Data Structure Analysis

Word count: ~1200

Introduction

This report discusses the simple and advanced data structures' effect on puzzle solving performance.

The simple method of solving the puzzle is to read in the dictionary from a text file and add each entry into a vector. It then iterates through each entry in the vector and compares the first letter of that entry against the grid. If there is a match then it checks the neighboring cells against the second letter of the entry. For each neighboring cell that matches a second letter, a search is carried out in one of the 8 directions (north, south, west, east, etc.). The entry's length is considered and the search to that direction is aborted if the current grid cell coordinate is out of range. A word is found when all of the entry's letters have been validated before the search is aborted. The found word, starting coordinates and direction number of that word are then added as a tuple into the found words vector.

The advanced solver takes advantage of the advanced dictionary while maintaining the simple grid structure. The advantage in doing so is that the number of unique dictionary entries needed to be checked is minimized and redundancies are eliminated by recognizing that words can share the same origin. The advanced dictionary consists of node struct objects in the heap. A node holds a letter, a pointer to its parent node, a boolean indicating if it's final (the end of a word), and a vector holding pointers to other subnodes (nodes which have this node as their parent). A word is constructed by reading each word as a string from the text dictionary file. Then that node's letter is checked against a vector of pointers to all primary nodes (nodes which hold the first letter of an entry). If the primary node already exists, a node pointer called current node is set to point to primary node and the current letter index of the string dictionary entry is incremented so that we look at the next letter. If a current node's subnode matches the letter in the string dictionary entry at the current index, the current node is set to the matching subnode and the above process is repeated until the letter in the string dictionary entry at the current index fails to match any of the current node's subnodes. When this happens, the rest of the word is built by the branchbuilder method. It does so by making a new node for every letter in the string dictionary entry after the last matched letter. Each new node will point to its parent node, will have its heap address saved in its parent node's subnodes vector, and will hold the next node's address in its subnode vector.

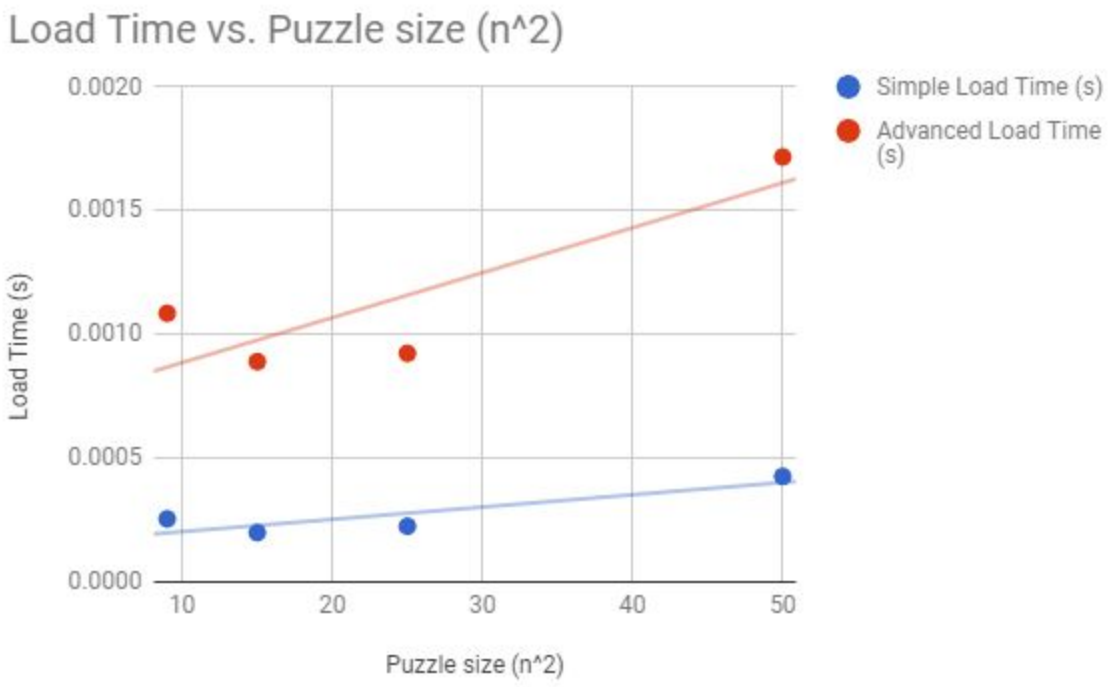
The advanced solving process is similar to the simple one in the sense that they share the same, dictionary to puzzle approach. Each primary node is checked against the grid. If there is a match all of its subnodes are compared against the surrounding grid cells. For every subnode match, a check is carried out in that direction until the end of the grid is reached, or until a subnode with no subnodes is encountered. When that happens, the last node found with an isFinal bool set to true, is taken to be either the last or first node (depending on whether the word is backwards or forwards facing) of a found word sequence.

Results

Puzzle size (n^2)	Simple Load Time (s)	Advanced Load Time (s)	Simple Solve Time (s)	Advanced Solve Time (s)	Simple Grid Cells Visited	Advanced Grid Cells Visited	Simple Dictionary Entries Checked	Advanced Dictionary Nodes Checked
9	0.00025	0.0011	0.00001	0.000004	1,303	890	10	99
15	0.00020	0.0009	0.00002	0.000014	8,108	4,701	20	536
25	0.00022	0.0009	0.00007	0.000043	27,579	17,402	30	1,221
50	0.00043	0.0017	0.00041	0.000137	185,884	65,331	50	5,334

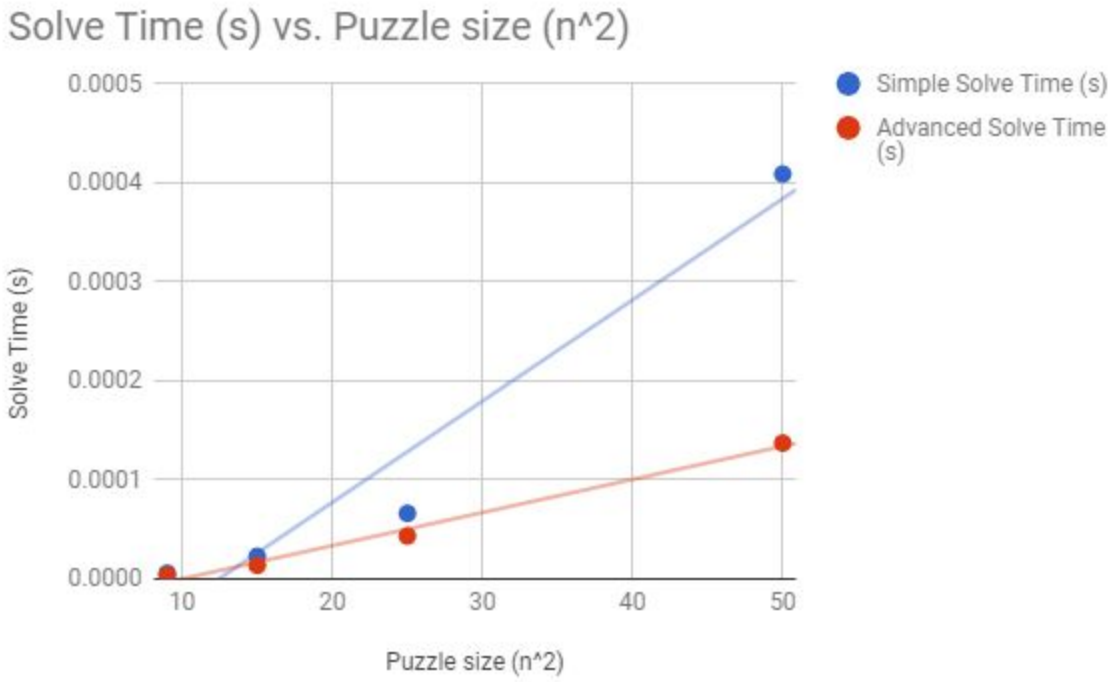
Data Presentation and Discussion

Chart 1.



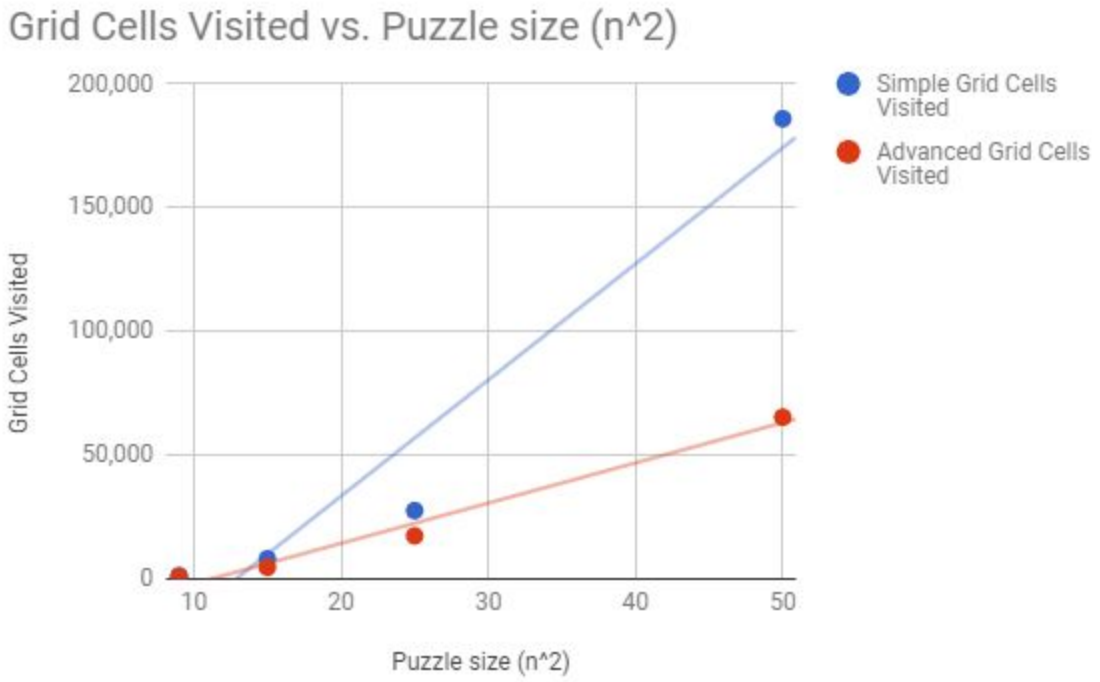
Load time increases linearly with both the simple and the advanced solver. The rate of increase in load time with the advanced data structure is greater than that of the simple one. This could be due to the increased complexity of the algorithm that makes the advanced dictionary. However, the increase in load time is a necessary trade off for eliminating redundancies (i.e. words that share the same origin) and solving the puzzle faster.

Chart 2.



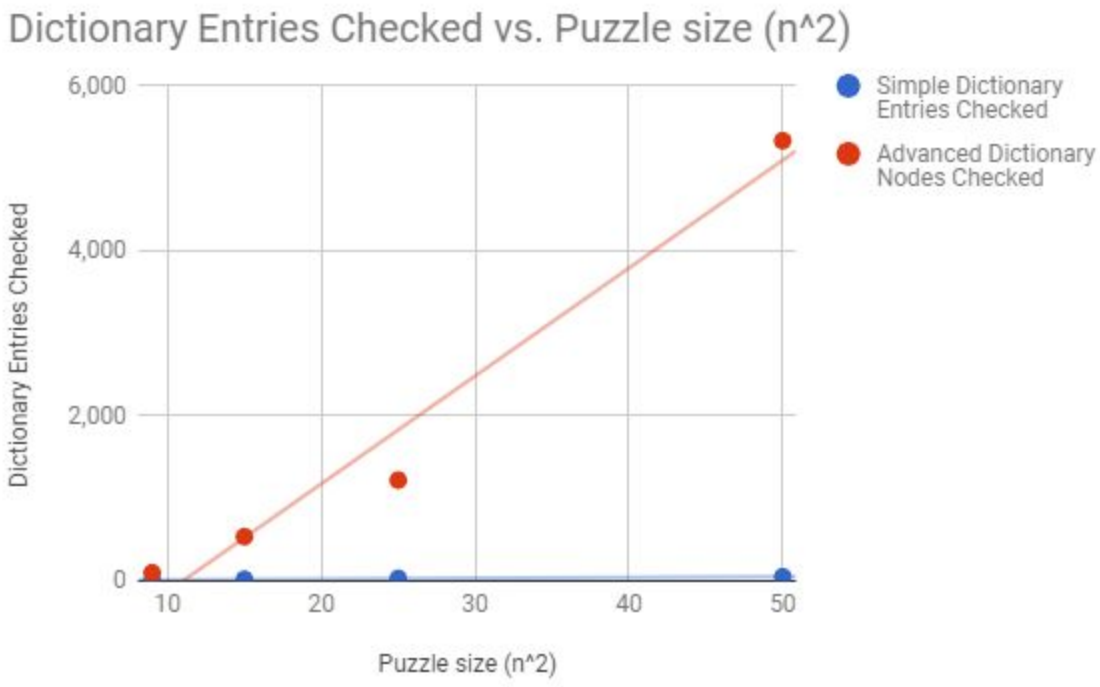
The advanced algorithm greatly outperforms simple one in terms of solve time. It is noteworthy that the advanced solver provides a greater advantage over the simple solver as the size of the dictionary and puzzle increases. This is due to the fact that with more words there will be more redundancies that the advanced dictionary will remove whereas the simple solver will be affected much more severely by redundant entries. As the puzzle and dictionary size increase so does the number of redundant checks the simple solver needs to do.

Chart 3.



More grid cells are visited in the simple dictionary structure because it doesn't acknowledge that words and sub words can share the same origin. Because of this, the simple solver has to step through the grid many more times in order to find multiple words that begin with the same letter, whereas the advanced solver will find them in one run.

Chart 4.



The advanced dictionary appears to go through more dictionary entries when in fact the count is greater because the nodes are counted as dictionary entries. The number of unique dictionary entries is drastically smaller (3 times less checks than the simple dictionary) because the advanced dictionary eliminates redundant entries.

Conclusion

Solve time can be notably improved through the implementation of the advanced algorithm which drastically decreases the number of distinctive dictionary entries. It is apparent that selecting words/nodes from the dictionary and then checking them against each puzzle cell is more efficient than first visiting each letter in the grid and then checking it against each dictionary entry. The reasoning for this is that in the grid there are less letters that match a first letter in any of the dictionary entries. Furthermore it is most likely that the words in the grid do not appear in the order in which they appear in the dictionary. As such, the cost of checking redundant, already found words is introduced. Attempting to mitigate this cost is futile because removing found words from the dictionary once they have been found is also expensive. In the alternative strategy where each dictionary entry is compared against letters in the grid these costs are avoided as the words are discovered in the order which they exist in the dictionary. Given that a word exists in the grid, it is guaranteed to be found in a single check against the

entire puzzle. Therefore words that have been discovered or proven not to exist in the puzzle will not be checked again and redundancies are avoided.

The alternative approach of dictionary to grid search does not require either a simple or advanced algorithm exclusively in order to function as it was implemented in both due to preference over the redundant approach. However, the efficiency of this approach can be leveraged better by the implementation of the advanced dictionary which minimizes the number of unique first letter checks required in order to discover the point of origin of a given dictionary entry which exists within the grid.

Using an advanced dictionary structure provides a performance advantage in terms of solve time at the cost of load time, especially with very large grids and dictionaries where the probability of a lot of words sharing a common prefix is high.