University of BRISTOL

Department of Computer Science

www.cs.bris.ac.uk

What is Computer Science ? # 10

Artificial Intelligence through Search: Solving Sudoku Puzzles

Tim Kovacs

November 12, 2008

"Artificial Intelligence is ten years away . . . and always will be." Anonymous.

1 Introduction

Depending on your point of view you may be disappointed or comforted to know that Artificial Intelligence (AI) of the kind we see in movies, where machines are as intelligent as humans, or more so, is a long way off. There are certain areas, however, including many games and puzzles where machines already far outperform us, and so act in ways which *appear* intelligent. We will see how this can be achieved with little or no intelligence on the part of the machine (although the machine's programmer needs some) by combining fairly simple strategies with modern computing power. If this is disappointing to you a couple of points may be of some comfort. First, the creation of human-like intelligence is one of the hardest, most significant, and, to my mind, most interesting problems there is. So, if you are really interested in the subject there's plenty of interesting work to be done. Second, although we will not go into them here, there are already more advanced AI methods which have more of the attributes we expect of intelligence, such as an ability to learn from experience, though these too are primitive compared to what we see in the movies.

Whether you are interested in the mysteries of human-like intelligence or simply want to write a Sudoku player there are good reasons to study methods which only behave as though they possess intelligence. They're simple and so provide a good place to start. They touch on many core subjects in computer science, such as complexity theory. They provide a foundation for the whole of AI, and more sophisticated methods can be thought of as extensions to them. And finally, the simpler methods are sufficient for many problems; indeed they often outperform more sophisticated ones!

2 Sudoku rules and terms

Sudoku [1] is a puzzle which has recently gained enormous popularity and which some players find so addictive that Ben Laurie has called it "a denial of service attack on human intellect".[1] The objective is to place a set of n symbols on an n x n square *grid* in a way which meets a set of constraints. Usually the grid consists of 9x9 *cells*, although other sizes are used. Each grid is divided into a set of non-overlapping *boxes* and the standard 9x9 grid is composed of nine 3x3 boxes. The constraints on the placement of symbols are that each must occur exactly once in each row, column and box of the grid. The most commonly used symbols are the digits 1 though 9. Sudoku puzzles are usually presented with a certain number of *givens* – cells for which the symbol already appears. Figure 1 contains a completed Sudoku puzzle generated by the software at `http://sudoku.sourceforge.net/`.

A Sudoku solution is a special kind of *Latin square*; a square grid in which each symbol only appears once in each column and row. The difference between the two is that Latin squares have no boxes. Since Sudoku imposes additional constraints the set of Sudoku solutions is a subset of Latin squares. Latin squares have applications in statistics in the design of experiments and in computer science in error correcting codes. Because 9x9 grids are large and complex our examples will use 4x4 Sudoku grids (which have four 2x2 boxes) or 2x2 Latin squares.

3 Theoretical background

Even if you have never played Sudoku and have been given no instructions on strategies, you can, within a few minutes, develop some basic strategies. Sudoku puzzles range in difficulty (for humans) from fairly easy to essentially impossible, so many people have implemented Sudoku-playing programs. Think for a moment about how you would go about doing this. Since human players have worked out various strategies [1] the obvious approach would be to somehow convert them into a program. This has been done, and can result in good solutions. If you enjoy programming and Sudoku you may be tempted to jump in head first and start coding up the strategies in your head but there are a couple of reasons to do a little reading first.

[1]A denial of service attack is an attempt to make a computer service such as a website unavailable, typically by overloading it with demands [2].

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

Figure 1: A completed Sudoku puzzle

Principle Humans and computers have different strengths.

While humans really do have some intelligence, and specialised hardware for processing images and sound, computers do not get bored, have precise memories and excel at doing simple things over and over. This suggests that the best way for a computer to solve Sudoku puzzles is not necessarily the same way humans do.

Principle Don't reinvent the wheel.

Problems and their solutions don't exist in isolation, and ignoring this fact will cost you. Sudoku is a kind of *constraint satisfaction problem* (CSP); a problem whose solution requires a set of variables and which imposes constraints on the values they can take [3]. In Sudoku the variables are the cells and the constraints are that rows, columns and boxes must contain each symbol exactly once. A little reading about constraint satisfaction will introduce you to the various general approaches used to solve these problems, some of which we will cover here.

The many forms of scheduling problems, in which resources are allocated over time, are well-known CSPs. One example is the problem of scheduling the exams for a university. Each course which has an exam must have it assigned to a particular room and time during the exam period; these are the variables of the problem. The basic constraints are fairly self-evident: a room cannot be occupied by too many students at a time and no student can write more than one exam at a time. Scheduling problems also arise in, for example, aircraft maintenance, crew timetabling and the months of procedures required to prepare a space shuttle for launch.

Many such CSPs have been studied and many *algorithms* (solution methods) developed for them. Given that Sudoku is a CSP we should suspect that existing CSP-solving algorithms can be applied to Sudoku, or adapted to it, as indeed they can. Given this, surely the thing to do is to look up the best one and use it. That would be nice, but there's a problem.

Principle There's no one best algorithm for all problems.

Computer science would be a lot simpler if that wasn't true! As it turns out, however, the most efficient algorithm for a given problem is one which is carefully crafted to take advantage of the problem's structure. In particular, we'll see that specialised Sudoku algorithms are more efficient than general-purpose search algorithms. On the other hand, the specialised algorithms are more complex, and may only apply to Sudoku. This leads to another principle.

Principle There's a tradeoff between reusing existing not-very-good solutions (which is easy) and creating efficient custom solutions (which is hard).

This happens in everyday life too: a customised car, clothes, or kitchen will cost you more than the off-the-shelf equivalent. How do we decide whether the customisation is worth the extra cost?

Unfortunately the decision is often complicated by the next principle.

Principle We often want to optimise multiple *mutually-conflicting* criteria.

We are constantly faced with this sort of dilemma in everyday life too. At a restaurant, for example, we may want to optimise two things: how much we enjoy the meal and how little we pay. If my favourite meal is more expensive than the other choices, then I can't maximise both criteria (enjoyment and cost) at the same time [4]. In this situation I need to decide on the relative importance of the criteria (and their relative importance may change – for example if I suddenly get a much better-paying job). Similarly, in selecting algorithms there are many possible criteria including at least the efficiency of the algorithm (how much computer time it uses when running) and the complexity of implementing it (how much programmer time this takes).

3.1 Computational complexity theory

How can we describe the difficulty of Sudoku as a whole, given that some instances (some puzzles) are harder than others? We usually refer to the *worst-case* complexity (the difficulty of the hardest instance) or the *average-case* complexity. We do not normally deal with the *best-case* complexity for the following reasons. Suppose Caroline is paying you to solve Sudoku puzzles (or, more realistically, scheduling problems). If she asks you how long it will take you to solve the current puzzle you should give the worst-case estimate because if the current instance turns out to be particularly easy she will be pleasantly surprised when you finish quickly.[2] This is much better than giving a best-case estimate and finding that you have a very tough instance to work on. Another reason the best-case is not very useful is that many problems have at least one very easy instance so comparing their best case does not tell you much. The average-case complexity is arguably more useful than either of the others but it can be more difficult to work out what it is.

What does it mean for an algorithm to solve a problem efficiently? Informally, you might consider an algorithm efficient enough if you are willing to wait long enough for it to finish, and indeed in more formal analysis the *time complexity* – the time an algorithm needs to run – is usually the issue. In particular, we are interested in how time complexity *scales* as we increase the size of the input to the algorithm. The input to a Sudoku solver is a Sudoku grid. While the standard grid measures 9x9 cells, both smaller and larger grids are used. Note that all 9x9 Sudoku puzzles *can* be solved quickly by computer, but only because this is small for a computer. NP-completeness tells us that Sudoku algorithms do not scale well to larger grids and solving, say, 9000x9000 grids is not feasible.

Although time is the usual consideration, the principle that we often find ourselves wanting to optimise multiple criteria appears again; among other things, we might want to minimise the amount of memory an algorithm requires. (We often have to sacrifice increased memory for reduced time, or vice versa.) For simplicity, however, we'll consider only time here. There are a number of difficulties in measuring the time an algorithm takes to run, including the efficiency of the language used to implement it, the speed of the machine it runs on, how much the programmer has optimised it, and which operating system is used. Consequently we do several things. Instead of measuring time itself we count the number of operations an algorithm specifies. This determines the time needed, but is not affected by the speed of the machine we're using. We also focus on the scalability of the algorithm by using the *Big O notation*, which expresses the number of operations needed as a function of the size of the input [6].

As an example let's calculate the time complexity of printing a Sudoku grid. We won't print the grid lines, just the numbers. The pseudocode is in figure 2. It uses two *for loops*, one inside the other. A for loop repeats itself once for each element in the list it is given. The list given to the first for loop contains the rows of the grid and the list given to the second contains the columns. The scope of each for loop is indicated by indentation: we repeat all indented lines which follow the line which starts the for loop, and we do this once for each element in the for loop's list. Because the second for loop is within the scope of the first we work through the list of columns once for each row. Since each cell has a unique row-column combination this allows us to deal with each cell

[2] Alternatively, if you're careful, you can take the rest of the day off

```
PrintGrid()
    for each row do
        prepare to print a new line
        for each column do
            print cell at this row
            and column
end
```

Figure 2: Pseudocode to print a grid

individually. By "prepare to print a new line" we mean the position of the point at which we print, whether it is on a screen or on paper, moves down one line and back to the left margin. Hopefully the pseudocode seems simpler to you than the English description!

Now let's calculate the number of operations specified by the algorithm. Let's ignore the for loops themselves for reasons which will become clear in a moment and count only the number of printing operations, of which there is one for each cell and one at the start of each row to prepare the new line. Since there are n x $n = n^2$ cells and n rows there are $n^2 + n$ operations and we would say the time complexity of printing an n x n grid is $O(n^2 + n)$. (The O is read as "order" and emphasises the value is approximate.) This is of course an approximation to what really goes on in the computer since we ignored the for loops, but also because each printing operation is quite complex, and preparing to print a new line might be more or less complex than printing a cell, and their relative complexity might depend on the machine and language used. This imprecision is not a problem, however, because our focus is not on exactly how long any particular instance takes, but on how the algorithm scales to larger inputs. To print a 9x9 grid we need $9^2 + 9 = 81 + 9 = 90$ operations of which the n^2 term (printing the cells) accounts for 81 out of 90 operations and the $+n$ (setting up new lines) accounts for 9 of 90 operations. As percentages 90% of operations print cells and 10% print new lines. For comparison, to print a 1000x1000 grid we would need $1000^2 + 1000 = 1,000,000 + 1000 = 1,001,000$ operations, of which about 99.9% print cells and less than 0.1% print new lines. The point is that the n^2 terms comes to dominate the time complexity as we deal with larger inputs and the $+n$ term becomes less and less significant. As a result, in Big O notation we focus only on the dominant terms and simply leave out all other terms. Thus we write the complexity of printing n x n grids as $O(n^2)$. For the same reason we ignore any constants, so if each cell contained a 5-digit number (instead of a 1-digit number) and so $5n^2$ operations were required to print the cells, we would still write $O(n^2)$. Big O notation ignores a lot of details, so what does it consider significant? Figure 3 shows a number of orders of complexity which are significantly different. The horizontal axis shows the values of n ranging between 0 and 10, although to avoid clutter not all the numbers are drawn. The vertical axis shows the time required to compute a solution. The lines show how the time required increases depending on the order of complexity of the algorithm used. The orders plotted are: $O(n)$, $O(n^2)$, $O(n^3)$ and $O(n^n)$, although on the figure we have dropped the O and the brackets to reduce clutter. The point of this figure is to illustrate why we consider these orders to be significantly different: notice how $O(n)$ grows far more slowly, and $O(n^n)$ far more quickly, than the others. In contrast, if we were to add $O(2n)$ to the graph it would not be much different from $O(n)$. In summary, Big O notation is a useful way to gloss over the details which don't matter but still capture the big differences which do. We do something similar when talking about "6-figure" salaries. We can describe both $£100,000$ and $£200,000$ as 6-figure salaries to gloss over the difference between them but still indicate that $£1,000,000$ or $£10,000$ would be significantly different.

Here's a final principle before we proceed to analyse Sudoku:

Principle Faster computers let you solve bigger problem instances in a reasonable amount of time. At some point, however, hard problems still become too big to solve within reasonable time.

We can see this in figure 3. Suppose the vertical axis shows time in hours. Suppose also that

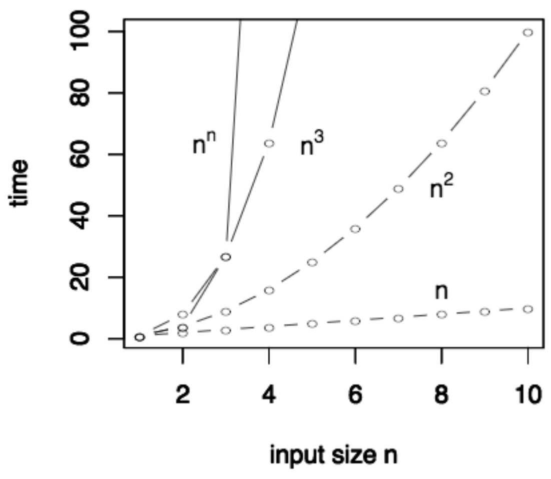

Figure 3: Scaling: how the time needed changes with the size of the input

we're willing to run an algorithm for 20 hours, perhaps because we want to be able to start the calculation at lunch time and have the answer ready the next morning. If you imagine drawing a horizontal line across the figure at the level of 20 hours, all dots below the line represent cases which can be solved within 20 hours. If an algorithm has complexity $O(2^n)$ we can solve problems of at most size $n = 4$ in those 20 hours.

If we buy a computer which is twice as fast then the vertical axis represents the number of half-hours needed and we can indicate our 20-hour time limit by drawing a line at 40 half-hours. Sadly, the faster computer can still only solve problems up to size $n = 6$ within 20 hours with our $O(2^n)$ algorithm. If we buy an even faster computer, one four times as fast as the original, we can still only solve problems up to size $n = 8$ within 20 hours. Of course things are even worse for algorithms with complexity $O(3^n)$ and $O(n^n)$. Even if we had a computer 100 or 1000 times faster we could only solve slightly bigger problems which require a complexity $O(n^n)$ algorithm. On the other hand, if we could find an algorithm which solves the same problem with complexity $O(n)$, the original computer could solve problems of size up to $n = 20$ in 20 hours. This is why, although the speed of computers increases every year, it is more important to find more efficient algorithms – if they exist.

3.2 Mathematical analysis of Sudoku

We can gain insight into the nature of Sudoku and ways of solving it with a little mathematical analysis [7]. Sudoku involves assigning digits to a grid and combinatorics can tell us the number of ways this can be done. For simplicity we start by computing the number of ways of filling an n x n grid with any n digits. We call such a grid an *unconstrained grid* since it ignores the Sudoku constraints that each row, column and box has unique digits. In combinatorics, a *permutation* is an ordering of a sequence of things. There are two kinds of permutations: those which allow repetition of the things in the sequence and those which do not. For example the permutations with repetition of length 2 of the values 0 and 1 are 00, 01, 10 and 11. The permutations without repetition are 01 and 10; naturally there are more permutations if we allow repetition.

An unconstrained Sudoku grid is a permutation of symbols so if we have a formula for the number of permutations we can use it to work out the number of unconstrained grids. This formula is a basic tool of combinatorics and we can work it out for ourselves. If we assume there are no givens the first cell can take n digits. Each of these n can be followed by any other n digits, so there are n x n or n^2 possible pairs of digits. Similarly there are n^3 permutations of 3 digits, and so on. In

n	**Latin squares of size n**
1	1
2	2
3	12
4	576
5	161280
6	812851200
7	61479419904000
8	108776032459082956800
9	5524751496156892842531225600
10	9982437658213039871725064756920320000
11	776966836171770144107444346734230682311065600000

Figure 4: The number of Latin squares of size n

general there are n^l permutations of n digits of length l. On a square grid there are $l = n^2$ cells and hence n^{n^2} such permutations. From a scalability point of view this double exponential is very bad news: the number of unconstrained grids grows very very quickly as the size of the grid increases. There are $2^{2^2} = 16$ unconstrained 2x2 grids, $4^{4^2} = 4,294,967,296 \approx 4 \times 10^9$ unconstrained 4x4 grids and $9^{9^2} \approx 6 \times 10^{86}$ unconstrained 9x9 grids.

With a little more thought we could incorporate at least some of the constraints into our calculation and avoid counting some invalid solutions. To begin, let's include the constraint that each row has only unique digits – this is a permutation without repetition of the possible digits. We will call any filled grid which satisfies this constraint a *row-constrained grid*. Assuming there are no givens, the first cell in the row can take on all n digits. The second, however, can take on only $n-1$ digits, the third $n-2$ digits and so on until the last which must take the only remaining unused digit. In other words, the number of possible rows with unique digits is $n \cdot (n-1) \cdot (n-2) \ldots \cdot 1 = n!$. This is read as "$n$ factorial" and it gives the number of permutations without repetition of length n. Now, since there are n rows in a square grid, there are $n \cdot n!$ row-constrained grids. By adding this one constraint we have reduced the space of possible solutions from n^{n^2} to $n \cdot n!$, or in big O notation, from $O(n^{n^2})$ to $O(n!)$. Even though this is a big improvement, factorial complexity still scales very badly: there are $2 \cdot 2! = 4$ row-constrained 2x2 grids, $4 \cdot 4! = 96$ row-constrained 4x4 grids and $9 \cdot 9! = 3,265,920$ row-constrained 9x9 grids.

Interlude on scientific notation

4×10^9 is an example of *scientific notation*, which we use to more easily write approximations to very large numbers [9]. To convert from scientific notation to regular numbers just take the leading number (in this case 4) and shift its decimal place e places to the right, where e is the value of the exponent (9 in this case). (In this example we did not show the decimal place, but 4 is equal to 4.0.) To convert from regular numbers to scientific notation, first decide how many digits you want to use to represent the number. The more digits you use, the greater the accuracy of the approximation. In this case we threw away all but the leading digit (4) from 4,294,967,296. First, however, we counted how many digits we were throwing away (9) and we used this as the exponent. Scientific notation makes it easier to write approximations to very large numbers and to compare and manipulate them in other ways. We know for example 4 $\times 10^{34}$ is $10^2 = 100$ times bigger than 4 $\times 10^{32}$.

If we add the constraint that each column contains each digit exactly once we have the constraints which define Latin squares. No simple formula for their number is known, although we can use search algorithms to count them. The results for all known grid sizes are shown in figure 4.

In 2005 Felgenhauer and Jarvis [10] computed that there are approximately 6.671×10^{21} valid Sudoku solutions for a 9x9 grid with no givens. (In contrast some puzzles have only one solution, which you can see by deleting one cell from any complete solution.) From the table above we can see there are approximately 5×10^{27} Latin squares of size 9. This shows that the additional constraint imposed by the boxes rules out a great many Latin squares from being solutions to Sudoku.

Incidentally, Felgenhauer and Jarvis used a combination of combinatorics and a backtracking search algorithm. In recent years many mathematical results have been obtained with the help of computers. We will cover backtracking search later.

That's enough theory. Now let's proceed to methods for solving Sudoku puzzles.

4 Solving Sudoku

We have seen that a Sudoku grid can be filled with many configurations of symbols, only a few of which are valid solutions. Solving the puzzle is like searching for a needle in a haystack and indeed the methods we will apply are called search algorithms. We will consider a number of search algorithms. We will start with the simplest, most naive and most inefficient and then progressively improve search by taking the structure of the problem into account. All search algorithms make use of the concept of a *search space*, which is the set of states being searched for a solution – the metaphorical haystack. We'll begin by using the set of unconstrained grids as our search space and then show how we can progressively exclude large parts of this space from consideration.

4.1 Guessing at random

The first algorithm is simply to generate a solution to the puzzle at random (that is, an unconstrained grid), then to check whether it is valid and to repeat until a valid solution is found. There's nothing to stop you from using this algorithm yourself by rolling a set of eighty-one 9-sided dice to obtain a number for each blank cell – nothing apart from boredom and the vast amount of time required. Boredom and shortage of time mean this is not the kind of solution humans use, but as noted earlier humans and machines have different strengths. In fact, this tolerance for endless repetition allows computers to solve many problems in ways mathematicians have overlooked for centuries.

This algorithm has some very nice properties: it's very simple and will work on any search problem. It is even guaranteed to find a valid solution – given enough time – because it will eventually try all of them. The catch of course is the time required. In the worst case it requires infinite time to find a correct solution, because only then is it guaranteed that every possible solution will be generated.

In order for each guess to be made at some point in the infinite future, we need only ensure that all guesses have a non-zero probability. To keep things simple we can give them all equal probability. This is not, however, just a simple way of setting the probabilities; we are specifying no preference among the possible guesses. This brings us to another principle.

Principle The more an algorithm exploits the structure of the problem the better.

Put another way, the more knowledge of the problem you can put into your algorithm, the better it will do. If we make all guesses equally likely we are putting absolutely no knowledge of the problem into the algorithm. On the other hand, if we know something about either the structure of Sudoku puzzles in general, or about the particular puzzle we are trying to solve, we can bias the probabilities to make some guesses more likely than others. Using equal probabilities is the extreme case of putting no knowledge into the solution. At the other extreme, if we already know a solution to the particular Sudoku puzzle at hand we can give that guess probability 1 (and all others probability 0 since probabilities must sum to 1). This finds the solution right away – although we don't gain anything by guessing about what we already know! Our knowledge of a problem is almost always somewhere in between the two extremes: we have some knowledge, and with some work we can exploit it to improve our algorithm.

Although simply guessing is not much use (unless the search space is sufficiently small), other algorithms which incorporate a certain amount of randomness can perform very well. Genetic algorithms [11] are perhaps the best known of such algorithms. The key difference of such algorithms is that they use feedback on the quality of previous guesses to alter the probability of future ones, allowing them to progressively adapt to the problem structure and home in on a solution.

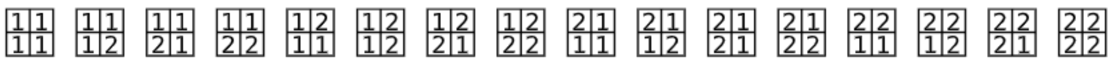

Figure 5: List of all unconstrained 2x2 grids

4.2 Exhaustive search

The next simplest way to search for a valid solution to a puzzle is to enumerate (that is, generate all possible instances of) the unconstrained grids. After generating each new grid you check to see whether it's a valid solution, and if so you're done. This algorithm is guaranteed to find a solution since it will check all grids if need be, but in the worst case the only solution is the last grid it checks. We call this an *exhaustive search* since it exhausts all the possibilities (and anyone who tries to do it by hand). Like guessing, this is an example of *brute-force search*, where instead of solving a problem with a clever (and probably complex) algorithm, we solve it with an inefficient but simple algorithm [12]. We're able to do so because powerful computers are able to solve some problems even with inefficient methods. As we know from section 3.1 this is only possible for easy problems which scale well, and for small instances of hard problems. We also know this will remain the case no matter how fast computers get in the future.

There's a simple, systematic way to enumerate unconstrained grids (or indeed any discrete state space) and you already know it because you use it when counting with written numbers. Suppose a doorman at a nightclub is using a mechanical counter to keep track of how many people have entered the club that night. At the start of the night the counter is set to 0 and each time the increment button is pressed it adds one to the count. If the counter has a 3-digit display the sequence of numbers begins: 000, 001, 002, . . . 009, 010, 011, . . . In other words, the counter enumerates the set of integers, starting with 0 and using 3 digits to represent each integer. (Technically this is the set of natural numbers since we're excluding negative integers.) The algorithm the counter uses has two nice properties. First, to obtain the next integer in the sequence we only need to look at the last integer and increment it by one. In particular, we don't need to remember all the previous integers; remembering the last one suffices. Second, counting generates each integer exactly once (until the counter is reset), which is what we want when searching for a solution.

We can use counting to enumerate unconstrained grids: instead of 3 digits we have n x n cells and instead of the symbols 0-9 we use n symbols. The only other difference is that the string of digits displayed by the counter is 1-dimensional while Sudoku grids are 2-dimensional. The doorman's counter always starts an increment by operating on the rightmost digit. If this is a 9 which needs incrementing, it is reset to 0 and a 1 is carried over to the digit to the left. If this too is a 9 we reset it and carry, and so on. What should we do with the grid? Let's update cells from right-to-left and bottom-to-top because it's easy to see how this is equivalent to the right-to-left order used by the doorman's counter. Imagine we split a 2x2 grid into rows and then append the bottom row to the right of the top row. This gives us a 1-dimensional string of digits like the one the counter uses. Figure 5 shows the unconstrained 2x2 grids enumerated this way. Only 2 are Latin squares and our exhaustive search will find the first on the 7^{th} grid it tries.

4.3 Search trees: visualising the structure of the problem

As stated earlier, the more an algorithm exploits the structure of a problem the better. What structure is there here? For one thing the grid is composed of cells and we can break down the process of generating a complete grid into a series of operations on cells. If you were to enumerate Sudoku grids with a pencil and paper you would do exactly this, by working out and filling in the value of one cell at a time. The key point here is that you construct the grid cell by cell, rather than all at once.[3] Completing the grid one cell at a time is useful because it allows you to make shortcuts which can speed up search enormously.

We can represent the possible ways of constructing a completely filled grid, cell by cell from a blank grid, by drawing a *tree diagram*. Trees diagrams are widely used in computer science, but are perhaps best known to the public for drawing family trees, which show the relationships between

[3]This is such a natural way to complete the grid that it may be difficult to imagine any other way, but in principle if you had enough friends helping you could simultaneously fill in one cell each

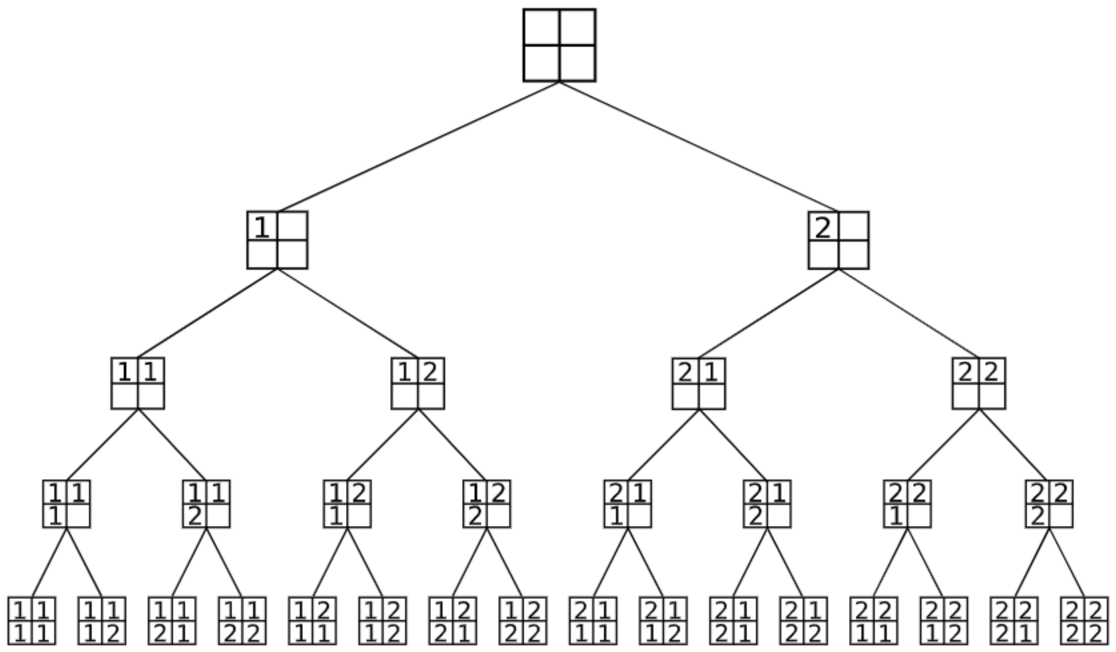

Figure 6: Tree of partially filled and complete 2x2 grids. The leaves are the permutations of the symbols 1 and 2.

people (specifically parents and children). We'll use the same kind of diagram, except we'll show the relationships between states (specifically between states which lead to other states, and states which are led to from other states). A tree is composed of *nodes*, which represent states, and *branches*, which represent the possible transitions between states. The initial state is called the *root* of the tree. If a node A leads to another node B, A is called the *parent* of B and B the *child* of A. Finally, any node without children is called a *leaf* node.

In searching for a solution to a Sudoku puzzle our search space is the set of unconstrained grids, but in constructing one grid, one cell at a time, the states are the partially completed grids, and these are the nodes in our tree. Figure 6 shows the tree of all possible partial and complete unconstrained 2x2 grids. As usual with tree diagrams the root is at the top and the leaves are at the bottom. The root is a blank grid because this is what we start with when constructing a grid. As we descend toward the leaves we add a symbol to a cell at each level. Following convention, in this diagram cells are filled in from left to right, top to bottom, although we could do it in any order. The 16 leaves are the completed grids from figure 5. Of course in standard 9x9 Sudoku the root has 9 branches to 9 children, and each child also has 9 branches, so the tree is much larger.

Incidentally, trees can help understand permutations. In section 3.2 we saw that there are n^l permutations of length l of n digits and figure 6 shows how this permutation can be derived by counting all the possible combinations of digits. If you ever have difficulty with a combinatorial problem it can help to draw the tree, or at least part of it.

Interlude on data structures

Many things in computer science are best understood with the use of diagrams, and this is certainly true for *data structures*, which, as the name suggests, are ways of structuring (organising) data [14]. A *list* is just what you think it is: a collection of data in some order. A *set* is just like a mathematical set: an unordered collection in which each value only appears once. Other data structures like *hash tables*, which map each of a set of *keys* to a value, are more complex and have interesting properties. For example, looking up a key and its value in a hash table is far more efficient than searching for a value in a list.

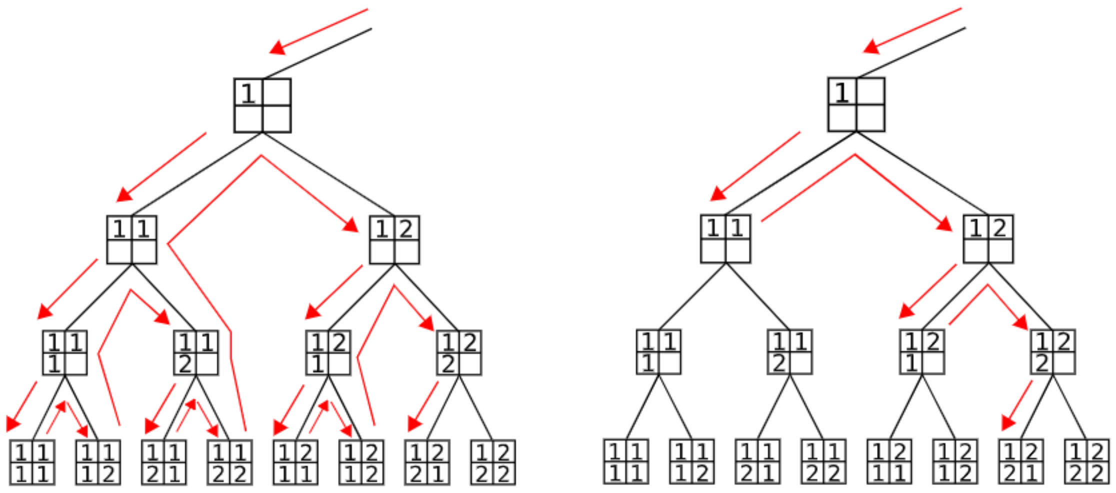

Figure 7: Left: path of depth-first search enumerating the left half of the tree. All nodes but one are visited before a solution is found. Right: path of backtracking depth-first search on the left half of the tree. This is much more efficient than enumerative search.

4.4 Backtracking search: exploiting the structure of the problem

In this section we'll see how incorporating knowledge of the problem can give us much more efficient algorithms, and how thinking in terms of search trees can help us understand how they work. First we will see an algorithm for exhaustive search which is described in terms of a tree. It's called *depth-first search* because if we trace the nodes it visits in the tree, it starts at the top and goes directly to the bottom in order to create the first complete grid. To generate each successive grid it returns to the last incomplete grid it visited which has branches it has not yet followed, and follows these branches, again going all the way down to the next leaf. By continuing this process it works its way through all possible leaves. The left half of figure 7 shows the path it follows while enumerating the left half of the tree from figure 6. Since it exhaustively tries all unconstrained grids it is equivalent to the counting algorithm we saw earlier, and as you can see the 7th grid it visits is a solution.

Now we'll see how we can modify depth-first search so that it does not need to examine all possible grids, even in the worst case. You may wonder how this is possible, since there might be only one solution – how can we be sure we don't need to check them all? The answer is to use the structure of the problem to our advantage. Suppose you are using depth-first search to solve a Sudoku puzzle by hand. As soon as you enter a number which already appears in that row, you know it makes the grid invalid. In fact, no matter what you do with the remaining blank cells, all resulting grids will be invalid. This allows for a shortcut: instead of exhaustively checking all these grids you simply skip over the ones you know will turn out to be invalid. You can do this by returning or *backtracking* [15] to your last decision and making a different choice of symbol. In terms of the tree of partial grids this corresponds to moving upward and trying the next untried branch. Again, this is called backtracking since we return to a previously visited node. The effect is to skip over the sub-trees of any partially completed grid which is invalid.

By backtracking when you detect an invalid row you will only complete row-constrained grids and your search space will be the set of row-constrained grids. As we noted in section 3.2 there are many fewer row-constrained grids than unconstrained grids so it should now be much easier to solve the problem. Specifically, the worst case complexity improves from $O(n^{n^2})$ to $O(n!)$. This is a nice example of how knowledge of the problem structure can substantially improve an algorithm.

The obvious next step is to add more knowledge to depth-first search by ensuring that all grids are both row and column constrained (and hence are Latin squares). But suppose you have the partially-filled grid on the left of figure 8 where you want to fill in the cell with the ? (cell L). Although we can satisfy either the row or column constraint we cannot satisfy both at once. As before the solution is to backtrack and our left-to-right top-to-bottom update order dictates we backtrack one cell to the

1	2	3	4
3	4	2	1
2	3	1	?

A	B	C	D
E	F	G	H
I	J	K	L

Figure 8: Left: a partially filled grid in which search has become stuck and must backtrack more than once before it can continue. Right: labels for the cells

left (to cell K). We increment its value from 1 to 2. Unfortunately, when we enforce both the row and column constraints it's possible that having backtracked to a cell you must then backtrack from that cell also. This is the case here since putting a 2 or 3 in cell K makes both K's row and column invalid, and putting a 4 there forces the use of a 4 in cell L, which makes L's column invalid. The solution is to backtrack another cell to the left and change the 3 to a 1 in cell J, and then continue search in the cells to the right. In this case backtracking two cells was necessary, but in the extreme you may have to backtrack all the way to the first cell you filled.

You may have noticed that using left-to-right top-to-bottom updates, and counting upwards through digits, we would have put a 1 in cell J initially rather than a 3 and so would have avoided the need to backtrack, but this would be only by chance and we would run into the same problem elsewhere. Similarly, if we enumerated cells in the opposite direction (equivalent to counting down rather than up) we would also run into this problem.

By backtracking as much as needed we will generate Latin squares rather than unconstrained (or only row-constrained) grids. In section 3.2 we saw there are approximately 6×10^{86} unconstrained 9x9 grids but only approximately 5×10^{27} Latin squares of that size. This is a big improvement, but the latter is still a very large space to search! We could go farther and constrain it to only generate complete grids which are valid Sudoku solutions by backtracking whenever we reach a partial grid which violates *any* of the Sudoku constraints and this would be even more efficient.

References

[1] Wikipedia: Sudoku.
Available at: http://en.wikipedia.org/wiki/Sudoku

[2] Wikipedia: Denial-of-service attack.
Available at: http://en.wikipedia.org/wiki/Denial_of_service

[3] Wikipedia: Constraint Satisfaction Problem
Available at: http://en.wikipedia.org/wiki/Constraint_satisfaction_problem

[4] Wikipedia: Multiobjective optimization
Available at: http://en.wikipedia.org/wiki/Multiobjective_optimization

[5] Wikipedia: Computational complexity
Available at: http://en.wikipedia.org/wiki/Computational_complexity

[6] Wikipedia: Big O notation
Available at: http://en.wikipedia.org/wiki/Big_o_notation

[7] Wikipedia: Mathematics of Sudoku
Available at: http://en.wikipedia.org/wiki/Mathematics_of_Sudoku

[8] Wikipedia: Permutation
Available at: http://en.wikipedia.org/wiki/Permutation

[9] Wikipedia: Scientific notation
Available at: http://en.wikipedia.org/wiki/Scientific_notation

[10] Mathematics of Sudoku I. Bertram Felgenhauer and Frazer Jarvis. January 25, 2006.
Available at: http://www.afjarvis.staff.shef.ac.uk/sudoku/

[11] Wikipedia: Genetic algorithm
Available at: http://en.wikipedia.org/wiki/Genetic_algorithms

[12] Wikipedia: Brute-force search
Available at: http://en.wikipedia.org/wiki/Brute_force_search

[13] Wikipedia: Depth-first search
Available at: http://en.wikipedia.org/wiki/Depth_first_search

[14] Wikipedia: Data structure
Available at: http://en.wikipedia.org/wiki/Data_structure

[15] Wikipedia: Backtracking
Available at: http://en.wikipedia.org/wiki/Backtracking

Where Next ?

- Sudoku and Latin squares are related to magic squares.

  http://en.wikipedia.org/wiki/Magic_squares

  Hardeep Aiden. *Anything but square: from magic squares to Sudoku.* Plus Magazine. Issue 38, March 2006.
  http://plus.maths.org/issue38/features/aiden/

- We have only scratched the surface of the subject of search algorithms.

  http://en.wikipedia.org/wiki/Search_algorithm

- *Machine Learning* is the branch of AI concerned with algorithms which improve with experience.

  http://en.wikipedia.org/wiki/Machine_learning

- *Constraint Propagation* can dramatically reduce the search space of a puzzle and can be combined with search to solve Sudokus much more quickly than search alone.

  http://en.wikipedia.org/wiki/Constraint_propagation

- Peter Norvig, the director of research at Google, wrote a Sudoku solver to convince his wife that since Sudoku could be solved by computer it did not need any more of her time. He failed to convince her, but the effort produced an elegant and efficient 100-line python program and an essay on how it works. You will need an advanced knowledge of programming to understand the code but the essay is of interest regardless.

  http://norvig.com/sudoku.html

Frequently Asked Questions (FAQ)

This document represents one part of a larger series which attempts to answer the question "What Is Computer Science ?". The intended readership isn't people who already know the answer to this question !

Why are you doing this ? Recruiting students to do Computer Science degrees is quite hard work. Partly this is because school subjects such as ICT often give the wrong impression of what Computer Science is about at University. This document is an attempt to explain what Computer Science is and motivate you to be interested in it. Of course, summing up such a broad subject is difficult; our approach is to showcase specific topics, demonstrating how they relate to other subjects (such as Mathematics) and how they solve real-world problems.

I've got a question or comment. We'd welcome any feedback, experience or comment on these documents both as a way of improving and extending them; to get in contact, email

`uga@cs.bris.ac.uk`

Can I use these documents for something ? We are distributing these documents under the Creative Commons Attribution License

`http://creativecommons.org/licenses/by/2.0/uk/`

This is a fancy way to say you can basically do whatever you want with them (i.e. use, copy and distribute it however you see fit) **as long as** correct attribution of the original source is maintained.

Why are all your references to Wikipedia ? Our goal is to give an easily accessible overview of a topic, so it didn't seem to make sense to reference lots of research papers. There are two reasons why: research papers are often written in a way which makes them hard to read, and although many research papers are available on the Internet, many are not (or have to be paid for). So although there are some valid criticisms of Wikipedia, for introductory material on Computer Science the articles seem a good place to start.

Why don't the examples include much programming ? Our goal is to focus on interesting concepts rather than the mechanics of programming. So even when we include example programs, the idea is to do so in a way where the meaning is fairly clear. For example we'd rather use pseudo-code algorithms or reuse existing software tools than complicate a description by including pages of C program. Even more attractive are interactive environments, for example BASH and Python shells, which allow easy demonstration of intermediate results of small programming steps rather than the monolithic result of a large program.

Some material in this document duplicates others from the series ! This is true, but isn't because we are lazy ! Each document is intended to be fairly self contained; obviously there will be some overlap between topics so there will inevitably be some repetition.