CS21120 - Assignment 1: Sudoku - Analysis of Algorithms

Alexander Brown

October 14, 2010

1 Notes

1.1 Algorithm Terms

Time Complexity refers to the number of nodes generated during a seach. Measured using the "Big O" notation.

Space Complexity refers to the maximum number of nodes stored in memory during a search. Again measured in the "Big O" notation.

Optimality refers to the guarentee that an optimal solution can be found.

Completeness refers to the guarentee that a solution can be found (if one exists).

Time and Space Complexity both assume the worst case, even if it was illegal within the rules of sudoku.

1.2 Something

$b =$ the brancing factor of a Tree Structure.

$d =$ the depth of the answer in a Tree Structure.

$m =$ the maximum depth of a Tree Structure.

$l =$ the depth limit of a Tree Structure.

2 Bredth First Search

2.1 Overview

All nodes at the current level are expanded before any nodes at the next level are expanded.

2.2 Time Complexity

$O(b^d)$
In the case of a regular 9×9 sudoku, where $b = 9$ and $d = 81$, there are 9^{81} attempts.

2.3 Space Complexity

$O(b^d)$
In the case of a regular 9×9 sudoku, where $b = 9$ and $m = 81$, there are 9^{81} nodes open at the deepest level.

2.4 Optimality

The Bredth First Search is Optimal.

2.5 Completeness

The Bredth First Search is Compete.

2.6 Implementation

The formal method would be to create a Tree data structure with a brancing factor of 9.
However, it is simpler to use a Queue. Due to the issue of Space Complexity, an Unbound Queue must be used.

2.7 Psuedocode

```
Sudoku root = the current sudoku
bool found = false
Queue queue = new UnboundQueue()

function void run()
{
  queue.insert(root)

  while(!found && !(stack.isEmpty()))
  {
    step()
  }

  if(found)
  {
    print("Solution Found")
  }
  else
  {
    print("Sudoku Impossible")
  }
}

function void step()
{
  Sudoku cursor = queue.remove()

  if(cursor.isComplete())
  {
    found = true
  }
  else if(cursor.isLegal())
  {
    int position = cursor.getNextFreeNode()

    for(int i=1;i<10;i++)
    {
      Sudoku child = new Sudoku(cursor, position, i);
      queue.insert(child)
    }
  }
}
```

3 Depth First Search

3.1 Overview

The Depth First Search expands the node at the deepest level. When it reachest a dead end it expands the shallowest node that has unexplored sucessors.

3.2 Time Complexity

$O(b^m)$

In the case of a regular 9×9 sudoku, where $b = 9$ and $m = 81$, there are 9^{81} attempts.

3.3 Space Complexity

$O(b \times m + 1)$

In the case of a regular 9×9 sudoku, there will only be $9 \times 81 + 1 = 730$ nodes open at a time.

However it is worth noting that each node has to store a whole sudoku, so the Depth First Search can take up a lot of space

Assuming a sudoku grid is a simple array with the length of 81, where each tile is a single byte character:

$(81 \times 8) \times (9 \times 81) = 472392bits \approx 58kB$.

3.4 Optimality

The Backtracking is not optimal.

3.5 Completeness

The Depth First Search is complete, if there is enough memory to store the tree in.

3.6 Implementation

The formal method would be to create a tree data structure with a branching factor of 9.

However, it is simpler to use a Stack. As the Space Complexity is not large, it is more efficient to use a Bound Queue.

3.7 Psuedocode

```
Sudoku root = the current sudoku
bool found = false
Stack stack = new BoundStack(MAX_SPACE) //MAX_SPACE = 730 in a typical sudoku

function void run()
{
  stack.push(root)

  while(!found && !(queue.isEmpty()))
  {
    step()
  }

  if(found)
  {
    print("Solution Found")
  }
  else
  {
    print("Sudoku Impossible")
  }
}

function void step()
{
  Sudoku cursor = stack.pop()

  if(cursor.isComplete())
  {
    found = true
  }
  else if(cursor.isLegal())
  {
    int position = cursor.getNextEmptyTile()

    for(int i=1;i<10;i++)
    {
      Sudoku child = new Sudoku(cursor, position, i)
      stack.push(child)
    }
  }
}
```

4 Depth Limited Search

4.1 Overview

An improvement of the Depth First Search, which imposes a cut off on the maximum depth.

It should be noted that for this particular problem, individually the Depth Limited Search is not useful. However it is used in the Iterative Deepening Algorithm, so it has to be included.

4.2 Time Complexity

$O(b^l)$

In the case of a regular 9×9 sudoku, where $b = 9$ and $l = 81$, there are 9^{81} attempts.

4.3 Space Complexity

$O(b \times l)$

In the case of a regular 9×9 sudoku, where $b = 9$ and $m = 81$, there are $9^{81} = 729$ nodes open.

4.4 Optimality

The Depth Limited Search is not Optimal.

4.5 Completeness

Is complete is $l \geq d$.

4.6 Implementation

As Depth First Search, only with a limit on the depth.

4.7 Psuedocode

```
structure DepthLimitedNode
{
  int depth;
  Sudoku sudoku;
}

DepthLimitedNode root = new DepthLimitedNode(0, current sudoku)
bool found = false;
depthLimit = 81
Stack stack = new Stack(MAX_DEPTH) //MAX_DEPTH = 9*depthLimit

function boolean run()
{
  stack.push(root)

  while(!found && !(stack.isEmpty()))
  {
    step()
  }

  return found
}

function void step()
{
  DepthLimitedNode cursor = stack.push()
  Sudoku sudoku = cursor.sudoku

  if(sudoku.isComplete)
  {
    found = true
  }
  else if(sudoku.isLegal)
  {
    if(cursor.depth < depthLimit)
    {
      int position = sudoku.getNextEmptyTile()

      for(int i=1;i<10;i++)
      {
        Sudoku childSudoku = new Sudoku(sudoku, position, i)
        DepthLimitedNode child = new DepthLimitedNode(childSudoku, cusor.depth+1)
      }
    }
  }
}
```

5 Iterative Deepening Algorithm

5.1 Overview

Tries all possible depth limits in turn, combining the advantages of Breadth and Depth First Searches.

5.2 Time Complexity

$O(b^d)$

5.3 Space Complexity

$O(b \times d)$

5.4 Optimality

The Iterative Deepening Algorithm is Optimal.

5.5 Completeness

The Iterative Deepening Algorithm is Complete.

5.6 Implementation

Consecutively calls the Depth Limited Search.

5.7 Psuedocode

```
bool found = false
int depth = 0

function void run()
{
  while(!found && depth < 81)
  {
    step()
    depth++
  }
}

function void step()
{
  DepthLitedSearch dls = new DepthLimitedSearch(depth)
  found = dls.run()
}
```

5.8 Optimisation

Using the above code will actually make the Time Complexity greater than $O(b^d)$. There should therefore be some way of finding out if a node has been previously expanded.

6 Elimination Algorithm

6.1 Overview

Moving away from brute forcing the problem, the Elimination Algorithm starts to act like a human might; eleiminating values from a particular position by looking it's row, colomn and sub-grid. If, after elimination, there is a single value left, then that must be the value which this position takes.

6.2 Time Complexity

$O\left(\frac{d^2+d}{2}\right)$

6.3 Space Complexity

$O(1)$

6.4 Optimality

The Elimination Algorithm is not optimal.

6.5 Completeness

The Elimination Algorithm is not complete.

6.6 Implimentation

There are not Abstract Data Types which can represent this algorithm.

6.7 Psuedocode

```
Sudoku sudoku = current sudoku
bool found = false
bool changed = false
int position = 0

function boolean run()
{

  do
  {
    changed = false

    while(!found && (position < 81))
    {
      step()
    }
    if(found)
    {
      return true
    }
    else
    {
      position = 0
    }
  }
  while(changed)

  return found
}

function void step()
{
  String possiblities = "123456789"
  String row = sudoku.getRow(position)
  String colomn = sudoku.getColomn(position)
  String subGrid = sudoku.getSubGrid(position)

  possibilites = remove(row, possibilities)
  possibilites = remove(colomn, possibilities)
  possibilites = remove(subGrid, possibilities)

  if(possibilites.length = 1)
  {
    int value = possibilies[0]
    sudoku.setValue(position,value)
    changed = true
  }
}

function String remove(String string, String from)
{
  for(int i=0;i<string.length;i++)
  {
    from.replace(string[i],""])
  }

  return from
}
```

7 Presistant Elimination Algorithm

7.1 Overview

An expansion on the Elimination Algorithm, but it stores the possibilities each tile can take. It can then use this to derive certain facts about the row, colomn and/or sub grid that it would not normally be able to.

For example, say we have a 3×3 sub grid like so (where superscript numbers represent values which the tile could take):

^{1}	2	^{57}
3	4	6
^{157}	8	9

5 and 7 effectively are placed in the two slots the could be in, as there is no where else for them to go. As 1 does, it would get placed in the top-left tile.

Another example:

9	8	7	...	...	5	...	...	...
			...	...	...	...	^{5}	^{5}
	1	2	...	...	...	...	...	...

5 cannot take any value in the middle row, so therefore must take the bottom-left tile in the left-most sub grid.

7.2 Time Complexity

$O\left(\frac{d^2+d}{2}\right)$

7.3 Space Complexity

Not entirely sure.

7.4 Optimality

The Persistant Elimination Algorithm is not optimal.

7.5 Completeness

The Persistant Elimination Algorithm is not complete, but more complete than the original Elimination Algorithm.

7.6 Implimentation

Based on the Elimination Algorithm, however it requires some changes to the Sudoku board, or needs a virtual one to function correctly.

7.7 Psuedocode

```
//todo
```