A Polynomial Time and Non-Heuristic Solution for NP-Complete Problems (Clique, TSP & Sudoku) (P=NP)

By

John Archie Gillis

Halifax, NS, Canada

johnarchiegillis@gmail.com

July 19, 2019

Background

The present innovation takes a novel approach to solving NP-Complete problems and provides steps that a computational device and software program can follow to accurately solve NP-class problems without the use of heuristics or brute force methods. The present methods are fast and accurate if utilized properly.

The present inventor states that the solution to solving the "Does P=NP?" question (and our ability to design algorithms to solve such problems efficiently) lies in a novel method presented for searching, filtering, combining and structuring data.

The present innovation also describes a novel method for breaking specific problems into logical groupings that the present inventor (John Archie Gillis) has defined as collaborative variables. They utilize novel binary representations/conversions, so that one can more easily and quickly determine selected and desired informational outputs. In a number of instances for the present system to work, we must organize two or more variables into larger variables (collaborative), so that we can determine logical NOT's. It seems that in many cases that finding constraints by looking at a problem as single variable data points in insufficient. This seems specifically true as will be seen in the TSP problem, as constraints or NOT's are difficult to come by unless two or more variables are joined as one. We can then utilize a system of permutations, logic and searching to find fast and accurate answers.

The present invention provides a number of methods for solving NP-class problems. NP-class problems include many pattern-matching and optimization problems that are of great practical interest, such as determining the optimal arrangement of transistors on a silicon chip, developing accurate financial-forecasting models, or analyzing protein-folding behavior in a cell. Since all the NP-complete optimization problems become easy with the present methods, everything will be much more efficient. Transportation of all forms can now also be scheduled optimally to move people and goods around quicker and cheaper. Manufacturers can improve their production to increase speed and create less waste.

Developments in vision recognition, language comprehension, translation and many other learning tasks will now become much simpler. The present inventor feels that by utilizing the systems of the present invention in numerous fields, that the invention will have profound implications for mathematics, cryptography, algorithm research, artificial intelligence, game theory, internet packet routing, multimedia processing, philosophy, economics and many other fields.

Clique

In computer science, the clique problem is the computational problem of finding cliques (subsets of vertices, all adjacent to each other, also called complete subgraphs) in a graph. It has several different formulations depending on which cliques, and what information about the cliques, should be found. Common formulations of the clique problem include finding a maximum clique (a clique with the largest possible number of vertices), finding a maximum weight clique in a weighted graph, listing all maximal cliques (cliques that cannot be enlarged), and solving the decision problem of testing whether a graph contains a clique larger than a given size.

The clique problem arises in the following real-world setting. Consider a social network, where the graph's vertices represent people, and the graph's edges represent mutual acquaintance. Then a clique represents a subset of people who all know each other, and algorithms for finding cliques can be used to discover these groups of mutual friends. Along with its applications in social networks, the clique problem also has many applications in bioinformatics and computational chemistry.

Most versions of the clique problem are hard. The clique decision problem is NP-complete (one of Karp's 21 NP-complete problems). The problem of finding the maximum clique is stated to be both fixed-parameter intractable and hard to approximate. Most experts in the field have concluded that listing all maximal cliques may require exponential time as there exist graphs with exponentially many maximal cliques.

Most, if not all experts in the field agree that to find a maximum clique, one can systematically inspect all subsets, but this sort of brute-force search is too time-consuming to be practical for networks comprising more than a few dozen vertices. Although (prior to the present paper), no polynomial time algorithm is known for this problem, more efficient algorithms than the brute-force search are known. For instance, the Bron–Kerbosch algorithm can be used to list all maximal cliques in worst-case optimal time, and it is also possible to list them in polynomial time per clique.

While a method for computing the solutions to NP-complete problems using a reasonable amount of time has remained undiscovered (until the publication of the present paper), computer scientists and programmers still frequently encounter NP-complete problems. NP-complete problems are often addressed by using heuristic methods and approximation algorithms. In many circumstances this is a very poor methodology and strategy. The polynomial solution that is presented in the present paper could have enormous impacts on humanity and particularly for human health. The present solution could aid in treatments for Alzheimer's, Cancer and many other illnesses as we will now be able make enormous headway with determining how a protein's amino acid sequence dictates its three-dimensional atomic structure.

The ability to compute solutions for problems such as clique (and many others) has been deemed the holy grail of computational complexity theory and is one of the Millennium Prize Problems. The Millennium Prize Problems are seven problems in mathematics that were stated by the Clay Mathematics Institute in 2000 for which a solution to any individual problem will award the solver $1,000,000.

2 Description of the New Approach

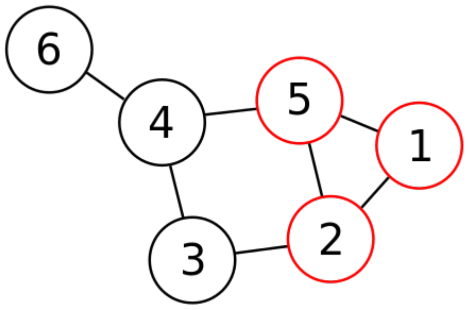
The present paper provides a novel approach to solving NP-Complete problems. We will begin by explaining how the solution and strategies can be applied to the clique problem(s). The author provides a means for greatly reducing the time that it will take a computer (or human) to solve for:

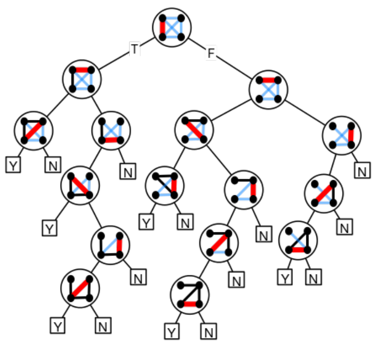
1. Maximum clique (a clique with the largest possible number of vertices),
2. Listing all maximal cliques (cliques that cannot be enlarged), and
3. Solving the decision problem of testing whether a graph contains a clique larger than a given size.
4. Finding cliques of a selected size, particularly largest cliques.

To solve the clique problem, we must discard previous graphing methods and start from scratch with a new strategy. We know that traditional methods have not worked.

The present novel method requires that the data and each of its variables be converted into a binary system so that each and every permutation can be accounted for and compared to each other in a perfectly and elegantly logical way. In the first of a number of steps in our new method, each variable will take on both a single binary place value, such as 1, 2, 4, 8, 16, 32, 64 etc. and also a numerical value that shows the single variables relationship to all other variables in the data set.

By organizing our data in this way we can account for EVERY possible permutation that might occur, no matter what the size of the input. This being stated, the present methods run much deeper than a simple conversion to a Boolean incidence matrix.





Present methods of working with graphs as seen above, are one of the problems why finding items, such as a largest clique, or clique(s) of a fixed size is a real problem for computational devices. Attempting to solve a clique problem of any significant size by utilizing these methods seems a very difficult, if not impossible endeavor.

I will use a social network to explain how the present system provides a better method for communicating with computational devices for the purpose of solving this difficult problem. A system must be employed that will allow a computer to sort and search for a requested output. Some algorithms for certain questions and inputs (particularly NP-Class problems) can in many instances (with many variables) take longer than the age of the universe to run. This is obviously not acceptable.

Mathematicians generally order elements of clique in an ordered progression 1, 2, 3, 4, 5, 6, etc., but herein lies the problem when trying to find specific size cliques or groups within the larger structure. The present invention matches a binary place valued number to the variables, which in this example are names prior to computation.

As an example;

John	becomes 1
Sue	becomes 2
Bob	becomes 4
Jenn	becomes 8
Colin	becomes 16
Maggy	becomes 32
Jim	becomes 64
Kelly	becomes 128

<u>TABLE 1</u>

CLIQUE		Kelly	Jim	Maggy	Colin	Jenn	Bob	Sue	John	
		128	64	32	16	8	4	2	1	
1	John	1	1	1	1	1	1	1	1	**255**
2	Sue	1	1	1	1	1	1	1	1	**255**
4	Bob	0	0	1	1	1	1	1	1	**63**
8	Jenn	0	0	1	1	1	1	1	1	**63**
16	Colin	0	0	0	1	1	1	1	1	**31**
32	Maggy	0	0	1	0	1	1	1	1	**47**
64	Jim	0	1	0	0	0	0	1	1	**67**
128	Kelly	1	0	0	0	0	0	1	1	**131**
		131	**67**	**47**	**31**	**63**	**63**	**255**	**255**	

The above table provides an example for only one instance of clique. It is a very simple version of clique with a very small input of only eight individuals. Most instances will have many more variables to be sorted, but for simplicity we will use this very basic example.

In the provided example of Table 1; John has been assigned the binary variable 1 and Sue 2. John and Sue are then further assigned the number 255. 255 shows their relationship to themselves, each other and everybody else in the data set.

By following this logic, we can also see that IF an individual is assigned a number 254, 253, 251, 247, 239, 223, 191 or 127, then they will be friends with six people, themselves and enemies with one. We can see this clearly in the binary representations. 11111110, 1111101, 11111011, 11110111, 11101111, 11011111, 10111111, 01111111. These binary numbers represent both themselves and all their friendships within the data set. A high or low number doesn't mean that an individual has more or less friends, but it instead tells us exactly who their friends are.

In a small data set like the one described here, one can easily start to see specific cliques, but when the data sets get larger and larger it becomes much more difficult to sort these data sets into various cliques.

By introducing what will be described as a "**Collaborative Variable**", we will be able to sort data sets into cliques much more efficiently than what is possible with presently available methods.

In the diagram below we sort our data via groups of two friends being friends with each individual. For example, John & Sue (A&B) are friends with everybody. Jim & Sue cannot be collaboratively friends with Kelly, even though Sue is friends with Kelly. Because Jim is not a friend of Kelly's we must utilize a 0 rather than a 1 to notate their relationship.

			Kelly	Jim	Maggy	Colin	Jenn	Bob	Sue	John	
a	b	(John & Sue)	1	1	1	1	1	1	1	1	**255**
a	c	(John & Bob)	0	0	1	1	1	1	1	1	**63**
a	d	(John & Jenn)	0	0	1	1	1	1	1	1	**63**
a	e	(John & Colin)	0	0	0	1	1	1	1	1	**31**
a	f	(John & Maggy)	0	0	1	0	1	1	1	1	**47**
a	g	(John & Jim)	0	1	0	0	0	0	1	1	**67**
a	h	(John & Kelly)	1	0	0	0	0	0	1	1	**131**
b	c	(Sue & Bob)	0	0	1	1	1	1	1	1	**63**
b	d	(Sue & Jenn)	0	0	1	1	1	1	1	1	**63**
b	e	(Sue & Colin)	0	0	0	1	1	1	1	1	**31**
b	f	(Sue & Maggy)	0	0	1	0	1	1	1	1	**47**
b	g	(Sue & Jim)	0	1	0	0	0	0	1	1	**67**
b	h	(Sue & Kelly)	1	0	0	0	0	0	1	1	**131**
c	d	(Bob & Jenn)	0	0	1	1	1	1	1	1	**63**
c	e	(Bob & Colin)	0	0	0	1	1	1	1	1	**31**
c	f	(Bob & Maggy)	0	0	1	0	1	1	1	1	**47**
c	g	(Bob & Jim)	0	0	0	0	0	0	1	1	**3**
c	h	(Bob & Kelly	0	0	0	0	0	0	1	1	**3**
d	e	(Jenn & Colin)	0	0	0	1	1	1	1	1	**31**
d	f	(Jenn & Maggy)	0	0	1	0	1	1	1	1	**47**
d	g	(Jenn & Jim)	0	0	0	0	0	0	1	1	**3**
d	h	(Jenn & Kelly)	0	0	0	0	0	0	1	1	**3**
e	f	(Colin & Maggy)	0	0	0	0	1	1	1	1	**15**
e	g	(Colin & Jim)	0	0	0	0	0	0	1	1	**3**
e	h	(Colin & Kelly)	0	0	0	0	0	0	1	1	**3**
f	g	(Maggy & Jim)	0	0	0	0	0	0	1	1	**3**
f	h	(Maggy & Kelly)	0	0	0	0	0	0	1	1	**3**
g	h	(Jim & Kelly)	0	0	0	0	0	0	1	1	**3**

After we notate who is collaboratively friends with whom, we will then sort our data into the outputted numbers.

			Kelly	Jim	Maggy	Colin	Jenn	Bob	Sue	John	
a	b	(John & Sue)	1	1	1	1	1	1	1	1	**255**
a	h	(John & Kelly)	1	0	0	0	0	0	1	1	**131**
b	h	(Sue & Kelly)	1	0	0	0	0	0	1	1	**131**
a	g	(John & Jim)	0	1	0	0	0	0	1	1	**67**
b	g	(Sue & Jim)	0	1	0	0	0	0	1	1	**67**
a	c	(John & Bob)	0	0	1	1	1	1	1	1	**63**
a	d	(John & Jenn)	0	0	1	1	1	1	1	1	**63**
b	c	(Sue & Bob)	0	0	1	1	1	1	1	1	**63**
b	d	(Sue & Jenn)	0	0	1	1	1	1	1	1	**63**
c	d	(Bob & Jenn)	0	0	1	1	1	1	1	1	**63**
a	f	(John & Maggy)	0	0	1	0	1	1	1	1	**47**
b	f	(Sue & Maggy)	0	0	1	0	1	1	1	1	**47**
c	f	(Bob & Maggy)	0	0	1	0	1	1	1	1	**47**
d	f	(Jenn & Maggy)	0	0	1	0	1	1	1	1	**47**
a	e	(John & Colin)	0	0	0	1	1	1	1	1	**31**
b	e	(Sue & Colin)	0	0	0	1	1	1	1	1	**31**
c	e	(Bob & Colin)	0	0	0	1	1	1	1	1	**31**
d	e	(Jenn & Colin)	0	0	0	1	1	1	1	1	**31**
e	f	(Colin & Maggy)	0	0	0	0	1	1	1	1	**15**
c	g	(Bob & Jim)	0	0	0	0	0	0	1	1	**3**
c	h	(Bob & Kelly)	0	0	0	0	0	0	1	1	**3**
d	g	(Jenn & Jim)	0	0	0	0	0	0	1	1	**3**
d	h	(Jenn & Kelly)	0	0	0	0	0	0	1	1	**3**
e	g	(Colin & Jim)	0	0	0	0	0	0	1	1	**3**
e	h	(Colin & Kelly)	0	0	0	0	0	0	1	1	**3**
f	g	(Maggy & Jim)	0	0	0	0	0	0	1	1	**3**
f	h	(Maggy & Kelly)	0	0	0	0	0	0	1	1	**3**
g	h	(Jim & Kelly)	0	0	0	0	0	0	1	1	**3**

Take notice of how awesome this is! For small data sets it seems like a lot of work, but as will be seen shortly, the efficiency of the method really starts to shine as the input size increases as will be described below.

The red data is deemed irrelevant as each of these initial two people were not friends to begin with. We could have eliminated them initially if we wanted. We sort our data into groups by looking at the furthest left hand side list. The data tells us that (255 John & Sue are friends with everyone), (131 Kelly, Sue and John are a 3 clique), (67 Jim, Sue and John are a 3 clique), (63 Jenn, Bob, Sue and John are a 4 clique), (47 Maggy, Bob, Sue, Jenn & John are a 5 clique), (31 Colin, Jenn, Bob, Sue, John are a 5 clique).

The data on the left hade side tells us the people that are in each specific clique. We can now easily see our largest cliques and any unique cliques of 3 or more. Other cliques not listed can easily be extrapolated. As an example Jenn, Bob, Sue and Maggy are clearly a four clique because they also belong to the 5 clique of Jenn, Bob, Sue, Maggy & John. This seems self-explanatory.
So why is this more efficient? It seems like a lot of work.

To find a maximum clique, one can systematically inspect all subsets, but this sort of brute-force search is too time-consuming to be practical for networks comprising more than a few dozen vertices. Although no polynomial time algorithm ~~is~~ **(was)** known for this problem, more efficient algorithms than the brute-force search are known. For instance, the Bron–Kerbosch algorithm can be used to list all maximal cliques in worst-case optimal time, and it is also possible to list them in polynomial time per clique.

In computer science, brute-force search or exhaustive search, also known as generate and test, is a very general problem-solving technique and algorithmic paradigm that consists of systematically enumerating all possible candidates for the solution and checking whether each candidate satisfies the problem's statement.

I will provide an example wherein the question is to list all cliques of three or more people. This is a difficult NP-Complete problem. So why is the presently described system better? To find all cliques one needs to apply a brute force methodology of trying all possibilities. The growth rate becomes exponential in nature. The presently described methods are polynomial and grow at a much smaller rate as the input size increases.

3 people	8	2+1=3	3x3=	9		Less efficient
4 people	16	3+2+1=6	6x4=	24		Less efficient
5 people	32	4+3+2+1=10	10x5=	50		Less efficient
6 people	64	5+4+3+2+1=15	15x6=	90		Less efficient
7 people	128	6+5+4+3+2+1=21	21x7=	147		Less efficient
8 people	256	7+6+5+4+3+2+1=28	28x8=	224	1.14	times more efficient!
9 people	512	etc… 36	etc…9	324	1.58	times more efficient!
10 people	1024	45	10	450	2.28	times more efficient!
11 people	2048	55	11	605	3.39	times more efficient!
12 people	4096	66	12	792	5.17	times more efficient!
13 people	8192	78	13	1014	8.08	times more efficient!
14 people	16384	91	14	1274	12.86	times more efficient!
15 people	32768	105	15	1575	20.81	times more efficient!
16 people	65536	120	16	1920	34.13	times more efficient!
17 people	131072	136	17	2312	56.69	times more efficient!
18 people	262144	153	18	2754	95.19	times more efficient!
19 people	524288	171	19	3249	161.37	times more efficient!
20 people	1048576	190	20	3800	275.94	times more efficient!
21 people	2097152	210	21	4410	475.54	times more efficient!
22 people	4194304	231	22	5082	825.33	times more efficient!
23 people	8388608	253	23	5819	1,441.59	times more efficient!
24 people	16777216	276	24	6624	2,532.79	times more efficient!
25 people	33554432	300	25	7500	4,473.92	times more efficient!
26 people	67108864	325	26	8450	7,941.88	times more efficient!
27 people	134217728	351	27	9477	14,162.47	times more efficient!
28 people	268435456	378	28	10584	25,362.38	times more efficient!
29 people	536870912	406	29	11774	45,598.01	times more efficient!
30 people	1073741824	435	30	13050	82,279.07	times more efficient!
31 people	2147483648	465	31	14415	148,975.63	times more efficient!
32 people	4294967296	496	32	15872	270,600.26	times more efficient!
33 people	8589934592	528	33	17424	492,994.41	times more efficient!

In the diagram above we see that brute force is more effective and efficient until we hit the 8 people mark, wherein the novel described new methods start to become extremely more effective and efficient. Our system is over twice as efficient for sorting cliques with an input of 10 people and over 1000x more efficient for sorting cliques of 23 people. It is almost a half a million times more efficient for an input size of 33 people.

https://jamesmccaffrey.wordpress.com/2011/06/24/the-maximum-clique-problem/

"It turns out that finding the maximum clique for graphs of even moderate size is one of the most challenging problems in computer science. The problem is NP-complete which means, roughly, that every possible answer must be examined. Suppose we have a graph with six nodes. First we'd try to see if all six nodes form a clique. There is Choose (6,6) = 1 way to do this. Next we'd examine all groups of five nodes at a time; Choose (6,5) = 6 ways. And so on, checking Choose (6,4) = 15, Choose (6,3) = 20, Choose (6,2) = 15, and Choose(6,1) = 6 possible solutions for a total of 63 checks. (For the maximum clique problem we can stop when we find the largest clique so let's assume that on average we'd have to go through about one-half of the checks).

The total number of checks increases very quickly as the size of the graph, n, increases. For n = 10 there are 1,023 total combinations.

For n = 20 there are 1,048,575 combinations. But for n = 1,000 there are 10,715,086,071,862,673,209,484,250,490,600,018,105,614,048,117,055,336,074,437,503,883,703,510,511,249,361,224,931,983,788,156,958,581,275,946,729,175,531,468,251,871,452,856,923,140,435,984,577,574,698,574,803,934,567,774,824,230,985,421,074,605,062,371,141,877,954,182,153,046,474,983,581,941,267,398,767,559,165,543,946,077,062,914,571,196,477,686,542,167,660,429,831,652,624,386,837,205,668,069,375, combinations. Even if you could perform one trillion checks per second it would take you 3.4 x 10^281 years which is insanely longer than the estimated age of the universe (about 1.0 x 10^10 = 14 billion years)."

- James McCaffrey

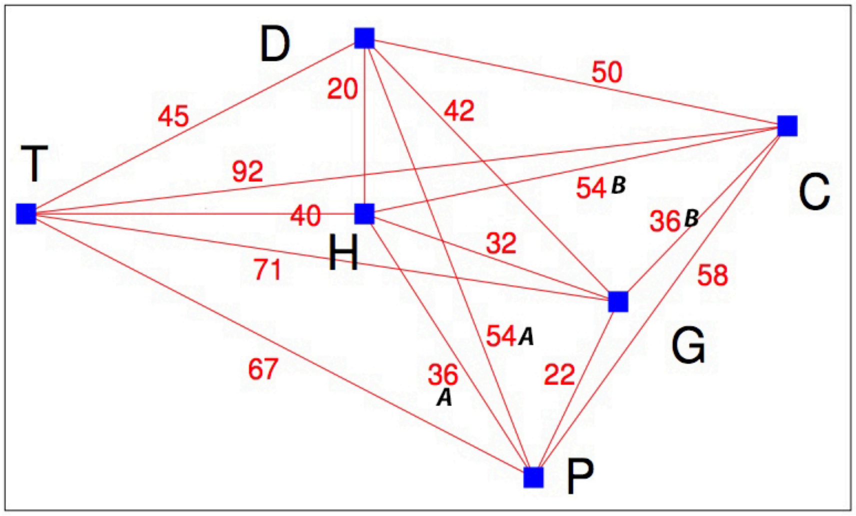
THE TRAVELING SALESMAN

The travelling salesman problem (TSP) asks the following question: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?" It is an NP-hard problem in combinatorial optimization, important in operations research and theoretical computer science.

The problem was first formulated in 1930 and is one of the most intensively studied problems in optimization. It is used as a benchmark for many optimization methods. Even though the problem is computationally difficult, a large number of heuristics and exact algorithms are known, so that some instances with tens of thousands of cities can be solved completely and even problems with millions of cities can be approximated within a small fraction of 1%.

The TSP has several applications even in its purest formulation, such as planning, logistics, and the manufacture of microchips. Slightly modified, it appears as a sub-problem in many areas, such as DNA sequencing. In these applications, the concept city represents, for example, customers, soldering points, or DNA fragments, and the concept distance represents travelling times or cost, or a similarity measure between DNA fragments. The TSP also appears in astronomy, as astronomers observing many sources will want to minimize the time spent moving the telescope between the sources. In many applications, additional constraints such as limited resources or time windows may be imposed.

TABLE 9



Note:

The above TSP example is from:

http://people.ku.edu/~jlmartin/courses/math105-F11/Lectures/chapter6-part3.pdf

Sabrina has the following list of errands and needs to find the shortest route:

(H) Home (The start and finish of her route).

(P) Pet store

(G) Greenhouse

(C) Cleaners

(D) Drugstore

(T) Target Store

The present invention can attack the TSP in a variety of ways! I will describe a couple of examples out of many possibilities with the present system. I begin by making sure that if there is a distance that is the same as another distance, that I can distinguish them. I do this by adding A or B etc., to the number. In the example provided we will see 36A and 36B and 54A and 54B. This is a critical step.

Each unique value is then assigned a binary number based on its size. The shortest path element receives the smallest binary number and vice versa. In the example provided we see that length 20 (the shortest) is linked to binary digit 1 and that length 92 (the longest) is linked to binary digit 16384.

We then need to create a situation wherein we can produce a number of logical "NOT" scenarios. To do this (in this example) we will use two or more variables together (collaborative variable). For example, 22 and 32 are connected at G and thus can become a single larger variable with a number of NOT's. We can also call this HGP as these are the nodes that they are connected to. HPG can also double as PGH. We know that H stands for HOME and thus it will be the beginning and the end point of the journey. This tells us that in the necessary six-point journey (travel to each place once from home and also end at home via the shortest route), that we will require two different collaborative variables from TABLE 10 and one from TABLE 11. All three will work together to create our shortest path(s). (Not in the present example, but in some instances there could potentially be two different shortest paths of the same length).

Using this logic, we can start taking notes of our logical NOT's. Looking at our first collaborative variable we can see that it is comprised of nodes HGP (32 & 22). Of course HGP reversed is also PGH (22 & 32). Being the same, neither can include 36A in its path from or to home because it would make the person on the journey visit the same place twice, which is not allowed in the TSP problem. If we look at the G node in this first collaborative variable, we can see that 32 and 22 are already used and 71, 36B and 42 can NOT be used because G has to go to P and H and thus these values are scratched or deleted as possibilities. Any value from P (22 already being used), is also scratched (67, 36A, 54A and 58), because logic states that we will need to use a number from Table 11 (with no H) for our middle collaborative variable and because any number from Table 10 would want to go straight home in two steps, which is NOT allowed.

<u>TABLE 10</u>

	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1		
	92	71	67	58	54B	54A	50	45	42	40	36B	36A	32	22	20		
H G P	1	🟥	🟥	🟥	1	🟥	1	1	🟥	1	🟥	🟥	🟩 1	🟩 1	1	6	17825
H P G	1	🟥	🟥	🟥	1	🟥	1	1	🟥	1	🟥	🟩 1	🟥	🟩 1	1	10	17825
H G C	🟥	🟥	1	🟥	🟥	1	🟥	1	🟥	1	🟩 1	1	🟩 1	🟥	1	20	4777
H D G	1	🟥	1	1	1	🟥	🟥	🟥	🟩 1	1	🟥	1	🟥	🟥	🟩 1	65	23592
H G D	1	🟥	1	1	1	🟥	🟥	🟥	🟩 1	1	🟥	1	🟩 1	🟥	🟥	68	23592
H D T	🟥	🟥	🟥	1	1	🟥	🟥	🟩 1	🟥	🟥	1	1	1	1	🟩 1	129	3102
H T D	🟥	🟥	🟥	1	1	🟥	🟥	🟩 1	🟥	🟩 1	1	1	1	1	🟥	160	3102
H D C	🟥	1	1	🟥	🟥	🟥	🟩 1	🟥	🟥	1	🟥	1	1	1	🟩 1	257	12334
H D P	1	1	🟥	🟥	1	🟩 1	🟥	🟥	🟥	1	1	🟥	1	🟥	🟩 1	513	25652
H P D	1	1	🟥	🟥	1	🟩 1	🟥	🟥	🟥	1	1	🟩 1	1	🟥	🟥	520	25652
H C G	🟥	🟥	1	🟥	🟩 1	1	🟥	1	🟥	1	🟩 1	1	🟥	🟥	1	1040	4777
H C D	🟥	1	1	🟥	🟩 1	🟥	🟩 1	🟥	🟥	1	🟥	1	1		🟥	1280	12334
H P C	🟥	1	🟥	🟩	🟥	🟥	🟥	1	1	1	🟥	🟩 1	1	🟥	1	2056	8421
H C P	🟥	1	🟥	🟩 1	🟩 1	🟥	🟥	1	1	1	🟥	🟥	1	🟥	1	3072	8421
H P T	🟥	🟥	🟩 1	🟥	1	🟥	1	🟥	1	🟥	1	🟩 1	1	🟥	1	4104	1365
H T P	🟥	🟥	🟩 1	🟥	1	🟥	1	🟥	1	🟩 1	1	🟥	1	🟥	1	4128	1365
H G T	🟥	🟩 1	🟥	1	1	1	1	🟥	🟥	🟥	🟥	1	🟩 1	🟥	1	8196	3849
H T G	🟥	🟩 1	🟥	1	1	1	1	🟥	🟥	🟩 1	🟥	1	🟥	🟥	1	8224	3849
H T C	🟩 1	🟥	🟥	🟥	🟥	1	🟥	🟥	1	🟩 1	🟥	1	1	1	1	16416	591
H C T	🟩 1	🟥	🟥	🟥	🟩 1	1	🟥	🟥	1	🟥	🟥	1	1	1	1	17408	591

We know that the green number 6 (the top number of the 2nd column from the right) of TABLE 10 must be 4 and 2 as binary numbers 4 and 2 are the only ones that will work to make 6. We also know that the white numbers are the ONLY compatible areas where the last two numbers for the journey can be found. TABLE 10 needs to provide both the first two and last two steps of the journey, due to the fact that the journey is six steps long. The middle two steps (two collaborative or added variables) will be provided from TABLE 11. In the present example we see that 6 and 129 are compatible and add up to **135.** 6 (a green number composed of two binary numbers 4 and 2) is a sub-group of 3102 (a white number composed of six binary numbers that also includes 4 and 2) and thus there is no conflict. They are compatible. Perhaps we can call them friends, like in the clique problem.

A computer program can easily determine this by consulting a list or program with the necessary elements. We can for example provide the software program with a list of compatible numbers. Rather than brute force however, the computer can simply look at the one number sums provided by our system to do its sorting, which will be much more efficient. Although it may seem like (and it is) a lot of work when dealing with only 6 travel points, we must remember that our system does not get much more complicated as we begin to add new data points, whereas the presently used brute force method gets exponentially more difficult. For a small number of cities, the present methods may seem slower than brute force, but as the number of cities or data points grows larger, our methods become dramatically better and faster than other available methods.

For example:

6 =	4, 2
7 =	4, 2, 1
14 =	4, 2, 8
22 =	4, 2, 16
15 =	4, 2, 8, 1
31 =	4, 2, 1, 8, 16
3102 =	4, 2, 2048, 1024, 16, 8

…all belong to a family of binary numbers that contain 4 and 2. This will dramatically assist a computing device with its sorting and searching methods.

6 and 160 are also compatible, which add up to **166**. 10 and 129 are compatible and add up to **139**. 10 and 160 are also compatible and add up to **170**. We will need to find their compatibility with Table 11 before we can make a determination of which will be the shortest path however.

…go to next page…

TABLE 11

			16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1	
			92	71	67	58	54B	54A	50	45	42	40	36B	36A	32	22	20	
C	G	P		1								1	1		1	1		18
D	G	P		1							1		1		1	1		66
D	G	C		1							1		1		1	1		80
T	D	G						1	1	1	1						1	192
D	C	G	1			1	1		1				1					272
C	D	G						1	1	1	1						1	320
T	D	C						1	1	1	1						1	384
D	P	G			1	1		1						1		1		514
T	D	P						1	1	1	1						1	640
C	D	P						1	1	1	1						1	768
C	P	G			1	1		1						1		1		2050
G	C	P	1			1	1		1				1					2064
D	C	P	1			1	1		1				1					2304
D	P	C			1	1		1						1		1		2560
T	P	G			1	1		1						1		1		4098
G	D	P			1			1		1	1						1	4160
D	T	P	1	1	1					1		1						4224
T	P	D			1	1		1						1		1		4608
T	P	C			1	1		1						1		1		6144
T	G	P		1							1		1		1	1		8194
T	G	C		1							1		1		1	1		8208
T	G	D		1							1		1		1	1		8256
D	T	G	1	1	1					1		1						8320
G	T	P	1	1	1					1		1						12288
T	C	G	1			1	1		1				1					16400
D	T	C	1	1	1					1		1						16512
T	C	D	1			1	1		1				1					16640
T	C	P	1			1	1		1				1					18432
C	T	P	1	1	1					1		1						20480
C	T	G	1	1	1					1		1						24576

We can then look at TABLE 11 to determine our shortest path!

TABLE 12 (example with more constraints/NOT's)

			16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1	
			92	71	67	58	54B	54A	50	45	42	40	36B	36A	32	22	20	
C	G	P		1		1	1			1	1	1	1	1	1	1		18
D	G	P		1				1			1		1	1	1	1	1	66
D	G	C	1	1		1	1		1		1		1		1	1	1	80
T	D	G	1	1	1			1	1	1	1	1			1		1	192
D	C	G	1		1	1	1		1		1		1		1		1	272
C	D	G			1		1	1	1	1	1		1		1		1	320
e	t	c.	etc.	etc.	etc.	etc.	etc.	etc.	etc.	etc.	etc.	etc.	etc.	etc.	etc.	etc.	etc.	etc.
			...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...

TABLE 13

			16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1		
			92	71	67	58	54B	54A	50	45	42	40	36B	36A	32	22	20		
H	G	P		1	1	1		1			1		1	1	1	1		6	32+22+
H	D	T	1	1	1			1	1	1	1	1					1	129	45+20+
T	C	P	1			1	1		1				1					18432	92+58=
																		18567	**269**

			16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1		
			92	71	67	58	54B	54A	50	45	42	40	36B	36A	32	22	20		
H	G	P		1	1	1		1			1		1	1	1	1		6	32+22+
H	T	D	1	1	1			1	1	1	1	1					1	160	45+40+
D	C	P	1			1	1		1				1					2304	58+54=
																		2470	**261**

			16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1		
			92	71	67	58	54B	54A	50	45	42	40	36B	36A	32	22	20		
H	P	G		1	1	1	1	1					1	1	1	1		10	36+22+
H	D	T	1	1	1			1	1	1	1	1					1	129	45+20+
T	C	G	1			1	1		1				1					16400	92+36=
																		16539	**251**

			16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1		
			92	71	67	58	54B	54A	50	45	42	40	36B	36A	32	22	20		
H	P	G		1	1	1	1	1					1	1	1	1		10	36+22+
H	T	D	1	1	1			1	1	1	1	1					1	160	45+40+
D	C	G	1			1	1		1				1					272	50+36=
																		442	**229**

In this example of the TSP, the example provided states that the traveler must begin at Home (H) and end at Home (H). This information actually provides us with information that allows us to make a number of logical assumptions straight away. If we did not know the starting and ending points we would have needed 60 groupings (HTD, HPG, etc.), but because we have this information we only required 50. For example, we know that we will never see a T**H**D or a C**H**G or the like, because H can never be in the middle, no matter what the situation. This eliminated its permutation as a possibility that the computational device will no longer need to check.

In an alternative embodiment to the examples just explained above in Tables 10-13 and to demonstrate why the present system, method and computing device that utilizes such a system is far superior than present day systems, (and to keep it simple), I will provide an alternative example wherein we do not know our starting point. As described below we can see that the standard brute force method would require checking 120 different paths for 6 cities.

http://www.businessinsider.com/p-vs-np-millennium-prize-problems-2014-9

"If we have a Traveling Salesman Problem with five cities, we have 4 × 3 × 2 × 1 = 24 paths to look at. If we have six cities, we have 5 × 4 × 3 × 2 × 1 = 120 paths.
As we can already see with these small numbers of cities, the number of paths grows extremely quickly as we add more cities. While it's still easy to take a given path and find its length, the sheer number of possible paths makes our brute-force approach untenable. By the time we have 30 cities, the number of possible paths is about a 9 followed by 30 zeros. A computer that could check a trillion paths per second would take about 280 billion years to check every path, about 20 times the current age of the universe.

There are algorithms for the Traveling Salesman Problem that are much more efficient than this brute-force approach, but they all either provide some kind of approximate "good enough" solution that might not be the actual shortest path, or still have the number of needed calculations grow exponentially with the number of cities, taking an unacceptably long time for large numbers of cities. There is no efficient, polynomial time algorithm known for the problem."

---Andy Kiersz

The present invention initially cuts this in half thus requiring only 60 variables to be analyzed, sorted and ordered into paths that satisfy the needed constraints of the TSP problem. For these small numbers the brute force system is likely better and faster, but as these numbers grow we can see how the present system becomes dramatically superior.

TABLE 14

Number of Cities	Brute Force Paths to Check	Varibles in the present system				
6	120	5 x	4 x	3	EQUALS	60
7	720	6 x	5 x	4	EQUALS	120
8	5,040	7 x	6 x	5	EQUALS	210
9	40,320	8 x	7 x	6	EQUALS	336
10	362,880	9 x	8 x	7	EQUALS	504
11	3,628,800	10 x	9 x	8	EQUALS	720
12	39,916,800	11 x	10 x	9	EQUALS	990
13	479,001,600	12 x	11 x	10	EQUALS	1320
14	6,227,020,800	13 x	12 x	11	EQUALS	1716
15	87,178,291,200	14 x	13 x	12	EQUALS	2184
16	1,307,674,368,000	15 x	14 x	13	EQUALS	2730
17	20,922,789,888,000	16 x	15 x	14	EQUALS	3360
18	355,687,428,096,000	17 x	16 x	15	EQUALS	4080
19	6,402,373,705,728,000	18 x	17 x	16	EQUALS	4896
20	121,645,100,408,832,000	19 x	18 x	17	EQUALS	5814
etc…	etc…	etc…				
30	8,841,761,993,739,700,000,000,000,000,000	30 x	29x	28	EQUALS	24,360

Whereas the numbers in the brute force approach get exponentially larger, our numbers actually get comparatively smaller. As the example in the chart above shows; for 6 cities we require 120 groupings, but for 7, we only require 210 and so on, which is less than double. The number of variables does obviously grow, but in smaller and smaller ratios. Here is the pattern from 6 cities. From 6 to 7 cities we 2x our number. From 7 cities to 8 cities we 1.75x our number and from 8 to 9 we 1.6x, etc. Notice how the ratios get smaller and smaller! How awesome is that!

TABLE 15

Cities	Variables	x factors
6	60	2.0000
7	120	1.7500
8	210	1.6000
9	336	1.5000
10	504	1.4286
11	720	1.3750

12	990	1.3333
13	1320	1.3000
14	1716	1.2727
15	2184	1.2500
16	2730	1.2308
17	3360	1.2143
18	4080	1.2000
19	4896	1.1875
		etc…

For the TSP problem we will need to check a number of the satisfied lower numbers as the smallest number may not be the shortest path. A low number is likely to be one of the satisfied lower paths, but recognize from the example provided that the second shortest path was actually the third lowest number.

As soon as one of our low numbered satisfied paths add up to a number lower than higher numbered potential paths (or potential partial paths), we then know that we can eliminate the upper numbers from our search! This can literally save billions of years of searching! For example; As 442 gave us a nice short path that was satisfied, then any number above this need not be investigated any further for our shortest path query. Only seven numbers were smaller than 442 in our example. Twenty-three partial paths were longer that our completely satisfied full path. Thus we can exclude these from the search, which is awesome because the computer can save an enormous amount of time.

Sorting the partial paths from smallest to largest was the key this success! The other essential and novel element of the present invention is that rather than trying to determine the shortest path by analyzing the system via its individual parts, (for which in this example there are 15), we instead combine two or more single elements, which then provide us with a number of impossible options for each selected path. This creates a NOT in logic terms, that we would not have had access to, if dealing with our variables individually. The system is made larger, but this seems like a necessity to create the necessary constrained environment.

50 partial paths (or collaborative variables), each with two variables (connected to three letters) are required if the start and end points are known. 60 partial paths are required if no starting and ending point are known. Our Sudoku example also benefited from this strategy.

SUDOKU

Sudoku is a logic-based, combinatorial number-placement puzzle. The objective is to fill a 9×9 grid with digits so that each column, each row, and each of the nine 3×3 sub grids that compose the grid (also called "boxes", "blocks", or "regions") contains all of the digits from 1 to 9. The puzzle setter provides a partially completed grid, which for a well-posed puzzle has a single solution.

Completed games are always a type of Latin square with an additional constraint on the contents of individual regions. For example, the same single integer may not appear twice in the same row, column, or any of the nine 3×3 sub regions of the 9x9 playing board.

French newspapers featured variations of the puzzles in the 19th century, and the puzzle has appeared since 1979 in puzzle books under the name Number Place. However, the modern Sudoku only started to become mainstream in 1986 by the Japanese puzzle company Nikoli, under the name Sudoku, meaning "single number". It first appeared in a US newspaper and then The Times (London) in 2004, from the efforts of Wayne Gould, who devised a computer program to rapidly produce distinct puzzles.

In computational complexity theory, the Cook–Levin theorem, also known as Cook's theorem, states that the Boolean satisfiability problem is NP-complete. That is, any problem in NP can be reduced in polynomial time by a deterministic Turing machine to the problem of determining whether a Boolean formula is satisfiable. The theorem is named after Stephen Cook and Leonid Levin.

An important consequence of this theorem is that if there exists a deterministic polynomial time algorithm for solving a problem that can be reduced to Boolean satisfiability, then every NP problem can be solved by a deterministic polynomial time algorithm. The question of whether such an algorithm exists to solve a problem that has been reduced to Boolean satisfiability in deterministic polynomial time is thus equivalent to the P versus NP problem, which is widely considered the most important unsolved (until the publication of this paper) problem in theoretical computer science.

We will now provide an example of Sudoku, but this is in no way meant to limit the present methods to this or any single problem. Any NP-Complete problem may be addressed with variations of the present invention as can be seen in the Cook-Levine papers or the proof based on the one given by Garey and Johnson[8]. Every NP problem can be reduced to an instance of a SAT problem by a polynomial-time many-one reduction.

First we will take an example of a standard 9x9 Sudoku puzzle.

TABLE 5

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

Transposing our clique solving process for Sudoku will require that we transform the Sudoku numbers into binary values. Sudoku numbers 1-2-3-4-5-6-7-8-9 will now become 1-2-4-8-16-32-64-128-256. This will better allow for yes/no answers for our collaborative variables and simplify our solving process. The present method will SEEM QUITE COMPLICATED, but the process will be much more efficient then presently available methods of brute force or single variable constraint programming.

Why?

Because rather than having 81 individual variables, that need to be considered, we have created a system wherein two sets of 27 variables (54 variables in total) can be used instead. In a 16x16 grid with 256 variables, the present system will only require two sets of 64 variables (128 in total). As chart 1 below shows, traditional methods grow extremely rapidly, wherein the present method (although the variables DO grow), the variables grow at a much smaller rate! The variables of the present invention in the 100x100 grid have only one-fifth the variables of traditional constraint methods.

CHART 1

Grid Size	Traditional Number of Variables	Variables of the present invention		Sum
9x9	81	27	27	54
16x16	256	64	64	128
25x25	625	125	125	250
36x36	1296	216	216	432
49x49	2401	343	343	686
100x100	10000	1000	1000	2000

As we will see in the upcoming explanation of how to solve TSP we will be using what I call collaborative constraints for Sudoku. If we now focus on A1 we will notice that it will be provided with constraining elements from A2 and A3 (the horizontal constraints) and also constraints from A4 and A7 (the 3x3 constraint). Any numbers (or collaborative sum) in any of these areas will affect the available options for numbers which can be input into A1. If A1 already has a number in its area, this will also limit available options.

It is essential that we keep in mind that A1, A4 and A7 share the same 3x3 block as D1, D2 and D3. This is one of the elements that will allow our logic to work. They share a commonality, but also are provided with differing constraints. This holds true for all the other 3x3 blocks as well. For example, B2, B5 and B8 share many constraints with E4, E5 and E6, but each also have unique ones that make our system function.

Thus, if we create a number of logic statements, we can solve the Sudoku. For example, in the Sudoku example provided in Table 5, **IF** E1 is a three-variable number (which it is) and **IF** I know that F2, B2 and B9 all contain a variable that contains an identical value (which it does), (32 in binary and 6 in Sudoku), **THEN** B4 and E3 must contain that same number. (For visualization purposes **B4Z is equal to E3Y**).

TABLE 6

A1	A2	A3
A4	A5	A6
A7	A8	A9
B1	B2	B3
B4	B5	B6
B7	B8	B9
C1	C2	C3
C4	C5	C6
C7	C8	C9

D1	D2	D3	D4	D5	D6	A3	D8	D9
E1	E2	E3	E4	E5	E6	E7	E8	E9
F1	F2	F3	F4	F5	F6	F7	F8	F9

TABLE 7 (not really necessary)

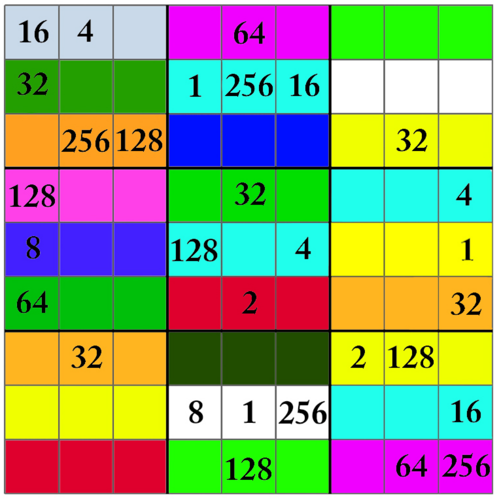
X	Y	Z	X	Y	Z	X	Y	Z
X	Y	Z	X	Y	Z	X	Y	Z
X	Y	Z	X	Y	Z	X	Y	Z
X	Y	Z	X	Y	Z	X	Y	Z
X	Y	Z	X	Y	Z	X	Y	Z
X	Y	Z	X	Y	Z	X	Y	Z
X	Y	Z	X	Y	Z	X	Y	Z
X	Y	Z	X	Y	Z	X	Y	Z
X	Y	Z	X	Y	Z	X	Y	Z

X	X	X	X	X	X	X	X	X
Y	Y	Y	Y	Y	Y	Y	Y	Y
Z	Z	Z	Z	Z	Z	Z	Z	Z
X	X	X	X	X	X	X	X	X
Y	Y	Y	Y	Y	Y	Y	Y	Y
Z	Z	Z	Z	Z	Z	Z	Z	Z
X	X	X	X	X	X	X	X	X
Y	Y	Y	Y	Y	Y	Y	Y	Y
Z	Z	Z	Z	Z	Z	Z	Z	Z

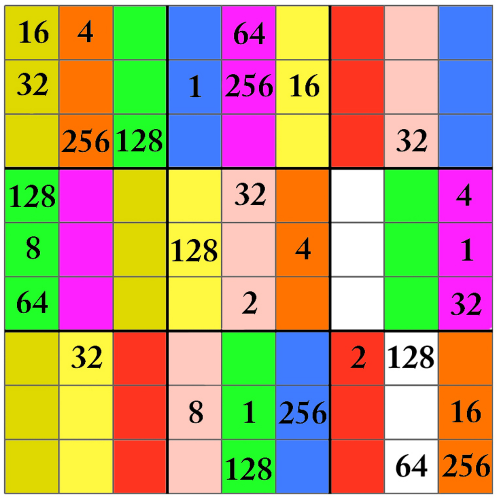
Table 7 shows further breakdown into xyz zones, but realistically we may not require these in the preferred embodiment as for example, if we were to say that A1 and D1 shared a number, the only possible place that they can share that number is in the top space of D1 or furthest left of A1. No other location is possible.

After we name our independent sections we turn the Sudoku numbers 1-2-3-4-5-6-7-8-9 into binary place values 1-2-4-8-16-32-64-128-256. This will make everything much easier for a computational device to comprehend. An essential part and novel aspect of the present solution that has been missing from the art is the ability to assist a computational device with fully understanding what variables do NOT need to be looked at during a searching effort. This will hold true in our next example (TSP) as well. The searching effort seems to be what takes up the computational devices time. The more possibilities that can be eliminated from the searching efforts the faster the output or answers will be found.

TABLE 8



16	4			64				
32			1	256	16			
	256	128					32	
128				32				4
8			128		4			1
64				2				32
	32					2	128	
			8	1	256			16
				128			64	256



16	4			64				
32			1	256	16			
	256	128					32	
128				32				4
8			128		4			1
64				2				32
	32					2	128	
			8	1	256			16
				128			64	256

We will need to list what I will call our number levels for Sudoku. There are three variations. Level 1 numbers are single variables and will include 1, 2, 4, 8, 32, 64, 128, and 256. Once the Sudoku has been solved by our algorithm three of these unique numbers will add up to a larger collaborative variable (Level 3 number) and will reside in one of the 3x1 or 1x3 sections (i.e. A1, A2, D1, D2, etc.). Level 2 numbers are collaborative variables and will be the sum of any two different level one numbers. Again, Level 3 numbers are also collaborative variables and will be the sum of any three different level 1 numbers. This strategy better provides a computer with the ability to answer yes or no questions, as is a requirement in Boolean logic. For example, if A1=7 then we will logically know that A1 consists of a 4, 2 and a 1. By utilizing a binary number system, we allow only the one possibility, no matter what the numbers. This also tells us that the sum of A2 and A3 must be 249.

This is not the case when we use 1-9 numbering. If for example, we were to use 1-9 integer values and A1 equaled 15. This would not really tell us ANYTHING! The 15 in this scenario could be comprised of 456, 951, 942, 861, 852, 834, 762, or 753. By Utilizing binary digits rather than 1-9 integers we can dramatically reduce the time required to find a solution to the Sudoku via faster and better method of constraint programming that have not prior to the present invention been utilized.

LEVEL 1 (one variable numbers) = 1, 2, 4, 8, 16, 32, 64, 128, or 256

LEVEL 2 (two variable numbers) = Any one of --- 3, 5, 6, 9, 10, 12, 17, 18, 20, 24, 33, 34, 36, 40, 48, 65, 66, 68, 72, 80, 96, 129, 130, 132, 136, 144, 160, 192, 256, 257, 258, 260, 264, 272, 288, 320, or 384

LEVEL 3 (three variable numbers) = Any one of --- 7, 11, 13, 14, 19, 21, 22, 25, 26, 28, 35, 37, 38, 41, 42, 44, 49, 50, 52, 56, 67, 69, 70, 73, 74, 76, 81, 82, 84, 88, 97, 98, 100, 104, 112, 131, 133, 134, 137, 138, 140, 145, 146, 148, 152, 161, 162, 164, 168, 176, 193, 194, 196, 200, 208, 224, 259, 261, 262, 265, 266, 268, 273, 274, 276, 280, 289, 290, 292, 296, 304, 321, 322,, 324, 328, 336, 352, 385, 386, 388, 392, 400, 416, or 448

We can now simply run a program with a number of logic steps again and again until the Sudoku is fully solved. It can be modelled as a much simpler and faster constraint satisfaction problem that would likely use many less lines of code.

Here are some examples of logic steps that may be applied **recursively** as numbers get filled in and the constraints increase. This type of program should be effective at solving any Sudoku that would be input into it without the requirement of backtracking or brute force methods.

- **IF** A1 and A2 both = a Level 3 (3 variables added) number **AND IF** A3 = a Level 2 number **THEN** the one empty box (the x, y **or** z) of A3 must = 256 minus the sum of A1, A2 and A3. A3 must then be modified from a Level 2 number to a Level 3 number and the new addition placed in the one empty x, y or z space that is available. The full horizontal line would now in this scenario be complete.
- **IF** E1 contains a Level 3 (3 variables added) number **AND IF** (F2 **OR** D2) AND (B2 **AND** B9) contain the same Level 1 number **THEN** E3 and B4 should share that number *(in the only place they can, which is the z location).*
- We can also be more specific and even use the xyz variables if desired. **IF** (D3 and E1) and (C3 and C8) contain the sum 128 (a Level 1 variable), or the sum 129, 130, 132, 136, 144, 160, 192, or 384 (a Level 2) or 131, 133, 134, 137, 138, 140, 145, 146, 148, 152, 161, 162, 164, 168, 176, 193, 194, 196, 200, 208, 224, 385, 386, 388, 392, 400, 416 or 448 (a Level 3), **THEN** C4(Y) is 128 (8 in Sudoku).

- We can also make generalized family systems. For example, **IF** B3 and B5 contain a same number (such as binary 4) or contain a higher level number that includes binary 4, (by binary necessity to acquire its sum such as 5 (4,1) or 13 (1, 8, 4), **AND** (E1 and E3) contain Level 3 numbers **THEN** E2(Z) should be filled with that number. The (example 4) number is then converted to the Sudoku number 3. *(Note: the pictured examples, do not provide this example).*
- MANY more logic operators can be created and will guarantee a solution to even the most complicated Sudoku. Brute force methods will no longer be required to solve these types of problems, but they may be used in collaboration. We can design a complete system utilizing only a portion of all possible logic operators. The simpler the Sudoku, the less operators that will likely be required.

The key to the Sudoku solution seems to be in breaking the puzzle down into a variety of smaller units, each of which share two unique logic systems. By logic systems I am referring to our 3x1 units which each belong to both a 9x1 horizontal and a 3x3 group **and** our 1x3 units which belong to 1x9 vertical and 3x3 groups. When logic operators make these two groups talk to each other, every blank in a satisfiable Sudoku can easily be found. Each Sudoku number is assigned a binary digit (1, 2, 4, 8, 16, 32, 64, 128 or 256). We can apply the logic operators recursively (again and again as more blank spaces are filled with numbers) to fill in the puzzle. Once it is complete and everything is satisfied logically, we can convert our numbers back to 1-9 and our puzzle is solved in polynomial time.

This method can also be utilized for larger puzzles, we simple need to increase our numbers. A 16x16 Sudoku would require 1x4 and 4x4 logic groupings and our binary numbers would go to 65536.

Note:
A similar approach can be used to solve the "Eight Queens Puzzle". If the chess board has an even length (i.e. 8x8) it is easy to visualize, but if it is an odd length (i.e. 9x9), we need to adjust the strategy wherein we have two 5x5's with some crossover between each sub-group…but that's for another paper.

SOME OTHER NP-COMPLETE PROBLEMS

Karp's 21 problems are shown below, many with their original names. The nesting indicates the direction of the reductions used. For example, Knapsack was shown to be NP-complete by reducing

Exact cover to Knapsack. The methods of the present invention can be modified and transposed to solve all of these problems (and many more) in a much shorter time than is presently available.

Satisfiability: The Boolean satisfiability problem for formulas in conjunctive normal form (often referred to as SAT)
0–1 integer programming (A variation in which only the restrictions must be satisfied, with no optimization)
Clique (see also independent set problem)
Set packing
Vertex cover
Set covering
Feedback node set
Feedback arc set
Directed Hamilton circuit (Karp's name, now usually called Directed Hamiltonian cycle)
Undirected Hamilton circuit (Karp's name, now usually called Undirected Hamiltonian cycle)
Satisfiability with at most 3 literals per clause (equivalent to 3-SAT)
Chromatic number (also called the Graph Coloring Problem)
Clique cover
Exact cover
Hitting set
Steiner tree
3-dimensional matching
Knapsack (Karp's definition of Knapsack is closer to Subset sum)
Job sequencing
Partition
Factoring
Max cut
…and many more
https://en.wikipedia.org/wiki/List_of_NP-complete_problems

About the Author

I (John Archie Gillis) am from Halifax, NS, Canada. This is my solution to the P versus NP problem. I have no computational complexity theory background, but I did find the systems and problems in this field somewhat similar to Jazz Music Theory, for which I obtained an advanced degree in guitar performance in 1999. The similarities of chords and scales to groups and sub-groups seems quite related and if my paper is found to be correct, this is likely the reason as to why I was able to come up with a solution.

If you have any questions or comments please don't hesitate to contact me at johnarchiegillis@gmail.com

F - References

1. Cook, S.A. (1971). "The complexity of theorem proving procedures". Proceedings, Third Annual ACM Symposium on the Theory of Computing, ACM, New York. pp. 151–158. doi:10.1145/800157.805047.
2. Wikipedia contributors. (2018, June 19). Clique problem. In *Wikipedia, The Free Encyclopedia*. Retrieved 13:57, June 28, 2018, from https://en.wikipedia.org/w/index.php?title=Clique_problem&oldid=846513850
3. Wikipedia contributors. (2018, May 14). NP-completeness. In *Wikipedia, The Free Encyclopedia*. Retrieved 14:00, June 28, 2018, from https://en.wikipedia.org/w/index.php?title=NP-completeness&oldid=841292328
4. Wikipedia contributors. (2018, June 20). Travelling salesman problem. In *Wikipedia, The Free Encyclopedia*. Retrieved 14:01, June 28, 2018, from https://en.wikipedia.org/w/index.php?title=Travelling_salesman_problem&oldid=846797001
5. Wikipedia contributors. (2018, June 22). Sudoku. In *Wikipedia, The Free Encyclopedia*. Retrieved 14:02, June 28, 2018, from https://en.wikipedia.org/w/index.php?title=Sudoku&oldid=847081582
6. The Traveling Saleswitch Problem. http://people.ku.edu/~jlmartin/courses/math105-F11/Lectures/chapter6-part3.pdf
7. Gödel's Lost Letter and P=NP https://rjlipton.wordpress.com/the-gdel-letter/

8. Garey, Michael R.; David S. Johnson (1979). Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman. *ISBN* *0-7167-1045-5*