A* Sudoku Solver

Michael Wade
Dr. Novobilski
CPSC 580 – Machine Learning

1.A* Sudoku Solver

The A* algorithm, typically a path planning algorithm, can be used to solve grid problems such as Sudoku puzzles. The algorithm works by evaluating each possible move for the current grid state (or location) and choosing the optimal choice. The evaluation of each move is one of the traits that separates A* from other search algorithms such as DFS and Greedy Search. Evaluation of a potential movement is based on the numerical approximation returned from the addition of two cost functions, g (n) and h(n). The function g(n) approximates the cost of moving to grid slot n and the h(n) function (the heuristic) is an estimation of the cost to reach the solution from the grid slot [1]. The optimal slot is chosen based on the scores of each slot and then used as part of the solution.

2.Slot Structure

To store information about the grid, each slot has been defined as its own entity using the struct statement from the C language. Within the grid slot structure [Figure 1] each slot is assigned the following variables: int status, int score, int i, int j, and int possibles[9]. The status variable states if the slot is open or closed. The score variable, defined by the equation g(n) + h(n), will change on each iteration through A* until the status become closed. Finally, the possibles[] array will be initialized to 1,2,3,4,5,6,7,8,9 and is used extensively by both the heuristic function and A*'s guessing mechanism.

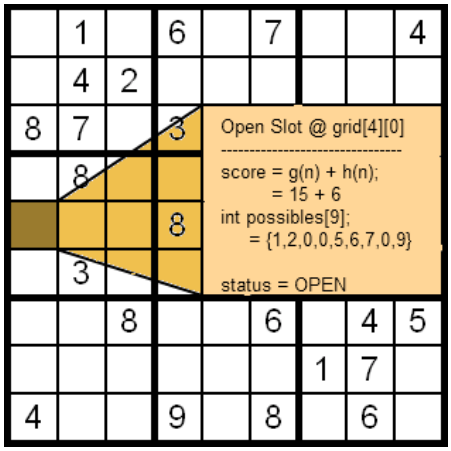

Figure 1 - Slot Structure

3.h(n)

The heuristic defined in this document contributes to solving Sudoku puzzles by extracting information about the current puzzle, using the rules of the game, and by exploiting the inner logistics of the A* algorithm. The commonality between heuristic functions for any puzzle is their purpose: to return an accurate estimate of how optimal a certain grid slot would be as a solution choice. Knowing this, one can use the rules of Sudoku and the context of a given puzzle state to calculate a potential cost.

The heuristic that was explored for this implementation returns the number of potential values for a blank slot based on the current puzzle state. Accomplished by checking each closed slot[1] on the same row, column and 3x3 region, the heuristic returns a value of one to nine. Returning values in the range of one to nine keeps the heuristic function from overestimating the cost. In the event that the calculation yields a zero for the slot, a nine would be returned and the function sets a flag to indicate an invalid choice was previously made. We know a previous choice was invalid since every open slot should have a possible choice unless a mistake exists within the grid. Since the A* algorithm was implemented using recursion, the flag would force the current iteration to return to the previous and continue to return until the error flag is removed. The next possible path is then chosen from the slot's potential values, which were stored as an integer array in the slot's structure.

4.g(n)

The other cost function, g(n), is used in addition (literally) to the heuristic function as an indicator of the slot's potential for yielding a solution. The less the value returned by g(n), the greater the chance that A* will choose the slot as optimal for the current iteration. The simplest way to estimate this is to subtract the number of open slots on the current slot's row, column, and 3x3 region from the total number of possible slots (8+8+4). The hypothesis is that the less connected the slot is to the rest of the grid the greater chance that A* will be able to correctly guess the slot's value. While the heuristic function produces an overall score, the g(n) function acts as an invisible hand that guides A* to different areas of the puzzle based on potential impact.

5.A* Exploitation

This implementation also exploits the A* guessing mechanism by storing the potential values for each slot in an integer array within the slot's structure. Initialized to one through nine, the array values are set to zero as the heuristic function finds the slot's potentials. Each slot's array of potentials is then used as the guessing pool from which A* will draw its decision. After selecting a potential guess, the recursive call is made to A* and the loop continues by scoring open slots, choosing an optimal slot, and then guessing at another potential. The process is complete when the grid is solved or determined to be unsolvable.

---

[1] The A* algorithm maintains two lists, open and closed. In Sudoku, the closed list contains all grid slots with a predefined value or a value that is assumed to be accurate. The open list contains all slots yet to be assigned a value.

6.Analysis

The Depth First Search (DFS) algorithm is a prime example of an algorithm that does not use a scoring method to choose optimal movements. The Depth First Search algorithm searches for possible solutions to the puzzle by randomly guess at each slot and moving to the next in a recursive manner. DFS has the ability to backtrack; however, its backtracking process will only randomly guess at another possibly incorrect solution. With the implementation of A* presented above, each chosen path is optimally selected based on the scores of open slots. The value placed in the slot is limited to valid choices and the backtracking of our A* implementation will only lead us down another path that is possible based on the rules of the game and the current game state.

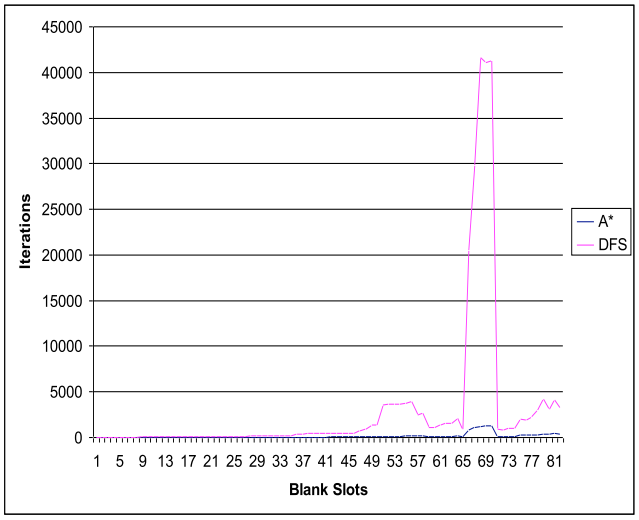

Figure 2 - Algorithm Comparison

The graph presented in Figure 2 is a comparison of the A* algorithm to DFS. Each algorithm was used to solve 82 different puzzles with zero to eighty-one blank slots per puzzle. The resulting graph can be used to support the fact that the heuristic function leads A* to the appropriate slot in most cases and minimizes the need to guess at potentially incorrect values. What is particularly interesting to notice is the similarity between the two graphs on different scales. While the A* algorithm is superior at solving puzzles, both algorithms show difficulties in the same areas.

The first noticeable increase in iteration size occurs when the number of blanks rises above fifty. Examining the graphs separately, Figures 3 and 4, the pattern becomes transparent, there is near parallelism. Both algorithms have trouble solving the solutions for grids with 67 to 71 open slots. However, at 72 open slots the amount of necessary iterations decreases close to the average before rising again.

The similarity between the graphs can be explained by the similarities between the algorithms. Both algorithms will attempt guesses at possible solutions until a flaw is recognized in the puzzle and then backtracking will attempt to recover from the incorrect solution. A* simply improves upon this approach by invoking a scoring system that enables selection of an optimal slot before guessing at the value. The average difference between the twos graphs in this analysis is 2914 iterations; however, in the high iteration section of the graphs, puzzles with 67 to 71 open slots, the average difference is 33725 iterations. Therefore, A* yields an increase in efficiency by 95% overall compared to DFS and 97% in areas of parallel difficulty.

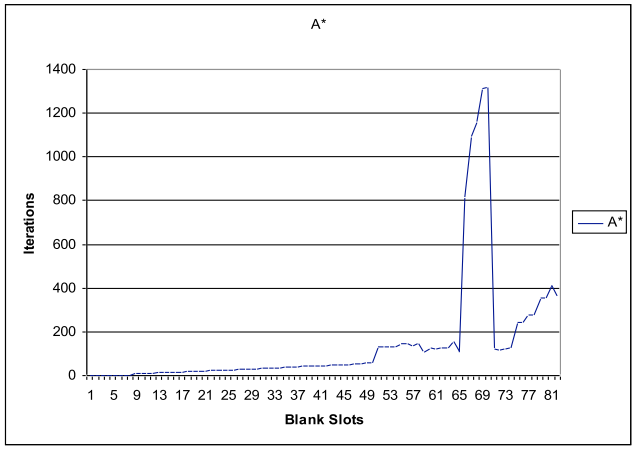

Figure 3 - A* Results

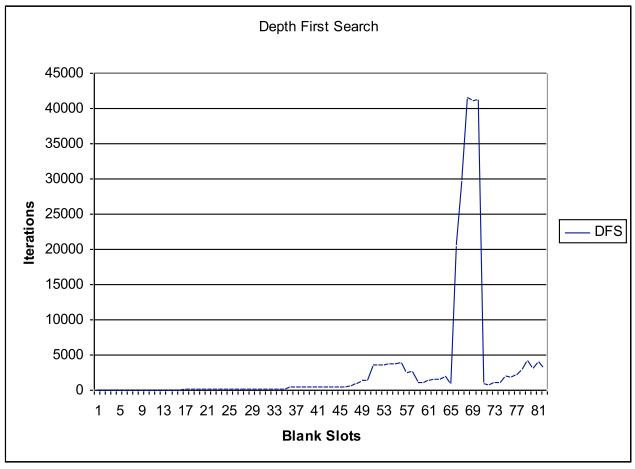

Figure 4 - DFS Results

7.REFERENCES

[1] Bourg, D. M., Seeman, G. *AI for Game Developers*. O'Reilly Media, Inc, Sebastopol, CA, 2004.