Introduction to Artificial Intelligence Homework 2 report

Andres Ponce(0616110) 彭思安

May 1, 2020

1 Background

Playing games has been an interesting research area in Artificial Intelligence for several decades, and has led to some of the greatest accomplishments in Artificial Intelligence, for example with Deep Blue. Several algorithms can be used to model certain types of games by modifying a search algorithm. One such an example is the game of Minesweeper. This game can be modeled by certain definite rules and thus lends itself very well to being playable by an algorithm.

In this assignment, there was a 6x6 board that consisted of three types of cells: hints, variables, and mines. Hint nodes contain a numeric value that describes how many bombs there are in its 8 possible surrounding cells. Given a limited number of bombs, the objective was to find a configuration of the board in which all bombs are used and every hint node has the amount of adjacent mines described in its value. The objective was to employ a backtracking search to carry out this search for a valid configuration.

2 Solution Description

The backtracking approach used in this assignment involved a simple main algorithm:

Algorithm 1: solve_main(root)

```
Result: valid board configuration
if reject(node) then
    return false;
if accept(node) then
    return true;
make current cell mine;
for children in node do
    found = solve_main(child);
    child = get_closest_cell(node);
    if found = true then
        return true;
restore cell to regular variable;
return false;
```

Following is a description of those methods.

2.1 Rejection

The `reject` method returns true if the valid board configuration is invalid and should be rejected, and returns false otherwise. This method takes in a node from the board (i.e. a cell in the board with some additional info) and performs a simple check on adjacenthint nodes, to see if there are more mine nodes than the hint's value allows for. After that, we move on to the accept method.

2.2 Accepting

The `accept` method will check whether the current board configuration is a valid one according to the constraints given. For a configuration to be valid, we need both all the mines to be used up, and the hint nodes should all have the exact amount of adjacent nodes described in their values. When this happens, then we know we have achieved a valid configuration. Otherwise, this method will return false.

2.3 Recursive Step

When the `reject` method returns, we know that placing a mine in the current cell is a valid move. So, we do that. We change the current cell's value to that of a Mine and adjust the adjacent hint values accordingly, subtracting 1 from the amount of mines they can have besides them. Then, we get the n closest nodes (where n is changeable) and recurse on them. That is, we check all the configurations of the board where the current node is a mine. If it is not possible for a node to be a mine and arrive at a valid end configuration, then we set the current cell to a regular VAR value and return false, meaning the solution is not this way.

The downside to this straightforward recursive approach is its time-complexity, which would be exponential on the number of children n each cell is allowed to check. One quick way to improve this algorithm would be to take a dynamic- programming approach and use an array to keep track of which node configurations work. Then once we hit a node which we have previously tested, we can know beforehand whether it leads to a possible solution or not.

3 Testing on programs

In this assignment, a straightforward Manhattan Distance formula is used to calculate the next node in the sequence. The main source of additional time-complexity comes from changing the amount of chidren that a particular cell may have, i.e. checking the n closest variable nodes which are not yet mines. Another way to calculate the next node to iterate on would involve using other heuristics, for example MRV(by using variable nodes whose domains are now restricted), Degree Heuristic(by measuring the amount of degrees constraining a variable), and

For the attempt with no heuristic in place, the result was always a large amount of nodes that needed to be expanded, since any node could be expanded since it is only based on physical proximity to the current node. And since if the node. In some cases, this would result in a noticeable time required for the algorithm to find a valid configuration of the board.

When attempting to modify the code to use other heuristics, there were some problems. The code was structured in such a way that the `get_closest_node()` function will simply return the next node to be expanded and remove that node from the unassigned queue for the time being, and if it does not lead to a valid configuration it is placed again on the unassigned queue. So the different heuristics were tried to be implemented as a function that retrieves a node by simply using another criterion for "best", other than a simple Manhattan Distance.

My main algorithm will pop off the node from the stack that is currently being examined, and will try to assign it as a mine. Thus, at any given time in the stack there will be only nodes that could either be mines or variables, and thus all elements in the unassigned data structure will have a domain of $\{0, 1\}$.

For the degree heuristic, since we should choose the node with the less amount of constraints, then we should choose the variable nodes with the least amount of unassigned nodes, since these have less constraints placed on them. However, given the code structure already in place, and the expected revisiting of nodes given different conditions on the board made implementing a dynammic programming solution impractical as well, since a node might not be useful given one state of the board but might be the best choice in a later one.

For Comparison, below are the results for the sample problems given no heuristic:

Problem	number of expanded nodes
P1	5374
P2	1892
P3	>30,000
P4	1843

4 Conclusion

From the above results, it appears that the problem of quickly and efficiently using a heuristic-driven approach to Minesweeper can bring great benefits to performance. A flexible code structure would reduce the complexity when using multiple heuristics, especially ones that depend on different aspects of the game state, such as domains or constraints. For this specific problem, a remaining question would involve other possible heuristics that could be used for selecting next nodes in our tree and to avoid repeated expansions of same nodes.

This assignment has taught me about the great usefulness that heuristics bring to a more brute-force method. For some of the sample problems, there was a rather long wait, especially with debug messages. The sheer amount of nodes was still able to be searched rather quickly in C++, however, for an nxn board with different constraints there might be an exponentially larger delay.

With respect to other games, an area that personally appears interesting might be simulating other simple games like Minesweeper using some heuristic-driven approach. What other games could be modeled in a similar manner? Deep Blue required special circuitry to calculate many more possible moves in a certain amount of time. While this in impractical for most, the field of AI has come a long way, and these days there are newer approaches to solving these problems, such as Deep Learning. It seems like a good excercise for a student to develop great problem-solving skills.

5 Notes on the code base

My program was split across several files: `main.cpp`, where the board is only input and passed on to a Board class; `Board.cpp`, where the files related to searching and the main algorithms are stored; and `Board.h`, where they are defined. `Util.cpp` defines some basic utility functions like the Manhattan Distance heuristic and check for validity of two integer coordinates x, y.

Since the `Board.cpp` file is quite long, I decided to include the screenshots of the main parts of the code, namely the methods described in a prevoius section, along with `Board.h`. Sorry for the inconvenience!

```
10 struct Node
11 {
12     int value = -1; //can be either mine(-1) or hint, with number of adjacent mines
13     int remaining; //how many mines can we assign to adjacent cells
14     int x, y;
15     int type;// = VAR;
16 };
17 class Board
18 {
19     private:
20         //Node root; //empty root
21         Node board[BOARD_SIZE][BOARD_SIZE]; //figure out how to make it changeable to other sizes
22         bool visited[BOARD_SIZE][BOARD_SIZE] ;
23 
24         int rows, cols, mines;
25         long int expanded;
26         std::deque<Node*> unassigned;
27         //bool check_adjacent_cells(const Node* node);
28         bool solve_main(Node* node);
29         void adjust_adjacent_values(const Node* node, int operation); //1 for adding 1 to adj. hint
    0 to subtract when assigning mine
30         Node* get_closest_node(const Node* node, int heuristic);
31         Node* no_heuristic(const Node* node);
32         Node* mrv(const Node* node); //minimum remaining values
33         Node* dh(const Node* node);  //degree heuristic
34         Node* lcv(const Node* node); //something or other
35     public:
36         Board(int rows, int cols, int mines)
37         {
38             this->rows = rows;
39             this->cols = cols;
40             this->mines = mines; //number of mines on the board
41             memset(visited, false, sizeof(visited));
42         }
43         void input(int x, int y, int value);
44         void print_out();
45         bool reject(const Node* node);
46         bool accept();
47         void solve();
48 };
49 
50 #endif
```

```
128 bool Board::solve_main(Node* node)
129 {
130     if(node == NULL){return false;}
131     /*Main steps in this function:
132      *1. Check adjacent nodes
133      *2. Check if it is a solution.
134      *3. Generate and recurse on the first child.
135      *4. Generate the successive children and recurse on them.*/
136 
137 
138     if(reject(node))
139     {
140         //std::cout<<"\tneed to reject ("<<node->x<<", "<<node->y<<")"<<std::endl;
141         return false;
142     }
143 
144     else if(accept())
145     {
146         return true;
147     }
148 
149     this->expanded++;
150     node->type = MINE;
151     adjust_adjacent_values(node, 0); //subtract 1 from possible hint nodes
152     this->mines--; //number of mines we've used
153 
154     for(int i = 0; i < CHILD_NO; i++){
155         Node* tmp = get_closest_node(node, HEUR_NONE);
156 
157         if(solve_main(tmp) == true){
158             return true;
159         }
160         unassigned.push_back(tmp);
161     }
162 
163     //if not part of a successful solution path, set to regular var
164     node->type = VAR;
165     adjust_adjacent_values(node, 1);
166     this->mines++;
167 
168     return false;
```

```
170 Node* Board::no_heuristic(const Node* node){
171 if(unassigned.empty()){return NULL;}
172     Node* tmp;
173     Node *best = unassigned.front();
174     int min_dist = this->rows * this->cols; //farthest away next node can be
175     int curr_dist, best_index;
176     std::deque<Node*>::iterator it = unassigned.begin(); //iterator to check for distance
177     std::deque<Node*>::iterator remove; //keep track of closest node iterator to remove it later
178 
179     //search for the closest one and remove it from the list
180     while(it != unassigned.end()){
181         curr_dist = man_distance(best, *(it));
182         if( curr_dist < min_dist){
183             min_dist = curr_dist;
184             best = *(it);
185             remove = it;
186         }
187         it++;
188     }
189     //std::cout<<"\t\tabout to erase: ("<<(*remove)->x<<", "<<(*remove)->y<<")"<<std::endl;
190     unassigned.erase(remove);
191     return best;
192 }
```

```
 9 bool Board::reject(const Node* node)
10 {
11     int x, y;
12     int adjacent_nodes = 8, mine_count = 0;
13     for(int i = 0; i < adjacent_nodes; i++)
14     {
15         x = node->x; y = node->y;
16         switch(i % adjacent_nodes){
17             case 0: //left
18                 y -= 1;
19                 break;
20             case 1: //diagonal up left
21                 x -= 1;y -= 1;
22                 break;
23             case 2: //straight up
24                  y -= 1;
25                 break;
26             case 3: //up right
27                 x += 1; y -= 1;
28                 break;
29             case 4: //right
30                 y+=1;
31                 break;
32             case 5: //down right
33                 x -= 1; y += 1;
34                 break;
35             case 6: //down
36                 y += 1;
37                 break;
38             case 7: //down left
39                 x -= 1; y += 1;
40                 break;
41         }
42         //skip this node if out of bounds
43         if(!is_valid(x, y)){continue;}
44 
45         Node* tmp = &board[x][y];
46         //adjacent hint node can't hold more mines
47         if(tmp->type == HINT && tmp->remaining <= 0){
48             return true;
49         }
50     }
51     return false;
52 }
```

```
56 bool Board::accept()
57 {
58     //check if we still have unassigned mines
59     if(this->mines > 0){return false;}
60 
61     //check all hint cells to see if there are no exceeding mines adjacent to hint cells
62 
63     for(int i = 0; i < this->rows; i++){
64         for(int j = 0; j < this->cols; j++){
65             if(board[i][j].type == HINT && board[i][j].remaining != 0){
66                 return false;
67             }
68         }
69     }
70 
71     return true;
72 }
```