Third Year Project 2013/2014

Report 6 Complexity, NP hardness and the Big O notation

A Sudoku Solver Using Constraint Satisfaction (& Other Techniques)

Thomas Paul Clarke

A report submitted in part fulfilment of the degree of

BSc (Hons) in Computer Science

Supervisor: Dave Cohen

Complexity, NP hardness and the Big O notation

Each program has a runtime and with this comes the complexity of the algorithms used, what the program is trying to do and how effectively it does this, there are cases why programs are simply not able to perform a task due to constraints on time and the speed to which a problem can be solved.

Complexity

The reason to find out and define the complexity of an algorithm to understand the amount of time it will take to solve the problem, this has to be done to really get an idea how long the algorithm will take, this can be done using a timer that can then be used to time a small section of the algorithm, i.e. a recursion and this can be incremented by the amount of times the algorithm could be run in the worst case. [1]

Complexity is not an accurate value is consists of multiplication of scenarios and uses this number to be able to suggest a value to the user on how long it could take, this is also a way to see how long a computer takes to complete the task, in few cases a faster computer can help situations that are within the scope we will be looking at later in this report.

An example of this could be the traveling salesman problem[2] that describes a situation that a salesman travels from several cities and the program will find the shortest route to travel to each city but walk the smallest distance. However each city has to be checked against each other city, this then happens for every city. This problem takes factorial time to be completed, 10 cities will take 10! Time.

Big O notation

Using the big O notation to specify the number of visits on the order n does analyse the performance of an algorithm. The use of the O notation is used to describe how the run time scales in relation to the input of the algorithm. An example of a Big O Notation is O(n!) what this actually means is that for every possible element of n it is explored.

[1]http://www.cosc.canterbury.ac.nz/csfieldguide/student/Complexity%20and%20tractability.html

[2] Time and Space, Designing Efficient Programs. Adrian Johnstone & Elizabeth Scott. Chapter 5,Page 68 – 75

The Class NP

Within this the main aspect to look at is a problem with a large set of possible nodes and a constraint such as time[3] normally an it is beneficial to find the best possible solution to a problem, however this is not always possible proven by the Traveling Salesman Problem. So it is possible to categorise problems that within this by using a threshold to limit the distance travelled between the cities, this limitation can be a used to allow the problem to run better than the threshold, if it is better then the decision problem is true else false. There are problems that can be similar and their complexity can cause them to be unfeasible to run in any practical amount of time, however there are "tractable solutions"[4] that can be adapted to others, and this group is NP-complete. "Complete means they can all be converted into each other"[4] and contains the hardest problems within NP. The be within the NP there needs to be a polynomial time algorithm for any of them, meaning all algorithms within NP have a polynomial time solution.

NP hard

NP hard are a set of problems are not easy to calculate, they are within the set NP-Complete but not within NP or P, this is due to the NP-hard set of problems are within the hardest NP-complete problems. The reason for this set is to distinguish the sets that are hard to be solved, for example the traveling salesman problem is NP-Hard even when it is cut down to only one node to be visited once.

Sudoku Is NP-Complete[5]

Proving that the Sudoku puzzle is within NP-complete[6] was documented in 2002, to summarise these findings, the puzzle contains a n2 x n2 grid that is split into n x n squares, this then contains n values. Each search is done on a value and to find the correct answer if there was to be a search on every cell then there are minimum n-1 cells in each block that could be searched, each with values of n different values. This could take n·n! Time.

That proves that the algorithm is hard, however there is also a case in which the Sudoku size can be reduced for n = 3, this can then be solved by using a backtracking method[57], known as a Latin Square this proves that it belongs in NP class due it its ability to be solved in polynomial time and this can be applied to the Sudoku puzzle as it can be a section of a grid, this can then be performed n times so it will now be O(n!), which is a vast improvement, but still has poor scalability

Conclusion

Finding out this information about the process of how the Sudoku is solved will now change the way the program will have to be designed and made, as now it is known that searching every cell to get the correct answer is not practical as there are two many iterations that will have to be made. It has also given me a secondary program to look into and that is the Latin Squares algorithm.

[3] Algorithms and Complexity 2, CS2870 By Gregory Gutin February 16, 2013. Page 93

[4]http://www.cosc.canterbury.ac.nz/csfieldguide/student/Complexity%20and%20tractability.html
[5] Graph Algorithms and NP-Completeness. Kurt Mehlhorn. Page 171- 211
[6] http://www-imai.is.s.u-tokyo.ac.jp/~yato/data2/SIGAL87-2.pdf
[7] http://www.math.cornell.edu/~mec/Summer2009/Mahmood/More.html