Artificial Intelligence

Assignment 1 Report

Jagat Sastry Pudipeddi-108721027
Ali Haider Zaveri-108661497

Pruning Techniques Used to Discard Unsolvable States

Board Symmetry

The 7x7 board configuration of the Peg Solitaire game is symmetric. The four rotations, and four rotations of a board's reflection are equivalent to the board itself. The algorithm takes advantage of this symmetry to prune the search space. We maintain a list of unsolvable states in which for a particular board state we insert all its rotational and reflection symmetries.

Pagoda Function

To prune the search space further we use a Pagoda function[1]. This is a powerful function that takes advantage of the fact that a peg in a specific position can never move to some given positions. A Pagoda function is a matrix with values assigned to each peg such that when pegs are moved around, the sum of values never increases. To this end, for a given Pagoda function, three consecutive holes a, b, c are assigned values such that a+b>=c, which means that when a is moved to c, the values of a+b is replaced with c. At any stage, if the pagoda value of a child state is lesser than the goal state, it means we can never reach the goal state, and we prune that state. The Pagoda functions we have used can be found in `Pagoda.java`

The same set of pruning techniques are used for all the algorithms

Solving Peg Solitaire using Depth first search

Summary of approach

Starting with the input board configuration, the algorithm generates each successor and traverses along that branch. When it reaches a board configuration that cannot be solved, it explores other successors. The algorithm terminates when the goal state is reached or when we have searched the whole state space and no solution is obtained.

Implementation Details

Invocation: `new ai.PegSolitaireSolver(Board).dfs()`

Algorithm

Given a board configuration we move across the board from top left to bottom right. The first peg that can be moved according to the rules of the game will be the next board configuration and we

call the above function recursively. The base condition is hit when we reach the goal state or a dead end is reached.

When we reach the dead end we backtrack and store the bitmap of the board configuration in a list of unsolvable states. We also store the bitmap configurations of the same state after rotation and vertical reflection. We do this take to advantage of the board symmetry and thus prune the search space.

When the goal is reached, we backtrack and add all moves to a stack which we then use to print the solution.

Pseudo code:

```
dfs(){

      if(goalstate)
            return true


      traverse board from top left to bottom right {

        if(peg can be moved){
            make move
            if(move not pruned and dfs() returns false){
                  store move in stack
                  return true
            }
            generate board configurations symmetric(rotation and
reflection) to current board configuration
            add all generated states to unsolvable list

            restore move
        }
      }
      Return false if goal state is not reached yet
}
```

Summary of performance when Depth First Search is used

On a 7x7 board with 32 pegs and an empty hole in the centre we found the following results:

Pruning Technique	**Expanded states**	**Improvement Factor**	**Time(ms)**	**Memory(MB)**
No Pruning used	7667770	1	10108	46.95
Pagoda Function	1024739	7	1847	**10.84**
Board Symmetry	8992	852	331	24.2
Both	**4577**	**1675**	**157**	14.67

Irrespective of what pruning technique we use, the number of **moves in the solution is 26**

Solving Peg Solitaire using A*

Summary of Approach

Provide a direction to the search by using a heuristic. A heuristic gives information to the search algorithm by informing it how close it is to the goal state. In our case, the quality of the heuristic depends on how well it predicts if the goal state can be reached from the given state. The algorithm starts at the board input configuration and generates all its successors. It applies the heuristic to all the generated successors and adds them to a priority queue. It then chooses the best among all the nodes in the priority queue and continues the search in the same manner till the goal state is reached or no further moves can be made.

Implementation details

Invocation: `new ai.PegSolitaireSolver(Board).aStar()`

Heuristics used

Manhattan distance

The sum of manhattan distance of all pegs relative to the centre peg is calculated as the heuristic value.

Intuition: As the goal state is to have a peg at the centre hole of the board, the main goal was to get the pegs at the periphery towards the centre. As the pegs move more towards the centre the value of the heuristic reduces.

Summary of pruning results with Manhattan distance heuristic:

On a 7x7 board with 32 pegs and an empty hole in the centre we found the following results:

Pruning Technique	Expanded states	Improvement Factor	Time(ms)	Memory(MB)
No Pruning used	135749	1	4005	587.28
Pagoda Function	61412	2.21	**1998**	**166.54**
Board Symmetry	120415	1.12	4105	559
Both	**57467**	**2.36**	2284	265

Irrespective of what pruning technique we use, the number of **moves in the solution is 28**

Weighted Cost*

A hole is assigned a weight based on how undesirable its position is on the board.

---404---
---000---
4030304
0001000
4030304
---000---
---404---

Intuition: Our goal is to assign weights based on how much a peg contributes to the board being unsolvable. Since corners have the lowest degree of freedom, they have the maximum chances of making a board unsolvable and hence we assign them the highest weight. The central hole is given a weight of 1 to make sure that there is no peg on it, to pave way for another peg, towards the end. Since a peg in position with weight 3 in the board above has a 50% probability of moving towards the corner and another 50% of moving towards another 3-weighted hole, we have assigned it a high weightage of 3.
Given a choice between two states, one with the total weight lower than the other, we would naturally choose the one with lower weight, since it provides us with the higher probability of moving towards a solution.
The **weighted cost** heuristic is **more informed than the manhattan distance** heuristic as it considers the properties of specific position the pegs are in rather than just their distance from the centre. Thus it is able to differentiate **more** between good and bad board configurations.

Summary of performance when weighted position heuristic is used

On a 7x7 board with 32 pegs and an empty hole in the centre we found the following results:

Pruning Technique	**Visited states**	**Improvement Factor**	**Time(ms)**	**Memory(MB)**
No Pruning used	50	1	16	1.85
Pagoda Function	50	1	18	1.85
Board Symmetry	34	1.47	24	1.85
Both	**34**	**1.47**	**24**	**1.85**

Irrespective of what pruning technique we use, the number of **moves in the solution is 25**

It is clear that the weighted position heuristic gives much better results when compared to the other heuristic

Pseudo code:

```
astar(){
      initialize priority queue with the start state
      while(goal state is not reached and priority queue is not
empty){
            generate all successors
                        for each successor {
                  if successor not visited yet and not pruned {
                  calculate its heuristic
                  add to priority queue
                  add it and all its symmetries to visited states
                    }
            }
            state=first element of priority queue
      }
       Return true if goal state has been reached, false otherwise
}
```

Avoidance of revisit of already visited states

To prevent a node from being considered twice, we store the visited states and their symmetric equivalents in a hash set. When considering a state, we first check the set if it has been visited already. To use memory efficiently, we use the following scheme to represent a board's state.

Representation of a board state

If the board is represented as a 7x7 array, it can occupy a large space (49 bytes each) if each state is stored that way. To prevent this, we compress the board state such that each hole uses just 1 bit. Since a board has 33 holes, we use a long variable (8 bytes) to represent a state's bitmap. This representation is used to remember the visited and unsolvable states, as well as to store the state in the priority queue used in A*. Memory usage statistics are given in the tables above.

Completeness: Since there are finite number of states and we make sure we don't visit the same state twice, it will either reach the solution or conclude that the problem is not solvable. This same reasoning applies for all the algorithms and heuristics used here.

References

1. Winning ways for your Mathematical plays, Volume 4 – Berlekamp et al, 2004
2. Modelling and Solving English Peg Solitaire Christopher Jefferson et al, 2004