannot get a deterministic result
  - the ambiguity case happens in the corners in most cases
    - so the grid winrate is similar between groups
- about **global constraint**
  - I guess the lower success rate of big map is caused by the global constraint is added too late compared to small boards

Resources Spent / Board Size

- my language is python3, so execution is slow :(
- evaluate by 100 random generated board for each setting(average)

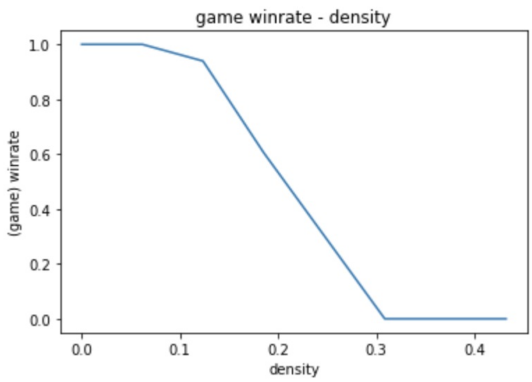
	(9,9), 10	(16,16), 25	(32,16), 99
Runtime	0.56(sec)	2.19(sec)	8.48(sec)
max_kb	194.13	338.65	1056.04
inserted_cls	343.48	762.98	2861.54

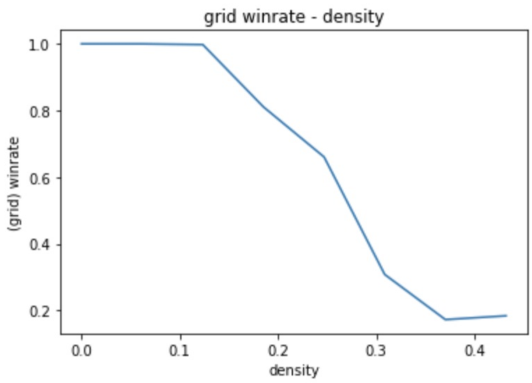
Discussion

- runtime is positively related to boardsize (~$O(n*m)$)
- analytically, runtime should be $O(|KB|^2)$
    - max_kb size show a trend for this
    - using avg_kb may be more useful for time evaluation then max_kb
- notice that inserted kb show linearly dependecy to runtime on the cases (16,16), 25 and (32,16), 99
    - I geuss it was that inference time is smaller compared to inserting time
- Future Work
    - conduct experiment to show the **ratio of time spent on inference and insertion**
    - add additional statistic avg_kb, matching_size(how match cls generated from matching)…

Winrate / Mine Density

- compare success rate on different mine density setting
    - (9,9) board with different number of mines with range(0,35, +5)





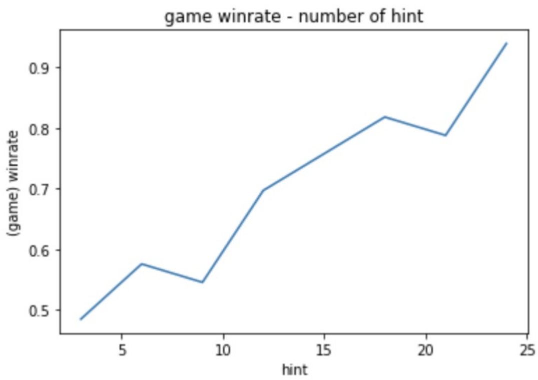
Discussion

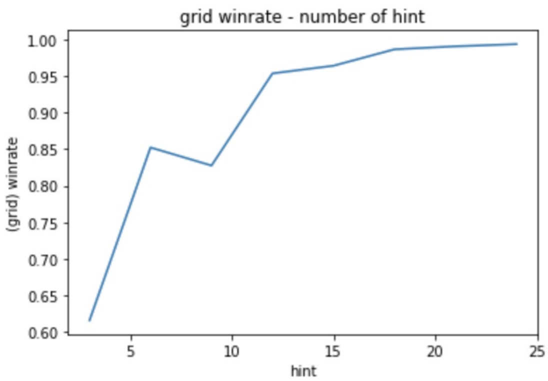
- Observation
    - higher density => lower winrate

- the function seems to be in the shape of logistic function!
- My Conjecture
  - higher mine density can cause
    - less hint cells
    - higher ambiguity
  - so higher density results in lower winrate
  - maybe increase MAX_ITERATION_LIMIT on matching can improve winrate when density is high

Winrate / Number of Starting Hints

- compare success rate on different number of starting hints
  - (9,9), 15 board with different number of hints given with range(3,27, +3)
- take (9,9), 15 because the winrate is close to about 50% given 9 hint according to previous experiments



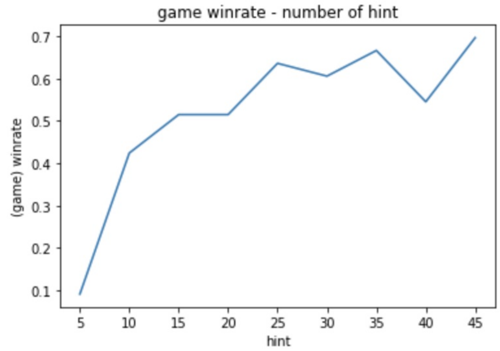


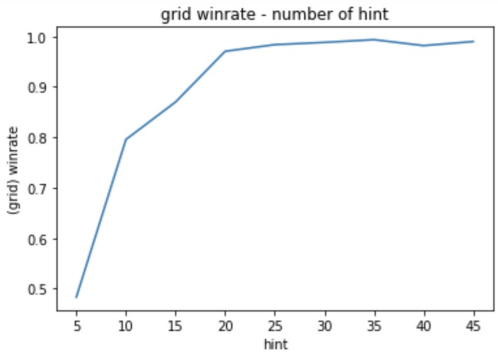
Discussion

- Observation:
  - winrate acends as number of given hints increases
  - shape of linear function(game winrate)
  - shape of logarithm function (grid winrate)
- My conjecture:
  - more information means more success rate without a doubt
  - very high grid winrate (>95%) when given >12 hints
    - note that the grid has 81 cells
    - 12 hint means ~1/7 cell is given
    - a cell has 8 neighbor and 72 non-neighbor cells
    - so almost every cell have one neighbor revealed intially(in probability)
    - I think this is one cause of high winrate
  - winrate cease to grow at around 10(~9), which is closed to n(board width)

Redo in 16*16

- for my conjecture, I redo the# of hint experiments to verify ny guess
- setting
  - 16*16 grid
  - 50 mines
  - starting hint number = range(5, 50, +5)





- the setting of nmine is calcuted by density ~0.2 for the experiment on density
  - it shares similar winrate graph as the graph of 9*9 board with same density!
- grid winrate shows shape like logorithm function again
- for 16*16 board with similar density, 20 hints is enough for 95%up (grid) winrate
- reach 90%up winrate for #hint>16
  - sqrt(grid_size) is enough for success of our logic agent
  - the setting given by TA/Teacher is reasonable!

Bonus Discussion

[Optional / Extra Credits]
These are for discussion only; no implementation/experiments required.
 How to use first-order logic here?
 Discuss whether forward chaining or backward chaining applicable to this problem.
 Propose some ideas about how to improve the success rate of “guessing” when you want to proceed from a
“stuck” game.
 Discuss ideas of modifying the method in Assignment#2 to solve the current problem.

How to apply FOL

- How to inference with FOL is not covered here(please refer to lecture notes and other references)
- We decribe the axioms for the task here

Definitions of symbols

- let B[i][j] be the square at the i-th row, j-th column
- we define M(i,j) as an indicator that there is a mine atB[i][j]
- we define A(i,j,k) as an indicator that there isexact k mines in the neighborhood ofB[i][j]

Axioms for the inference engine

- M(i, j) is well defined itself
- For A(i, j, k) with domain of k is {0, 1, 2, …, 8}, we need to define the evaluation of A(i, j, 0) to A(i, j, 8)
- Take domain = {0, 1, 2} as example, domain with higher maximum of k can done similarly
  - rule of global constraint
  - rule of A(i, j, 0)
  - rule of A(i, j, 1)
  - rule of A(i, j, 2)

$$\exists i_1\ \exists j_1\ \exists i_2\ \exists j_2\ (M(i_1,j_1) \wedge M(i_2,j_2) \wedge (i_1 \neq 1 \vee j_1 \neq 1) \wedge (i_2 \neq 1 \vee j_2 \neq 1) \wedge (i_1 \neq i_2 \vee j_1 \neq j_2))$$
$$\forall i_1\ \forall j_1\ \forall i_2\ \forall j_2\ \forall i'\ \forall j'\ (M(i_1,j_1) \wedge M(i_2,j_2) \wedge (i_1 \neq i' \vee j_1 \neq j') \wedge (i_2 \neq i' \vee j_2 \neq j')) \rightarrow \neg M(i',j')$$
$$\forall i'\ \forall j'\ (\forall i\ \forall j\ (|i-i'| \leq 1) \wedge (|j-j'| \leq 1) \wedge \neg(i=i' \wedge j=j') \rightarrow \neg M(i,j)) \rightarrow A(i',j',0)$$
$$\forall i_1\ \forall j_1\ \forall i'\ \forall j'\ (M(i_1,j_1) \wedge (|i_1-i'| \leq 1) \wedge (|j_1-j'| \leq 1) \wedge \neg(i_1=i' \wedge j_1=j') \wedge$$
$$(\forall i_2\ \forall j_2\ ((|i_1-i'| \leq 1) \wedge (|j_1-j'| \leq 1) \wedge (i_1 \neq i_2 \vee j_1 \neq j_2)) \rightarrow M(i_2,j_2))) \rightarrow A(i',j',1)$$
$$\forall i_1\ \forall j_1\ \forall i_2\ \forall j_2\ \forall i'\ \forall j'\ (M(i_1,j_1) \wedge M(i_2,j_2) \wedge (|i_1-i'| \leq 1) \wedge (|j_1-j'| \leq 1)$$
$$\wedge (|i_2-i'| \leq 1) \wedge (|j_2-j'| \leq 1) \wedge \neg(i_1=i' \wedge j_1=j') \wedge \neg(i_2=i' \wedge j_2=j') \wedge \neg(i_1=i_2 \wedge j_1=j_2))$$
$$\rightarrow A(i',j',2)$$

- reference: https://www2.cs.duke.edu/courses/spring06/cps102/notes/sweeper.pdf

Forward/backward chainning applicable?

- We “cannnot” apply FC or BC here with original KB “directly”
- Since FC and BC require all clauses to be “horn clause”
  - horn clause: clauses with exactly one or no positive literal
- The task’s KB is made of all positive or all negtive clauses
- However, we can make all clauses horn by negating Mine(x,y) to Safe(x,y) or conversely
- But on doing this, there will beO(|clause|) horn clause generate by a single clause
- The complexity is linear to |KB|, but now size KB is actually bigger(O(# of clause * average|clause|))

How to improve winrate by guessinng when stuck

- When it comes to guessing, it means that:
  - We cannot make a singleton clause out of limited resolutions
- We must do backtraching search in this situation to try to get a solution
- May use proof contradiction
  - add not Mine(x,y)to KB0 and do resolution + pairwise matching
  - see if there is empty clause(negative tautology) generated
  - if there is, Safe(x,y) is True
- Can use similar concept to CSP solver:
  - MRV
    - Is of no use here, since {0,1} is possible for all non-singleton clasues
  - Degree
    - I think degree heuristic can help a lot
    - Since adding a big degree guess to tmp KB0 can do a lot of resolutions
  - LCV

        - If the “least constraint” can be well-defined, LCV is going to help in my opinion
        - a suitable definition of “constraint” can be the KB’s degree of freedom
            - after assgin the guess, want the number of singleton term to be smaller
            - or define a fuction like $f(KB) = \sum w(i) * \#(i)$ where i is the size of a term
- Another why is to do probabilistic estimation on the chance of there is a mine in certain grid

How to modify Assignment 2 to solve iteractive version

- we can view the known constraints(hints) as KB
- do backtracking search as we do in programming assignment 2
- for the case that a guess is necessary, can use some approach to estimate the probability of a mine is in the position
- add constraint when accessing a new safe position(insert to KB)
- with more constraint the domain of variables will change accordingly

Conclusion

- Propositional Logic Agent alone is enough to inference a great percentage of MineSweeper game with smaller board
- We analysis the following:
    - the resources spent on different game setting
    - winrate on different board size
    - winrate on different mine density
    - winrate on different numbers of starting hints
- Future Works
    - There is a lot of method that worth trying in the bonus discussion area.
    - more statistics
        - time on inference(matching), time on insert
        - kb size changes(average, or as graph to how it goes throughout the process of game)
    - experiment on hyperparams
        - maxtching limit of clause size
        - when to put in global constraint
        - maximal iteration constraint
    - different implementation
        - FOL
        - Propositional logic solver with enumeration of horn clause + FC
            - forward chainning is better than BC since we want the whole result
        - CSP solver
        - add Guessing Stategy
            - calculate Prob
            - calculate heristics
    - I have made most of my conjecture by guessing, should provide more mathematical insights to it!

Appendix A - Code

lab3_2.py

- core of algorithm and resource/setting exp

- version 2(version 1 was bugged)

```
# lab3-2.py
# author = Yan-Tong Lin
# !/usr/bin/env python
# coding: utf-8

import numpy as np
import random
import itertools
from copy import deepcopy
import math

# hyper params
MATCH_ITER_LIMIT = 3 # n => n^2 => n^4 => n^8 => n^16, or can use CLS_LIMIT
MATCH_SIZE_LIMIT = 2 # subset with size <= limit can match wih other
GLOBAL_LIMIT = 1000 # when to add global limit, when C(n,m) is less than global limit
CLS_LIMIT = 1000
MAX_GRID = 500 # check var, use constant there
CHECK_SUB_ON_INSERT = True

# function to print a 2-D board
def printA(A):
    n = len(A)
    # m = len(A[0])
    for i in range(n):
        print(A[i])
    print()

class Board:
    B = None #board
    H = None #hint
    n, m, nmine = None, None, None

    #@staticmethod
    def inrange(self, i, j):
        # print(i, j)
        return i >= 0 and i < self.n and j >= 0 and j < self.m

    def __init__(self, n, m, nmine): # return a list of starting hints
        self.n = n
        self.m = m
        self.nmine = nmine
        all_coordinate = [(i, j) for j in range(self.m) for i in range(self.n)]
        # print(all_coordinate)
        mine_pos = random.sample(all_coordinate, nmine)
        self.B = [[0]*self.m for i in range(self.n)]
        self.H = [[0]*self.m for i in range(self.n)]
        for p in mine_pos:
            self.B[p[0]][p[1]] = 1
        #printA(self.B)
        for i, j in itertools.product(range(self.n), range(self.m)):
            if(self.B[i][j] == 1):
                continue
            for dx, dy in itertools.product(range(-1,2,1), range(-1,2,1)):
                if(not (dx==0 and dy==0) and self.inrange(i+dx, j+dy)):
                    self.H[i][j] += self.B[i+dx][j+dy]
        #printA(self.H)
        return

    # for printing
    def __repr__(self):
        n = self.n
        m = self.m
```

```
        ret = ""
        for i in range(n):
            ret += str(self.B[i])
            ret += "\n"
        return ret

    def get_start(self, nstart = None):
        # get starting hints(starting safe positions)
        if(nstart == None):
            nstart = int(np.sqrt(self.n*self.m))
        hint_coordinate = list(filter(lambda x: self.B[x[0]][x[1]]==0 ,[(i, j) for j
        # printA(self.B)
        # print(hint_coordinate)
        start_pos = random.sample(hint_coordinate, nstart)
        #start = [[-1]*self.m for i in range(self.n)]
        #for p in start_pos:
        #    start[p[0]][p[1]] = self.H[p[0]][p[1]]
        return start_pos

    # query
    def q(self, i, j): #return -1 if is mine, return number if is hint
        return -1 if self.B[i][j] == 1 else self.H[i][j]
    def q(self, p): #return -1 if is mine, return number if is hint
        return -1 if self.B[p[0]][p[1]] == 1 else self.H[p[0]][p[1]]




class VAR:
    pos = None
    T = None # true / false, Mine/ Safe

    def __init__(self, pos, T):
        self.pos = pos
        self.T = T
    def __repr__(self):
        if self.T :
            return "M(%d,%d)"%(self.pos[0], self.pos[1])
        else:
            return "S(%d,%d)"%(self.pos[0], self.pos[1])

    def __eq__(self, rhs):
        if(isinstance(rhs, VAR)):
            return (self.pos == rhs.pos and self.T == rhs.T)
        else:
            return False
    # collision?
    def __hash__(self):
        return hash(self.pos[0]*1000*10+self.pos[1]*10+self.T)

    def neg(self):
        return VAR(self.pos, not self.T)

class CLS: # Or of Variables
    # use set of variable is enough
    # to insert into KB need to be imutable(frozenset)
    pass


# Treat as an ADT(abstact data structure) that support desired op.s
# store kb and kb0
# kb: And of Variables
# kb0: inferenced ground truth
# fcls is inmmutable(frozen set)
class KB:
```

```
    kb = set()  # set of cls(frozenset of variable)
    kb0 = set() # set of variable
    max_kb = None
    cls_inserted = None

    def __init__(self):
        self.kb = set()
        self.kb0 = set()
        self.max_kb = 0
        self.cls_inserted = 0

    # return kb0's size
    def atom(self):
        return len(self.kb0)
    # return positive kb0's size
    def pos_atom(self):
        ret = 0
        for var in self.kb0:
            ret += var.T
        return ret

    # return a singleton cls, or None
    def get_single(self):
        for fcls in self.kb:
            if len(fcls) == 1:
                return next(iter(fcls))
        return None

    # add to kb0
    def add_kb0(self, var):
        assert(var.neg() not in self.kb0)
        self.kb0.add(var) # kb0 is a set
        return

    # remove cls from kb
    def remove(self, fcls):
        assert(fcls in self.kb)
        self.kb.remove(fcls)
        return

    # remove cls and add to kb0 + resolution
    def transfer_to_kb0(self, var):
        self.remove(frozenset([var]))
        self.add_kb0(var)
        # resolution
        new_kb = []
        for fcls in self.kb:
            if var in fcls:
                #tautology
                continue
            elif var.neg() in fcls:
                cls = set(fcls)
                cls.remove(var.neg())
                fcls = frozenset(cls)
                assert(len(fcls) != 0)
                new_kb.append(fcls)
            else:
                new_kb.append(fcls)
        self.kb = set(new_kb)
        return


    # insert cls to kb
    # resolution with kb0 and check is not supperset(or eq) to other
    # assert not negative tautology
    def insert(self, cls, CHECKSUB = CHECK_SUB_ON_INSERT):
```

```python
    # resolutions with kb0
    if(isinstance(cls, frozenset)):
        cls = set(cls)
    for truth in self.kb0:
        if truth in cls:
            # tautology
            return
        elif truth.neg() in self.kb0:
            # this part is never True
            cls.remove(truth) # error handling should be redundunt

    # should not be negative tautology
    assert(len(cls) != 0)

    # check not supper set or equal to other this may be too much cost
    flag = True
    if(CHECKSUB):
        for fcls2 in self.kb:
            if(fcls2.issubset(cls)):
                flag = False
                break
    if flag:
        fcls = frozenset(cls)
        self.kb.add(fcls)
        self.cls_inserted += 1
        self.max_kb = max(self.max_kb, len(self.kb))
    return

@staticmethod
# passing cls2, cls are immutable
# return set
def get_match(cls2, cls):
    if(cls.issubset(cls2)):
        return cls
    elif(cls2.issubset(cls)):
        return cls2
    #check complements
    comp = []
    for var in cls2:
        if(var.neg() in cls):
            comp.append(var)

    if(len(comp) > 1):
        return None
    elif (len(comp) == 1):
        var = comp[0]
        resolution_cls = set(cls.union(cls2))
        resolution_cls.remove(var)
        resolution_cls.remove(var.neg())
        return resolution_cls
    else :
        return None

# do pair wise match for cls with size 2
def match(self):
    self.deal_sub()
    kb2 = [fcls for fcls in self.kb if len(fcls) <= 2]

    match_kb = []
    for fcls2 in kb2:
        for fcls in self.kb:
            m = KB.get_match(fcls2, fcls)
            if(m != None):
                match_kb.append(m)
    for cls in match_kb:
```

```
            self.insert(cls)
        return

    # check pairwise subsumption and remove the less restricting ones
    # no dup in set
    # O(n^2)
    def deal_sub(self):
        new_kb = []
        n = len(self.kb)
        for fcls1 in self.kb:
            has_sub = False
            for fcls2 in self.kb:
                if(fcls1 != fcls2 and  fcls2.issubset(fcls1)):
                    has_sub = True
            if(not has_sub):
                new_kb.append(fcls1)
        self.kb = set(new_kb)
        return


class LogicAgent:
    b = None # game board in agent's hand
    B = None # solution, not used, will declare in solve as answer and return
    kb = None # current kowledge, include kb0(marked)

    def __init__(self):
        pass

    def solve(self, b, MATCH_ITER_LIMIT = MATCH_ITER_LIMIT, PRINT_FAIL = True, PRINT=
        # initializing solver
        self.b = b # game engine
        self.kb = KB() # KB
        B = [[-1]*b.m for i in range(b.n)]# answer, -1 not decided, 0 no mine, 1 mine
        kb = self.kb # sugar
        #b = self.b

        # get init positions
        start_pos = self.b.get_start()
        # add inital safe position to KB
        for p in start_pos:
            kb.insert({VAR(p, 0)})

        # start solving
        iteration = 0
        while(1):
            # is done
            if(kb.atom() == b.n*b.m):
                return B, (kb.atom(), b.n*b.m)

            if PRINT:
                iteration += 1
                print("ith iteration, i = ", iteration)
                printA(B)
                print(b)

            # check whether add global constraint now

            def comb(n, r):
                return math.factorial(n) // math.factorial(r) // math.factorial(n-r)

            if(comb(b.n*b.m - kb.atom(), b.nmine - kb.pos_atom()) < GLOBAL_LIMIT):
                undecided = []
                V0 = []
                V1 = []
                for i in range(b.n):
                    for j in range(b.m):
```

```
                    for j in range(b.m):
                        if(B[i][j] == -1):
                            undecided.append((i,j))
                            V0.append(VAR((i,j), 0))
                            V1.append(VAR((i,j), 1))
                assert(b.n*b.m - kb.atom() == len(V0))
                m = len(V0)
                n = b.nmine - kb.pos_atom()
                for cmb in itertools.combinations(V1, m-n+1):
                    kb.insert(set(cmb))
                for cmb in itertools.combinations(V0, n+1):
                    kb.insert(set(cmb))

            # get singleton
            cnt = 0
            while(kb.get_single() == None and cnt < MATCH_ITER_LIMIT):
                kb.match()
                cnt += 1
            if(kb.get_single() == None): # no any sigleton after matching limit
                if PRINT_FAIL:
                    print("Matching Limit Exceed!")
                return B, (kb.atom(), b.n*b.m)

            ## deal with sigleton
            a = kb.get_single() # a singleton cls's only var
            hint = b.q(a.pos)
            assert((hint == -1 and a.T == 1) or (hint != -1 and a.T == 0))
            ## common
            # 1. move to kb0
            # 2. matching of new kb0 to remainning cls
            i = a.pos[0]
            j = a.pos[1]
            kb.transfer_to_kb0(a)
            B[i][j] = a.T
            if(a.T == 0): # is safe and with hint res
                i = a.pos[0]
                j = a.pos[1]
                # undecided = []
                V0 = []
                V1 = []
                n = hint # should - positive B
                for dx, dy in itertools.product(range(-1,2,1), range(-1,2,1)):
                    nx, ny = i+dx, j+dy
                    if(not (dx==0 and dy==0) and b.inrange(nx, ny)):
                        if(B[nx][ny] == -1):
                            # undecided.append((nx, ny))
                            V0.append(VAR((nx,ny), 0))
                            V1.append(VAR((nx,ny), 1))
                        if(B[nx][ny] == 1):
                            n -= 1
                m = len(V0)
                # clause type, C m choose n, now at i, used array
                # use iter tool for higher performance
                for cmb in itertools.combinations(V1, m-n+1):
                    kb.insert(set(cmb))
                for cmb in itertools.combinations(V0, n+1):
                    kb.insert(set(cmb))


def exp(n, m, nmine, T=100, PRINT=100):
    import time
    grid_succ = 0
    grid_norm = 0
    all_succ = 0
    all_norm = 0
    max_kb = 0.0
```

```
    inserted_cls = 0.0
    elasped_time = 0.0

    for i in range(T):
        # init
        b = Board(n, m, nmine)
        agent = LogicAgent()

        #timing and solving
        start_time = time.time()
        B, ri = agent.solve(b)
        end_time = time.time()

        #statistics
        elasped_time += end_time - start_time
        inserted_cls += agent.kb.cls_inserted
        max_kb += agent.kb.max_kb

        grid_succ += ri[0]
        grid_norm += ri[1]
        all_succ  += ri[0]==ri[1]
        all_norm  += 1
        if(i%PRINT == 0):
            print("inserted cls", agent.kb.cls_inserted)
            print("maximum size of kb", agent.kb.max_kb)
            print("time spent", end_time - start_time)
            print("solution")
            printA(B)
            print("task")
            print(b)
    return (grid_succ, grid_norm), (all_succ,all_norm), (elasped_time/T, inserted_cls


GLOBAL_LIMIT = 5000
exp(32, 16, 99, 10, PRINT=10)
```

exp_density.py

- paste only plotting part for simplification

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
# when executing I am using copy of ipython notebook
# I paste only plotting part for simplification
from lab3_2.py import *

d_exp = []
for i in range(8):
    expi = exp(9, 9, 5*i, 33, False)
    d_exp.append(expi)
    print(i, "done")

r1 = []
r2 = []
for i in range(8):
    r1.append(d_exp[i][0][0]/d_exp[i][0][1])
    r2.append(d_exp[i][1][0]/d_exp[i][1][1])

plt.plot(np.asarray(list(range(0,8)))*5/81, r1)
plt.title('grid winrate - density')
plt.xlabel('density')
plt.ylabel('(grid) winrate')
plt.show()

plt.plot(np.asarray(list(range(0,8)))*5/81, r2)
plt.title('game winrate - density')
plt.xlabel('density')
plt.ylabel('(game) winrate')
plt.show()
```

exp_hint.py

- paste only plotting part for simplification

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# when executing I am using copy of ipython notebook
# I paste only plotting part for simplification
# there is actually change that NSTART is used for number of  starting hints in lab3_
from lab3_2.py import *

d_exp = []
for i in range(1, 9):
    NSTART = i*3
    expi = exp(9, 9, 15, 33, False)
    d_exp.append(expi)
    print(i, "done")

r1 = []
r2 = []
for i in range(8):
    r1.append(d_exp[i][0][0]/d_exp[i][0][1])
    r2.append(d_exp[i][1][0]/d_exp[i][1][1])

plt.plot(np.asarray(list(range(1,9)))*3, r1)
plt.title('grid winrate - number of hint')
plt.xlabel('hint')
plt.ylabel('(grid) winrate')
plt.show()

plt.plot(np.asarray(list(range(1,9)))*3, r2)
plt.title('game winrate - number of hint')
plt.xlabel('hint')
plt.ylabel('(game) winrate')
plt.show()

# redo for bigger size

d2_exp = []
for i in range(1, 10):
    NSTART = i*5
    expi = exp(16, 16, 50, 33, False)
    d2_exp.append(expi)
    print(i, "done")

r1 = []
r2 = []
for i in range(9):
    r1.append(d2_exp[i][0][0]/d2_exp[i][0][1])
    r2.append(d2_exp[i][1][0]/d2_exp[i][1][1])

plt.plot(np.asarray(list(range(1,10)))*5, r1)
plt.title('grid winrate - number of hint')
plt.xlabel('hint')
plt.ylabel('(grid) winrate')
plt.show()

plt.plot(np.asarray(list(range(1,10)))*5, r2)
plt.title('game winrate - number of hint')
plt.xlabel('hint')
plt.ylabel('(game) winrate')
plt.show()
```