Fundamentals of Data Structures and Algorithms for AI

Sudoku Implementation

Prepared By

Sintayew Zekarias

Fikir Awoke

Course Instructor - Dr. Beakal Gizachew

Addis Ababa Institute of Technology

School of Information Technology and Engineering

February 6, 2022

1. Introduction

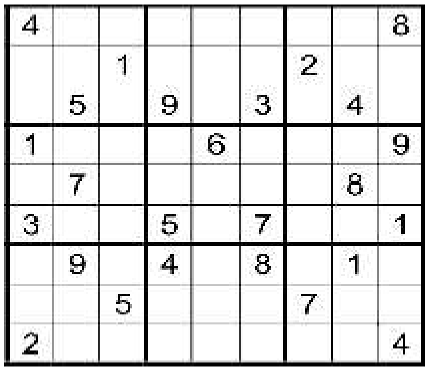
Searching algorithms are used to solve problems from simple to more complex problems. Time and space is an important element to decide which algorithm is designated to solve a specific problem. There are various searching algorithms for solving problems. In this assignment we tried to show five searching algorithms to solve the Sudoku problem. The Breadth-First Search, Depth-First Search, Backtracking search, Heuristic search and A* search.

2. The Sudoku Puzzle

Sudoku is a game that is played by a single individual trying to fill the blank space with the appropriate numbers based on the rule of the game. The puzzle has a board that contains nine by nine squares where some of the squares are filled with random numbers and some others are empty. The objective of the game is to fill all the remaining empty spaces. To fill up the spaces there are certain rules that must be followed by the player in order to win the game. The rules are as follows:

- Ø If the puzzle is played with numbers, only integer values from one to nine are acceptable.
- Ø Every square in the same row and column must be filled in with a different number.
- Ø Each one of the nine boxes must have different numbers.

The main idea to solve the game is to try each combination of numbers in each empty spot making sure that the rules are not violated.



4								8
		1				2		
	5		9		3		4	
1				6				9
	7						8	
3			5		7			1
	9		4		8		1	
		5				7		
2								4

Figure1: The Sudoku puzzle

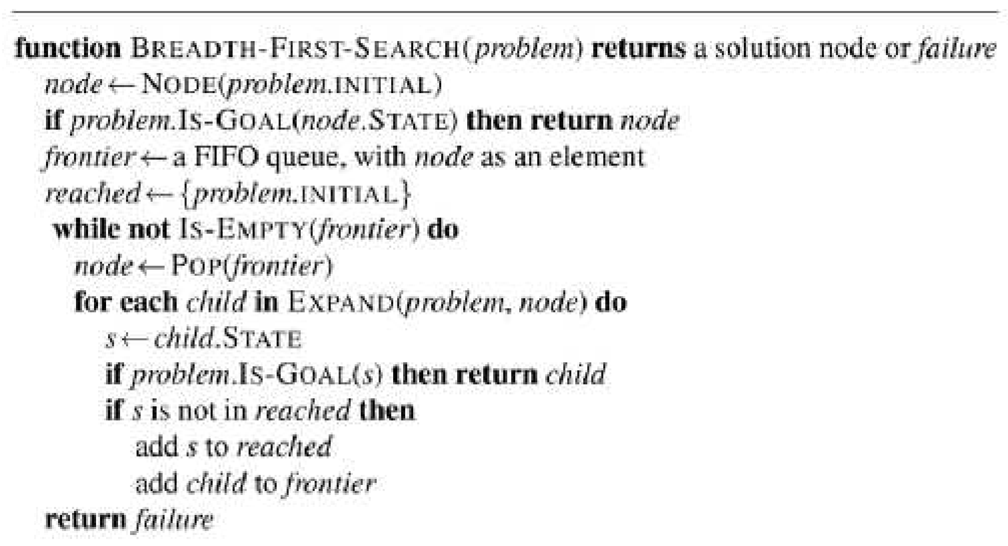
3. The Searching Algorithms

Breadth-First Search and Depth-First Search are uninformed search strategies that use the concept of a queue to expand the states from a graph or a tree structure.

3.1. Breadth-First Search

In breadth-first search the root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on until it gets to the bottom level. It uses a FIFO (First-In-First-Out) queue. This is a systematic strategy that is therefore complete even in an infinite state space.

The BFS algorithm is shown below:

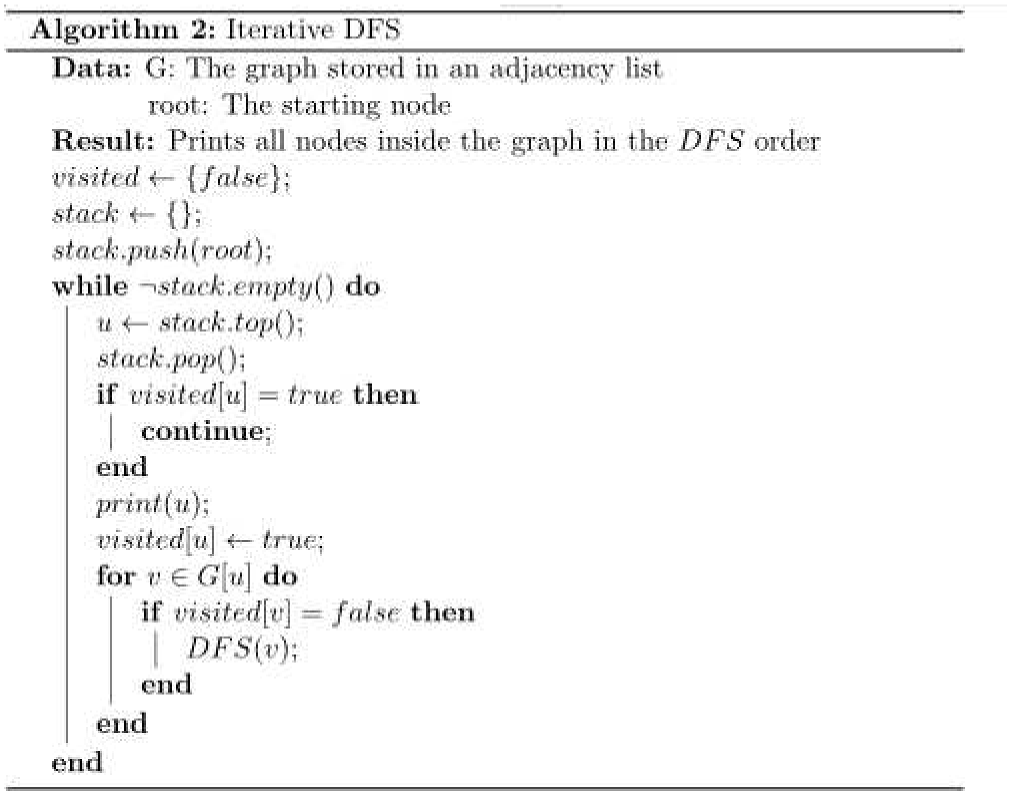


```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
  node ← NODE(problem.INITIAL)
  if problem.IS-GOAL(node.STATE) then return node
  frontier ← a FIFO queue, with node as an element
  reached ← {problem.INITIAL}
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if problem.IS-GOAL(s) then return child
      if s is not in reached then
        add s to reached
        add child to frontier
  return failure
```

3.2. Depth-First Search

The depth-first search always expands to the deepest node in the frontier first. It will go deeper until it finds a goal or a dead end. The search then “backs up” to the next deepest node that still has unexpanded successors. The algorithm uses the LIFO (Last-In-First-Out) queue. Depth-first search is not cost-optimal; it returns the first solution it finds, even if it is not the cheapest solution.

The DFS algorithm is shown below:



```
Algorithm 2: Iterative DFS
  Data: G: The graph stored in an adjacency list
        root: The starting node
  Result: Prints all nodes inside the graph in the $DFS$ order
  $visited \leftarrow \{false\}$;
  $stack \leftarrow \{\}$;
  $stack.push(root)$;
  while $\neg stack.empty()$ do
      $u \leftarrow stack.top()$;
      $stack.pop()$;
      if $visited[u] = true$ then
          continue;
      end
      $print(u)$;
      $visited[u] \leftarrow true$;
      for $v \in G[u]$ do
          if $visited[v] = false$ then
              $DFS(v)$;
          end
      end
  end
```

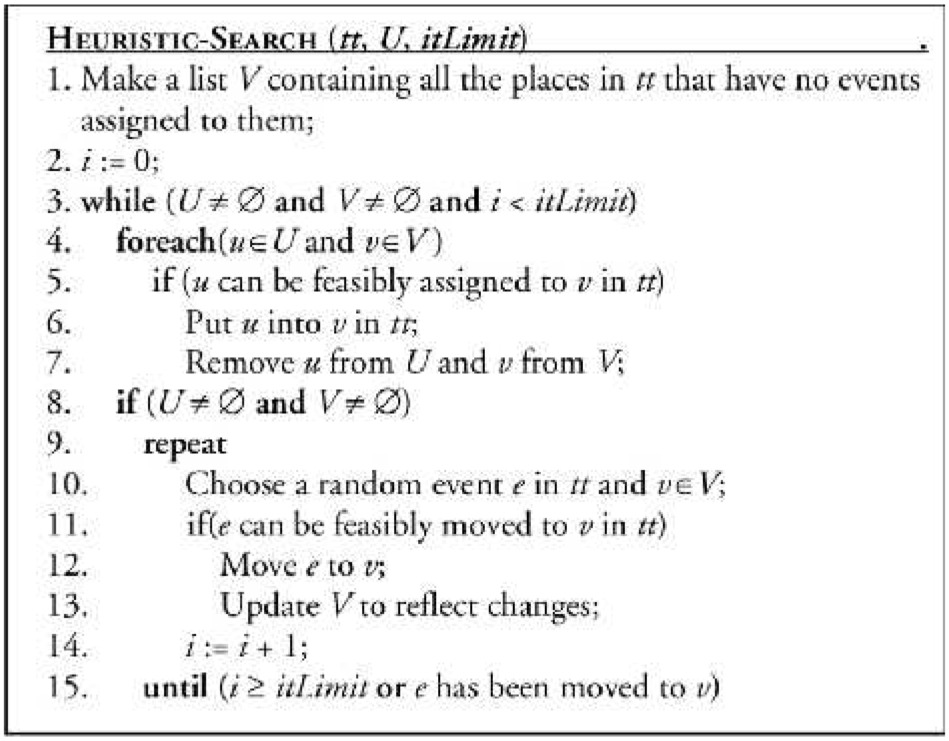
3.3. Backtracking search

Backtracking is an algorithmic-technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point of time.

3.4. Heuristic search

Heuristic search is defined as a procedure of search that works to upgrade an issue by iteratively improving the arrangement dependent on a given heuristic capacity or a cost measure. This technique makes decisions by ranking every available choice at each branch of a search and then chooses the best option of those presented. Rather than focusing on finding an optimal solution like other search methods, heuristic searching is designed to be quick, and therefore finds the most acceptable option within a reasonable time limit or within the allocated memory space. It is considered an Informed Search, and it finds the most promising path.

The Heuristic search algorithm is shown below:



```
HEURISTIC-SEARCH (tt, U, itLimit)
1. Make a list V containing all the places in tt that have no events
   assigned to them;
2. $i := 0$;
3. while ($U \neq \emptyset$ and $V \neq \emptyset$ and $i < itLimit$)
4.     foreach($u \in U$ and $v \in V$)
5.         if (u can be feasibly assigned to v in tt)
6.             Put u into v in tt;
7.             Remove u from U and v from V;
8.     if ($U \neq \emptyset$ and $V \neq \emptyset$)
9.         repeat
10.            Choose a random event e in tt and $v \in V$;
11.            if(e can be feasibly moved to v in tt)
12.                Move e to v;
13.                Update V to reflect changes;
14.            $i := i + 1$;
15.        until ($i \geq itLimit$ or e has been moved to v)
```

3.5. A* search

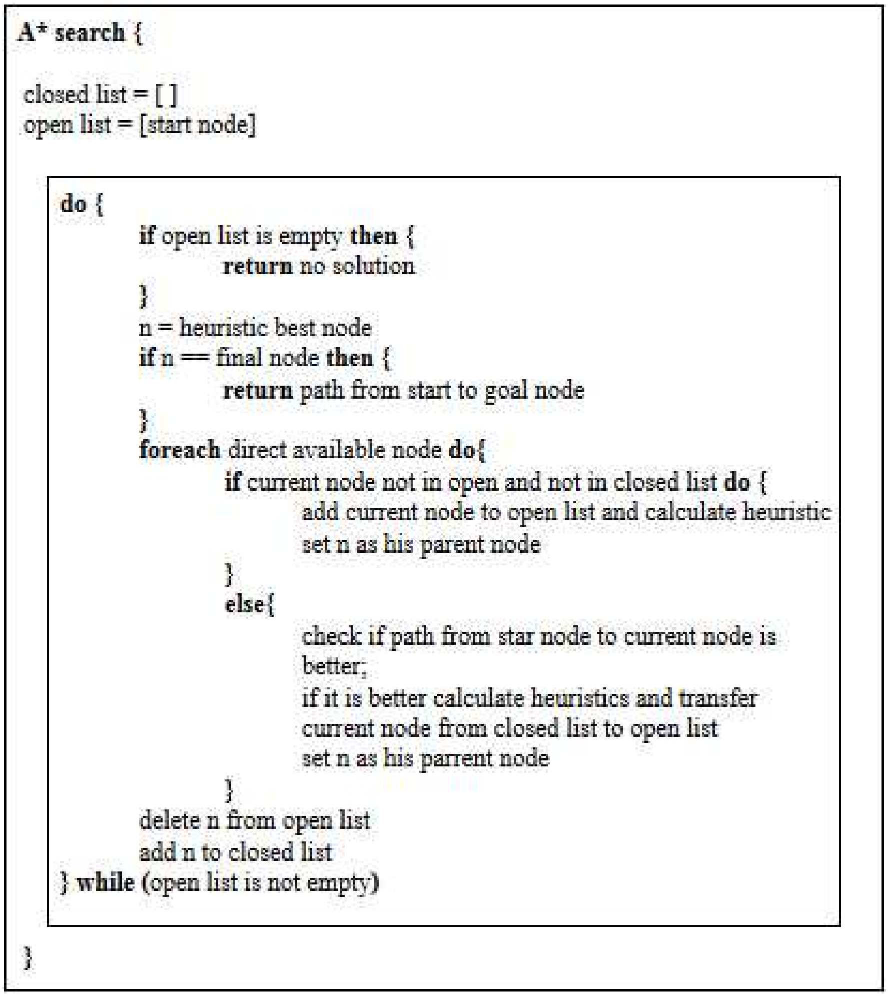
A* search is a graph traversal and path search algorithm, which is often used in many fields of computer science due to its completeness, optimality, and optimal efficiency. One major practical drawback is its O (b d) {\displaystyle O(b^{d})} space complexity, as it stores all generated nodes in memory.

Unlike other algorithms, A* decides to take up a step only if it is convincingly sensible and reasonable as per its functions. Which means, it never considers any non-optimal steps. This is why A* is a popular choice for AI systems that replicate the real world – like video games and machine learning.

A* algorithm works based on heuristic methods and this helps achieve optimality. A* is a different form of the best-first algorithm. Optimality empowers an algorithm to find the best possible solution to a problem. Such algorithms also offer completeness. If there is any solution possible to an existing problem, the algorithm will definitely find it.

When A* enters into a problem, firstly it calculates the cost to travel to the neighboring nodes and chooses the node with the lowest cost. If The f(n) denotes the cost, A* chooses the node with the lowest f(n) value. Here ‘n’ denotes the neighboring nodes.The heuristic value has an important role in the efficiency of the A* algorithm. To find the best solution, you might have to use different heuristic functions according to the type of the problem.

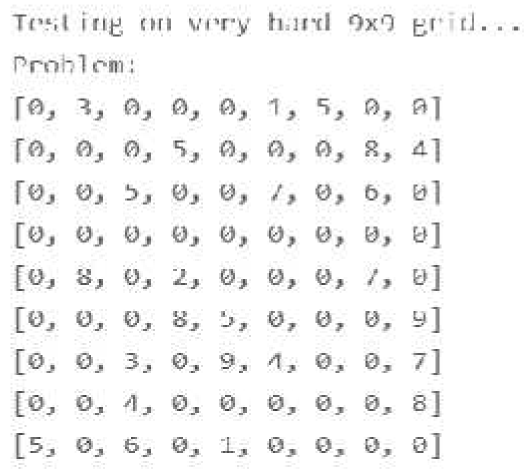
The A* algorithm is shown below:



```
A* search {

 closed list = [ ]
 open list = [start node]

    do {
            if open list is empty then {
                    return no solution
            }
            n = heuristic best node
            if n == final node then {
                    return path from start to goal node
            }
            foreach direct available node do{
                    if current node not in open and not in closed list do {
                            add current node to open list and calculate heuristic
                            set n as his parent node
                    }
                    else{
                            check if path from star node to current node is
                            better;
                            if it is better calculate heuristics and transfer
                            current node from closed list to open list
                            set n as his parrent node
                    }
            delete n from open list
            add n to closed list
    } while (open list is not empty)

}
```

4. Experimental Result

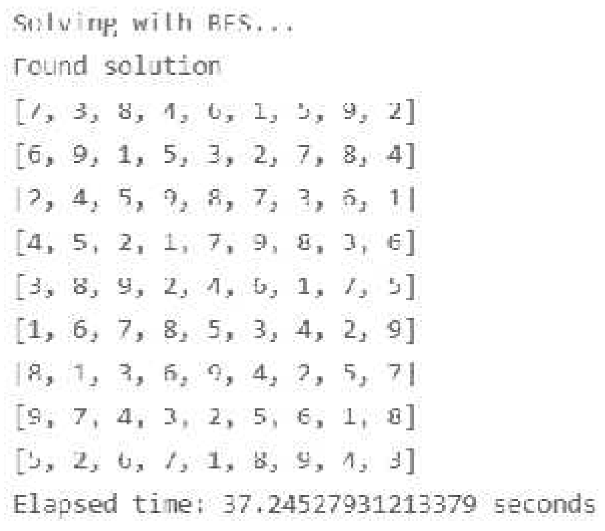
We took as an exampl testing on very hard 9x9 girid problem, the result and time complexity of each algorithm pressented as follow



```
Testing on very hard 9x9 grid...
Problem:
[0, 3, 0, 0, 0, 1, 5, 0, 0]
[0, 0, 0, 5, 0, 0, 0, 8, 4]
[0, 0, 5, 0, 0, 7, 0, 6, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 8, 0, 2, 0, 0, 0, 7, 0]
[0, 0, 0, 8, 5, 0, 0, 0, 9]
[0, 0, 3, 0, 9, 4, 0, 0, 7]
[0, 0, 4, 0, 0, 0, 0, 0, 8]
[5, 0, 6, 0, 1, 0, 0, 0, 0]
```

4.1. BFS Result

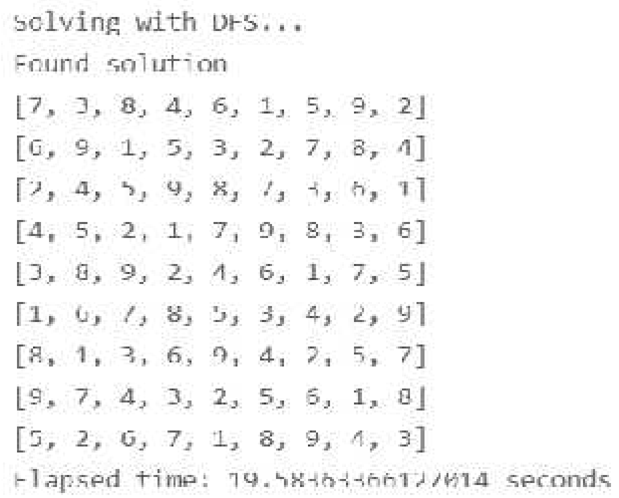
The following result shows the solved puzzle using the BFS algorithm, with the respective elapsed time in seconds.



```
Solving with BFS...
Found solution
[7, 3, 8, 4, 6, 1, 5, 9, 2]
[6, 9, 1, 5, 3, 2, 7, 8, 4]
[2, 4, 5, 9, 8, 7, 3, 6, 1]
[4, 5, 2, 1, 7, 9, 8, 3, 6]
[3, 8, 9, 2, 4, 6, 1, 7, 5]
[1, 6, 7, 8, 5, 3, 4, 2, 9]
[8, 1, 3, 6, 9, 4, 2, 5, 7]
[9, 7, 4, 3, 2, 5, 6, 1, 8]
[5, 2, 6, 7, 1, 8, 9, 4, 3]
Elapsed time: 37.24527931213379 seconds
```

4.2. DFS Result

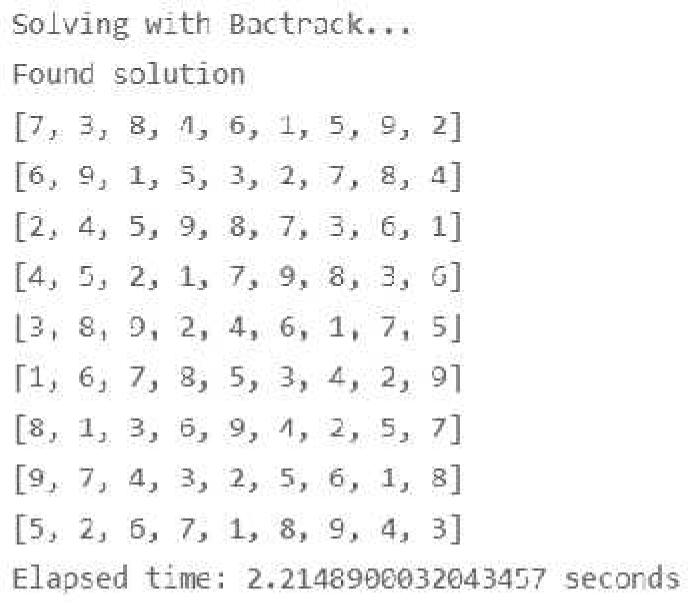
The following result shows the solved puzzle using the DFS algorithm, with the respective elapsed time in seconds.



```
Solving with DFS...
Found solution
[7, 3, 8, 4, 6, 1, 5, 9, 2]
[6, 9, 1, 5, 3, 2, 7, 8, 4]
[2, 4, 5, 9, 8, 7, 3, 6, 1]
[4, 5, 2, 1, 7, 9, 8, 3, 6]
[3, 8, 9, 2, 4, 6, 1, 7, 5]
[1, 6, 7, 8, 5, 3, 4, 2, 9]
[8, 1, 3, 6, 9, 4, 2, 5, 7]
[9, 7, 4, 3, 2, 5, 6, 1, 8]
[5, 2, 6, 7, 1, 8, 9, 4, 3]
Elapsed time: 19.58363366127014 seconds
```

4.3. Backtracking Result

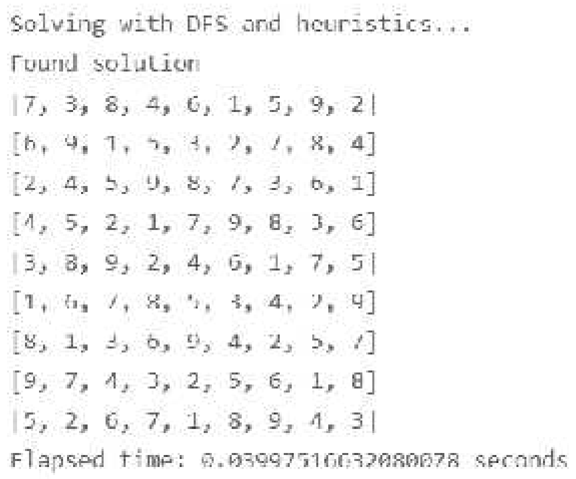
The following result shows the solved puzzle using the Backtracking search algorithm, with the respective elapsed time in seconds.



```
Solving with Bactrack...
Found solution
[7, 3, 8, 4, 6, 1, 5, 9, 2]
[6, 9, 1, 5, 3, 2, 7, 8, 4]
[2, 4, 5, 9, 8, 7, 3, 6, 1]
[4, 5, 2, 1, 7, 9, 8, 3, 6]
[3, 8, 9, 2, 4, 6, 1, 7, 5]
[1, 6, 7, 8, 5, 3, 4, 2, 9]
[8, 1, 3, 6, 9, 4, 2, 5, 7]
[9, 7, 4, 3, 2, 5, 6, 1, 8]
[5, 2, 6, 7, 1, 8, 9, 4, 3]
Elapsed time: 2.2148900032043457 seconds
```

4.4. Heuristic Result

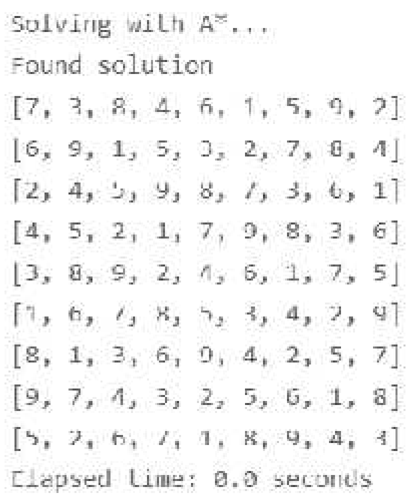
The following result shows the solved puzzle using the Heuristic Search algorithm, with the respective elapsed time in seconds.



```
Solving with DFS and heuristics...
Found solution
[7, 3, 8, 4, 6, 1, 5, 9, 2]
[6, 9, 1, 5, 3, 2, 7, 8, 4]
[2, 4, 5, 9, 8, 7, 3, 6, 1]
[4, 5, 2, 1, 7, 9, 8, 3, 6]
[3, 8, 9, 2, 4, 6, 1, 7, 5]
[1, 6, 7, 8, 5, 3, 4, 2, 9]
[8, 1, 3, 6, 9, 4, 2, 5, 7]
[9, 7, 4, 3, 2, 5, 6, 1, 8]
[5, 2, 6, 7, 1, 8, 9, 4, 3]
Elapsed time: 0.03997510032080078 seconds
```

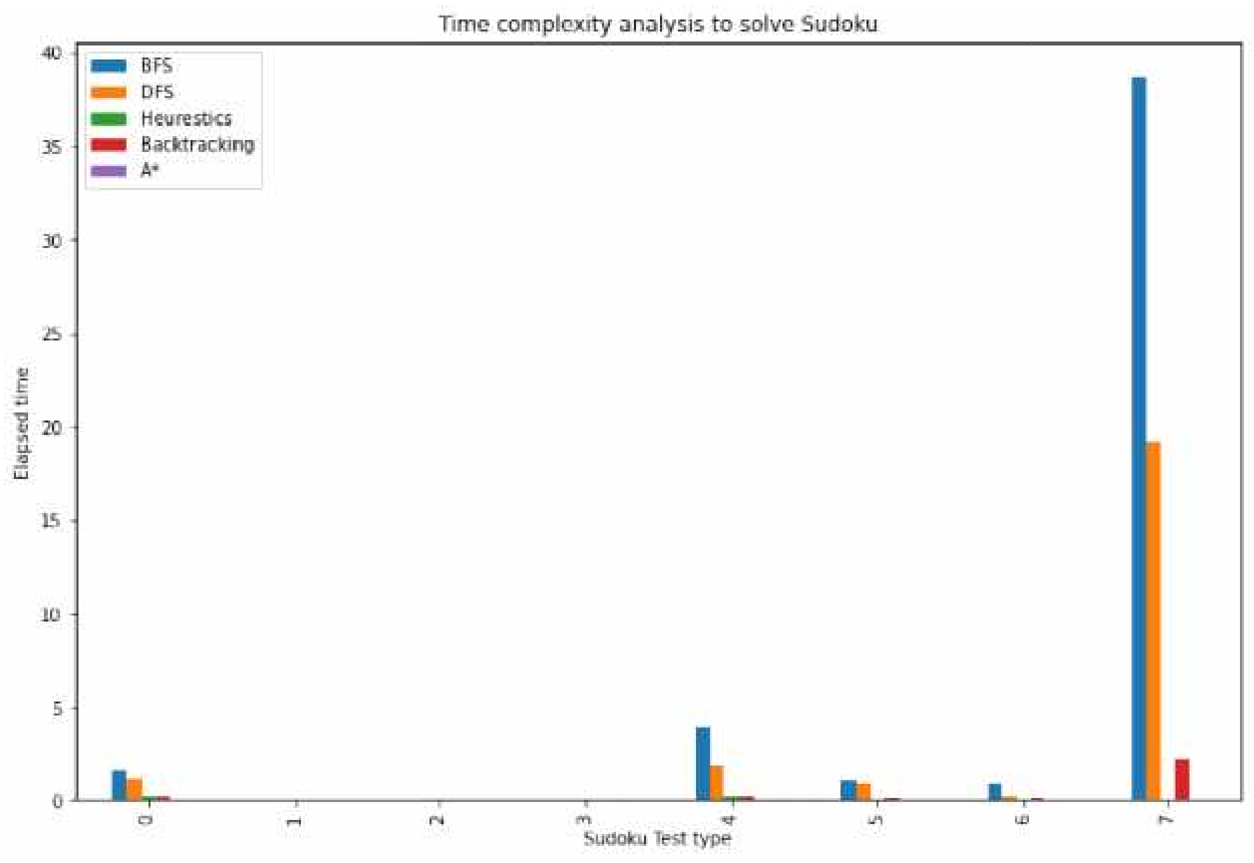
4.5. A* result

The following result shows the solved puzzle using the A* algorithm, with the respective elapsed time in seconds.

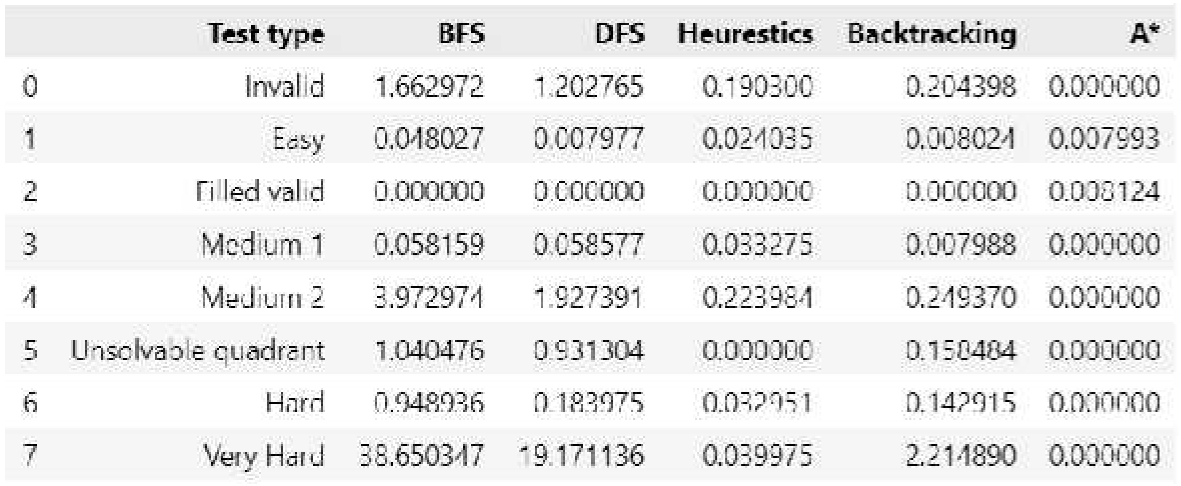


```
Solving with A*...
Found solution
[7, 3, 8, 4, 6, 1, 5, 9, 2]
[6, 9, 1, 5, 3, 2, 7, 8, 4]
[2, 4, 5, 9, 8, 7, 3, 6, 1]
[4, 5, 2, 1, 7, 9, 8, 3, 6]
[3, 8, 9, 2, 4, 6, 1, 7, 5]
[1, 6, 7, 8, 5, 3, 4, 2, 9]
[8, 1, 3, 6, 9, 4, 2, 5, 7]
[9, 7, 4, 3, 2, 5, 6, 1, 8]
[5, 2, 6, 7, 1, 8, 9, 4, 3]
Elapsed time: 0.0 seconds
```

4.7. Plotted sudoku Result



4.8. Comparison of Algorithms



	Test type	BFS	DFS	Heurestics	Backtracking	A*
0	Invalid	1.662972	1.202765	0.190300	0.204398	0.000000
1	Easy	0.048027	0.007977	0.024035	0.008024	0.007993
2	Filled valid	0.000000	0.000000	0.000000	0.000000	0.008124
3	Medium 1	0.058159	0.058577	0.033275	0.007988	0.000000
4	Medium 2	3.972974	1.927391	0.223984	0.249370	0.000000
5	Unsolvable quadrant	1.040476	0.931304	0.000000	0.158484	0.000000
6	Hard	0.948936	0.183975	0.032951	0.142915	0.000000
7	Very Hard	38.650347	19.171136	0.039975	2.214890	0.000000

5. Conclusion

This work included five searching algorithms to solve the sudoku puzzle, which are the Breadth-First Search, Depth-First Search, Backtracking search, Heuristic search and A* search. Different concepts were implemented throughout the process in order to obtain the best performer in solving the puzzle. Based on our testing, the sudoku puzzle on A* search has a better result with the lowest elapsed time compared to all the other algorithms implemented.

Reference

https://en.wikipedia.org/wiki/Sudoku_solving_algorithms

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.646.8115&rep=rep1&type=pdf

https://en.wikipedia.org/wiki/A*_search_algorithm

https://www.mygreatlearning.com/blog/a-search-algorithm-in-artificial-intelligence/