Project1 Report

Name: Kaichen Zhang ID:40000160

1. Describe and justify how you approach the problem
The logical dots is an n*n board game, with row and column constraints, and given blocks which can't be filled with anything in the n*n board. The constraint is how many dots can exist in the line.

For the game design, I use 0 and 1 integer representation to simulate the dots and blocks, which is 1 representing dots and 0 representing blocks.

So I can check whether the constraints are matched, by calculating the sum of each row and each column, and comparing the sums with the corresponding constraint.

Because we can win the game by two ways: one is locking all grids which will not have dots, the other is filling all grids which have logical dots. So I will find the solution by filling dots and checking whether there are blocks.

Also the game have the requirement of "no other dots are permitted to show around existing dots shapes". After some research and experiments, I find this rule is equivalent to "no dots will appear in the diagonal position of each existing dot". So I added this rule to the design.

Also this board game can be represented as constraint domain programing problem and search problem. So for the first time, I used CLPFD library (http://www.swi-prolog.org/pldoc/man?section=clpb) for SWI-Prolog, this library is popular in solving the constraint domain problems. And I managed to solve the 7*7 even 9*9 puzzle within a shot. Although many useful predicates in SWI-Prolog which are needed to be loaded from CLPFD library, are build-in predicates in GNU-prolog, to avoid the ambiguity I give up this path and represent the game as a simple search problem.

The searching algorithm I used is depth-first search, since the searching space is quite large, and the branching factor is acceptable. Initially I want to search for the state spaces, but generating the successor of current board is hard to define. So I tried to search for the girds step by step, from the left top start to the right bottom end.

2. Describe your heuristic

My heuristics are as follow:

1).checking the diagonal positions of each grid with value 1, if the position has the value of 1, then change it to be 0, so there will be no other dots around existing shapes

2).if the current grid is checked having value of 0 then we realize this is the block.

3).for each row and column check whether all grids only have value 0 or 1.

4).for each row and column check whether the sum of the line matched with corresponding constraint.

5).the evaluation function of A* search is represented as f(n)=c(n)+h(n), where c(n) is the cost from start to current position, h(n) is the heuristic cost from current position to the goal. Every time we check the constraint first, if there can't exist the dot we skip the grid and consider it as a block. I represent the cost of each step as 1, each step meaning putting one dot in the grid. Because the total number of dots is the sum of row constraint or column constrain, each putted dots will lead to a closer position to the goal. So we calculate the current position heuristic by add row heuristic and column heuristic. Row heuristic is the difference of the constraint of the row and number of dots of the line. Column heuristic is similar. For example the row constraint is 4 and column constraint is 5, there are 3 dots in the row and 1 dot in the column, h(n)=(4-3)+(5-1)=5, and we have put 10 dots from start, f(n)=10+5=15. We arrange the node by smallest f(n) in the open list.

3. Describe your program

The files:

There are three files to test for 5*5, 6*6 and 7*7 puzzles.

The data structure:

I represent the game board as a list of lists. And separate the constraints of rows and column as two separate lists. Each grid is represented as X/Y/V, where X is the X coordinate , Y is the Y coordinate, V is the value or saying dot or block of the grid. Dots are represented as 1s, and blocks are represented as 0s.

ProblemR=[4,3,3,3,1,6,1],
ProblemC=[5,2,3,5,1,1,4],
Puzzle = [
[0/0/_,0/1/_,0/2/_,0/3/_,0/4/0,0/5/0,0/6/_],
[1/0/_,1/1/0,1/2/_,1/3/_,1/4/_,1/5/0,1/6/_],
[2/0/_,2/1/0,2/2/_,2/3/_,2/4/0,2/5/_,2/6/_],
[3/0/_,3/1/_,3/2/_,3/3/_,3/4/_,3/5/_,3/6/_],
[4/0/_,4/1/_,4/2/_,4/3/_,4/4/_,4/5/_,4/6/_],
[5/0/_,5/1/_,5/2/_,5/3/_,5/4/0,5/5/_,5/6/_],
[6/0/_,6/1/_,6/2/_,6/3/_,6/4/_,6/5/_,6/6/_]],
Puzzle = [A,B,C,D,E,F,G]

Running:

First we input the row constraint, the column constraint and the puzzle, invoke the main function dfs_solver(ConstraintR,ConstraintC,Input,Path), which will then call the dfs(ConstraintR,ConstraintC,StartPosition,Counter,Visited,Node,Visited) function to perform the depth first search recursively. In the search algorithm, we define move and goal as predicates, and check whether the current node is visited to avoid loop, and backtracking when the goal can't be met.

In the goal predicate, I flat the board as one list, and check whether each gird only have value 0 or1. And then construct a new list by appending the row constraint with the board list representing row by row, check each row whether the sum of line is the same as the constraint of the line. Then check whether each diagonal position of each gird is blocked, to guarantee there are no dots around existing shapes. Then convert the row representation to column representation and check the same thing again.

```
goal(ConstraintR,ConstraintC,Rows) :-
   append(Rows,ConDomain1),all_member(ConDomain1),
   append([ConstraintR],Rows,Horiz),
   maplist(sumGoal,[Horiz]),diag_posiiton(ConDomain1,ConDomain1),
   convtR_C(Rows,Columns),
   append(Columns,ConDomain2),all_member(ConDomain2),
   append([ConstraintC],Columns,Verti),
   maplist(sumGoal,[Verti]),diag_posiiton(ConDomain2,ConDomain2).
```

The all_member/1 predicate will recursively check whether all girds have value of only 0 or 1.

```
all_member([]).
all_member([Head|Tail]) :- Head=_/_/Value,member(Value,[1,0]),all_member(Tail).
```

The sumGoal/1 predicate will recursively sum each row, the base case is the list of an empty list.

```
sumGoal([[]]).
sumGoal([Constraint|Puzzle]) :-
      headTail(Constraint,A,B),
     headTail(Puzzle,C,D),
     C=[_/_/X1,_/_/X2,_/_/X3,_/_/X4,_/_/X5,_/_/X6],
     L=[X1,X2,X3,X4,X5,X6],
     sum_list(L,A),
     append([B],D,R),
     sumGoal(R).
```

I defined blockCheker predicate for move, so whenever there is a block the move will skip the grid.

```
blockCheker(Position,[]).
blockCheker(Position,[Head|Tail]) :-
      Position=X/Y/V,
     Head=XS/YS/VS,
     (X=XS,Y=YS,VS=0
     ->V is 0
     ;blockCheker(Position,Tail)
     ).
```

4. Describe and explains the challenges and the results obtained

The challenges:

For the prolog programing I was confused about everything, like how to pass the parameters, how to modify elements in the list if we can only fetch the head of the list, how to represent a blank grid in the input.

And in the first version of my code, I used CLPFD library, and I got the results for 7*7 puzzle and 9*9 puzzles within 1 second. But the professor told me I can't use such library. Although I find that those predicates I used loaded from CLPFD for SWI-PROLOG are build-in predicates in GNU-Prolog, I use all the most basic predicates for both platform to avoid ambiguity.

The logic and facts of my current implementation is correct, since I checked the 5*5 puzzles, and 6*6 puzzles. And the processing time is within seconds. But the efficiency of the program is not sufficient enough to solve 7*7 puzzle. Even I used depth first search to the program, it's not enough, maybe A* implementation will work.

But I'm also confused how to implement the A* search, since the professor said we should not define move predicates like AI classes, the only permission is listing facts, and goal states.

I think the biggest problem is that the system flow will always go to check the basic conditions of searching algorithm, so when I changed the searching algorithm to BFS,, IDS , the results and time consuming are the same.

Also debug for prolog costs lots of time, if the predicates or the facts have some flaws.