CSCI4511W - Assignment 1: Sudoku - BFS

Name : Wen Chuan Lee
x500 : leex7095
ID : 4927941

Usage

(The command to run your code.)

I have already represented the provided problem sudokus in my script, simply executing my script (or importing `sudokuSolver()` and calling `sudokuSolver.runApp()` in an interpreter) **will solve the sudoku and display runtimes as well.**

However, alternatively, in order to use the sudoku solver, first define a valid sudoku problem with unknowns marked as 0, and then pass it in through the sudoku_driver() function like so:

```
sixBySixFirst = [
        [1, 5, 0, 0, 4, 0],
        [2, 0, 0, 0, 5, 6],
        [4, 0, 0, 0, 0, 3],
        [0, 0, 0, 0, 0, 4],
        [6, 3, 0, 0, 2, 0],
        [0, 2, 0, 0, 3, 1],
    ]

sudoku_driver(sixBySixFirst)
```

There is also an option to not use pruning, as I have designed the solver by first solving the solution naively with BFS, by directly executing the breadth-first search function with a Problem constructor like so:

```
breadth_first_search(Problem(fourByFour, prune=False))
```

The sudoku driver will print the original sudoku being solved, the solution sudoku (if there is one) as well as the solution as a list (from the child SolutionNode to the parent). The time it took to find the solution will also be printed, this was done with use of the time module, the set-up gives a rough estimate of elapsed time (link to reference)

Bugs and Incomplete Parts

(Identify any incomplete parts; Identify any bugs that exist in the code)

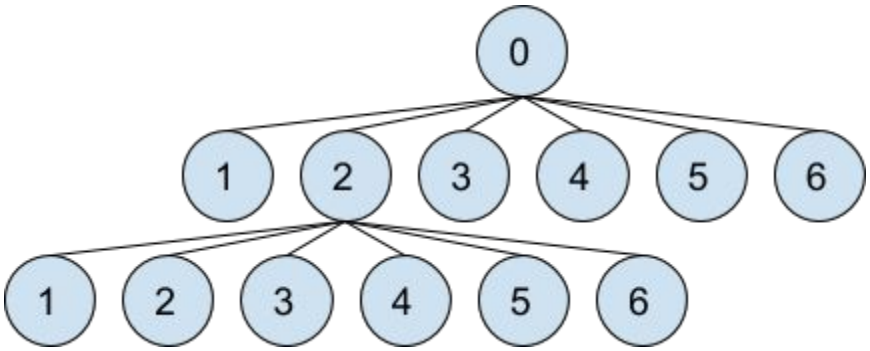
There are no known bugs and incomplete parts of this assignment.

Node Representation

(Describe what a node represents in your implementation.)

For my BFS sudoku solver, each node represents a *possible value/state* that can fill the unknown/blanks in a sudoku. This would mean that the maximum number of possible nodes per level is *n* nodes where *n = dimension of the sudoku*. For example, there are a maximum of 9 nodes per level in a 9x9 sudoku, and maximum of 6 nodes in a 6x6 sudoku. Thus, there are *n* number of nodes at each level in an $n * n$ sudoku. The highest parent node will have a state of 0, it is used as an indication that we have reached the highest point of the tree.

The values per node will therefore range from $1 - n$.



Pruning

(Describe what you pruned (i.e. what constituted an illegal action).)

Pruning is implemented after the solver was able to naively solve the sudoku, this worked fine on boards with few blank values, or on 4x4 boards, but could not even finish executing after 10 minutes to obtain a solution for the 9x9 sudoku given.

Illegal actions in sudoku are when there are more than one occurrence of the same number in a row, column, or subgrid.

The tree was pruned first by checking if a given action (possible state) is already in the row or column of the board. If the number already exists, a *None* type is returned, which will not generate a node as it is considered an invalid state.

Subgrid pruning is later implemented for the 6x6 and 9x9 boards. This was harder to implement given my node implementation was only a single possible value and not an entire subgrid. The 6x6 and 9x9 individual subgrid checking is hardcoded by checking a particular subgrid if the possible state being tested falls within a range of indexes for the particular subgrid, as well as an optional flag that considers the action a legal one if there exist zero values (as the solution is not fully generated, it is still a possible and legal action). This sped up BFS significantly by avoiding traversal down impossible solutions.

Implementation Details

(Describe any modifications that you made to the BFS (or indicate that you created your own).)

Aside from using a proper function call for a queue in Python and adding the option to print a tree out, the `breadth_first_search(Problem)` function remains unchanged.

However the Problem framework has been changed significantly. The main change is that I have passed in the entire parent node to the result function as I require previous parent states to check for a valid solution when pruning, as well as keep track of the depth/ which unknown index I am checking for. A lot of helper functions (for example a function that returns the entire column of a board as a list) that check for correctness of a sudoku solution was added as well. These are used in pruning and in the final check for correctness.

The Problem framework has a single instance of the Sudoku board, as well as other variables that are used by other functions to refer to board length or list of tuples that contain the ‘blanks’/unknowns of the Sudoku board.

The `expand()` and `child_node()` functions in the Node class was also modified to return nodes that are not of type None. This was needed as only valid states should be put into the frontier for exploration and traversal.

The reason I have chosen a single value as my node representation for state is because I initially felt that creating only nodes that hold possible values at each unknown would be more memory efficient as opposed to storing multiple permutations of a grid in each node. As Python lists are not duplicated unless deepCopy() is used, having a list of index values that update only 1 instance of the sudoku board is far more memory and performance efficient. As possible values are updated into the board and then checked.

This however resulted in a very different implementation that requires a lot more logic keeping track of the index in the right way as well as much more code needed to correctly index and check the board. However, the performance was great after implementing pruning. Memory usage was also fairly small on my system (unless a completely blank board was passed in).

Pruning is done in the result() function as the action() function generates a list of possible states unless we have reached the depth (number of unknowns) we need. I chose to prune in the result function as it is easier to pass in an entire parent node from the result function as return a valid state given how the given framework is set up and how my problem representation is set up.

Questions and Answers

1. **Write the equation using b and d that represents the size of the search tree when there is no pruning. Is this different from the search space (briefly justify)?**

Answer:

The size of the search tree when there is no pruning is b^d.

This is **not any different** from the search space, as the search space without pruning will also be b^d at the worse case where the solution is found at the final node.

2. **When there is no pruning, what are the minimum and maximum number of nodes you might need to explore to find a solution? Briefly justify.**

Answer:

The **minimum** number of nodes **without pruning** on BFS is $b^{d-1}+1$ as the best case for a solution to be found is at the first node of the lowest depth of the tree. For example, in a 9x9 sudoku solution with 5 blank/unknown states, the minimum number of nodes to be explored using BFS to find the solution is $9^{5-1} + 1 = 6562\ nodes$.

The **maximum** number of nodes **without pruning** on BFS is b^d as the worst case for the solution is found at the last node of the lowest depth of the tree. For example, in a 9x9 sudoku solution with 5 blank/unknown states, the maximum number of nodes to be explored is $9^5 = 59049$.

This effect is amplified as the number of unknowns increase, the number of nodes to explore increases exponentially.

3. **Quantify the effects of pruning on the size of the search tree. You can do this empirically by counting nodes within your code or derive a theoretical bound.**

Answer:

The effects of pruning significantly reduce the size of the search tree on a given problem as it avoids expansion and traversal down impossible/illegal solution paths. This effect is compounded early on for problems with a lot of blank/unknown states as every depth generates the number of nodes **exponentially**.

Through an empirical analysis of the a simplified 6x6 sudoku puzzle (attempting to run the provided 6x6 problems without pruning took up 5GB of RAM on my machine which had 8GB available and ran for 4 minutes with no sign of completion), the number of nodes added to the frontier to be traversed by BFS without pruning on the *simplified* 6x6 with **7 unknown/blank states** is **841526** before a solution was found (with an elapsed time of 10.512 seconds).

The number of nodes added to the frontier *with pruning* for the same problem is only **9**. *The nodes were calculated by adding a counter that incremented whenever a node was added to the frontier.* By not generating nodes with an illegal state and adding them to the frontier early in the tree we have reduced the amount of computation and memory needed to solve a sudoku problem, of which for larger problems with more unknown/blank states would have a time and space complexity a personal computer could not handle.