Afraz Salim

r0439731

Time and Moves to solve puzzles

Puzzel	Moves (Manhattan)	Moves(Hamming)	Manhattan(time)
puzzle28.txt . . .	28 Moves	28 Moves	0.8seconds
puzzle30.txt . . .	30 Moves	30 Moves	0.98seconds
puzzle32.txt . . .	32 Moves	GC overhead limit exceeded	8.409seconds
puzzle34.txt . . .	34 Moves	GC overhead limit exceeded	3.071seconds
puzzle36.txt . . .	36 Moves		29.158 seconds
puzzle38.txt . . .	38 Moves		22.47 seconds
puzzle40.txt . . .	40 Moves		4.43 seconds
puzzle42.txt . . .	42 Moves		51.66 seconds

Conclusion:

As I mentioned above that some puzzles get GC overhead limit exceeded, the reason that It happens is, that i create too many objects(new Arrays).I Checked my heurisctic function It never over-estimates the cost.The only solution I see is that I need to keep all successors in hashtable but to compute a hashfuntion() everytime, takes quadratic time which will drastically slow down the program.So I don,t see where is the problem I also tested all the methods used by hamming distance.

It seems that Hamming distance is not a good choice for the data represented in grid and the second thing that I notice that if hamming distance visits the pre-visited state than it makes more movements but solves puzzle easily.

Number of Array accesses while computing manhattan Distance.

Manhattan distance uses ~(3N^2/2) array accesses to compute the distance.

```
public int manhattan()
{
    this.setManhattenDistance(0);
    for(int i = 0; i < this.getSize();i++){
        for(int j = 0; j< this.getSize();j++){
            if(this.board[i][j] > 0){                                   1 array acces
                int xCo = ((this.board[i][j]-1)/this.getSize());        2nd array acces
                int yCo = ((this.board[i][j]-1)%this.getSize());        3rd array acces
                this.setManhattenDistance(this.getManhattenDistance()+ Math.abs(xCo-i)+Math.abs(yCo-j));
            }
        }
    }
    return this.getManhattenDistance();
}
```

Order of growth of manhattan distance to compute the distance is quadratic.A mathematical model can be use to see that the result is valid.

function

First loop runs N times ==> ~N

second loop runs N*N times ==> ~N^2/2(N is the size as input not N*N pieces)

Every time we run a loop the number of array accesses are 3. so by just simply multiplying with the cost of loops gives us the number of array accesses. In this case number of array accesses used by manhattan distance are. ~3N^2/2.

Number of array accesses with Hamming.

The running time of the hamming distance is quadratic as we have to check all the elements to see which elements is misplaced.

```
public int hamming()
{
    int index = 1;
    for(int i = 0; i< this.getSize();i++){
        for(int j = 0; j < this.getSize();j++){
            if(this.board[i][j]  != index)                  one array access
                this.setHammingDistance(this.getHammingDistance()+1);
            index++;
        }
    }
    this.setHammingDistance(this.getHammingDistance()-1);
    return this.getHammingDistance();
}
```

The mathematical function is same as manhattan(number of times that the loops will be executed).

function

First loop runs N times ==> ~N

second loop runs N*N times ==> ~N^2/2

since the loops executes ~N^2/2 and the number of array accesses are one. if we multiply number of array accesses with loops(number of times that the loops are being executed) we can get the result. So total array accesses are ~(N^2/2).

3:

Analysis of isSolveable Method

To analyse the running time of isSolveAble method we need to see two different cases because the first step in isSolveable method, is to see whether the the empty tile is at the end or not. If the empty tile is at the end then we don,t need to move that tile at the end.We just convert it in one dimensional array and count the inversion.This step is bit different from the given in assignment.Insteed of dividing to get the -1 or 1 just take the inversions as power of -1 if the inverions are even it will result in positive otherwise It will give -1. More information can be found here .

case 1:

```
public int getInversions(int[][] board) {
        int inversions = 0;
    if(board[this.getSize()-1][this.getSize()-1] == 0){
        int [] makeOneArray = this.getArrayElements(board);
        inversions = this.countInversions(makeOneArray);
     }
```

first step

second step

third step

First step is needed so that if empty tile is at the end we don,t need to run loops to check.First step take also one array access.

Second step just converts the two dimensional array into one dimensional array.

```
    protected int[] getArrayElements(int[][] board) {
       int [] array = new int[this.getSize()*this.getSize()];
       for(int i = 0; i< this.getSize();i++){
           for(int j =0 ; j < this.getSize();j++){
               array[i*this.getSize()+j] = board[i][j];
           }
       }
        return array;
    }
}
```

(1) Initializing the array uses N array accesses.

(2) The last step at the end of the loops does N (N means N*N for two dimensional array) data moves and 2N array accesses.In total we get 3N array accesses for the this help method.(N array accesses to initialize and 2N array accesses to move data).

Third step counts the the number of inversion by checking how many elements are smaller than the current element and they come after that element.

```
377
378
379
380
381 private int countInversions(int[] makeOneArray) {
382    int inversions = 0;
383      for(int i  = 0; i< this.getSize()*this.getSize()-1;i++){
384          for(int j = i+1; j< this.getSize()*this.getSize()-1;j++){
385              if(makeOneArray[j] < makeOneArray[i] )
386                  inversions++;
387          }
388       }
389     return inversions;
390     }
391
392
```

since we are using one dimensional array and the size of the board is N*N but we write N for one dimensional array which represents N*N.

We see that the first element is compared with the rest of (N-1) elements. Totals number of comparisons are ~N^2/2 .

Total array accesses are 2 everytime we run the loops .Again we multiply the cost of loops with the cost of array acesses to get total.

Loops execute ~N^2/2

Number of array accesses are 2.

Total number of array accesses are 2.(N^2/2) which is equal to ~(N^2). Note(N is not same as it was for two dimensional array)

Case 2:

In the first case we supposed that the empty tile is always at the end of the initially given array.If the empty tile is not at the end of the given array then we need to move it to the end. One easiest way to do that is to move it to the most right corner and then to the down because the position of empty tile is bottom right corner.

```
359
360  public int getInversions(int[][] board) {
361          int inversions = 0;
362      if(board[this.getSize()-1][this.getSize()-1] == 0){
363          int [] makeOneArray = this.getArrayElements(board);
364          inversions = this.countInversions(makeOneArray);
365        }
366      else
367      {
368          int [][] array = new int[this.getSize()][this.getSize()];
369          int [][] arr = this.getEmptyTileAtEnd(this.copyfromOriginal(array));
370          int [] makeOneArray = this.getArrayElements(arr);
371          inversions = this.countInversions(makeOneArray);
372
373      }
374          return inversions;
375      }
376
377
```

2nd Step

(1) Again we have to transfer all the elements from the original array to the new array so it does not effect the original array.which makes N data moves(as the size of the board is N*N).

(2) Second step is to move the empty tile to the bottom right corner.strategy to do that has been already discussed in previous section. Number of moves to move empty tile at the end depends on the initially given board but loops will be executed ~N^2/2 times.(One thing to see about two dimensional arrays is that we say N for the input size, for example for a board representing 9 item , N is equal 3 and N^2 is equal to 9).

(3) Third step is also already discussed in previous section.

Space complexity of A*

Time complexity of A* depends on the on the heuristic functions.If the heuristic always returns exact cost every time, to reach the goal in that case the time complexity will be linear since at each step we have to travel down the tree.

Worst-Case Space:

A star has exponential space complexity in the worst-case.The minimum number of nodes which can be expanded anytime is 2 and maximum are 4(right, left, bottom and top) so the space complexity in worst case will be ~(b)^N but average number for branching factor is 2 so we can say that the space complexity in worst case will be ~(2)^N (where N is the depth of the tree). We take 2 as base because minimum nodes which can be expanded at any point are two(for example if the empty tile it in top left corner or bottom right). This is lowest possible upperbound.

proof:

(1) initially we insert the first puzzle in priority queue.

Since we say that the branching factor is 2. In the first step the initially puzzle will be removed and two succesors will be inserted in priority queue.(Note that the previous board is deleted) from the queue.Second step the two boards will be deleted and 4 boards will be added to priority queue.(Everytime we delete a board then we also add successors to queue and the minimum numbers of successors are two) at this step the length of the tree is 2 and the branching factor is 2 which gives us 2*2 = 4.

Choice between memory, time and priority function.

If we use more memory than I don,t think that the puzzle will be solved in less time and to solve the puzzle in shorter time we need a better priority function.A better priority function will be good choice because if we say that we use extra memory to solve a puzzle then this statement would not hold.Extra memory usage ==> more nodes expanded ==> more time to solve and check each node. So a priority function can balance both memory usage and time taken to solve the puzzle.One thing I just find important that the heuristic should be consitant.

Better priority functions.

Pattern-DataBase:

Pattern-database stores the exact cost to the solution from every subproblem and some tiles are kept unique. For example if the last row in a puzzle is solved(3*3 puzzle) then we can keep those tiles as unique.which will save both time and memory as explained here .

Better algorithm than A*

Bidirectional Search:

I think Bidirectional search will be a better choice because it searches from initial to goal and from goal to initial and both trees meet each other In middle and the memory usage will also be less than A*.If we start search from initial to goal then as the tree grows it starts expanding more nodes but if we also start searching from goal and meet the first tree then the nodes will be less.(simply node from begin to middle are less than the node from begin to end).Bidirectional search will also improve the running time.For example recently checked node was middle node in the tree and then A* finds a node with lower cost which is 1 steps before the last checked node.Instead of A* if we use bidirectional search we could simply check the both trees are handeling same node then we could easily terminate search and reconstruct the path.More information can be found here .