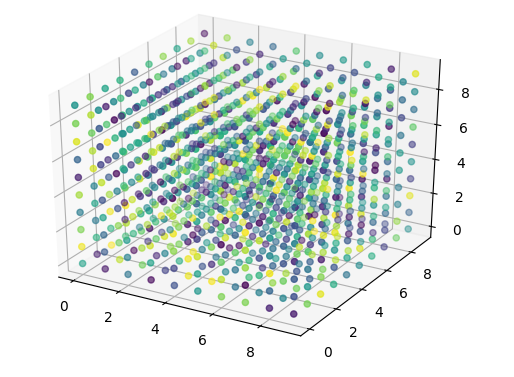
**P = NP and some other relevant things.**

**Welcome to the Matrix**



**Authored by:**

**Zackary Seger**

Undergraduate Student at

St. Louis Christian College

[zackary@seger.us](mailto:zackary@seger.us)

224 Saint Benedict,

Pevely, MO 63070

**Abstract**

The purpose of this paper is to prove that P = NP. By showing the following solution alone, and taking into consideration the statement,

P = { L | L = L(M) for some Turing machine M that runs in polynomial time} [1].

All that is meant by that statement is taken into consideration in this paper. From the current state of what I have been able to find on the internet, you would think that saying P = NP is a sure way to make yourself into a fool. Either a fool, or the smartest man alive. I hope to not be seen as either, but here you go.

Items: A = { x ∈ A | x = an }

Whitelist: B = B = { y ∈ B | y = ( x1 , x2 ) }

Random List: C = C = { z ∈ C | z = ( zn , zlen(B) - z ) }

Sorted List: D = D ⊆ C = { | any z ≠ any z and any ∈ zn ≠ any other ∈ zn }

P = NP because the above algorithm is a polynomial time solution for NP problems. Its time complexity is O(log(n)).

Most is this paper is dedicated to explaining the solutions proving *N = NP*, and the rest is meant to explain programming solution examples and application, and then the scope of the application of the math, given the current state of computer science. It is curious that so many believed that these problems were simply unsolvable, that it does not seem like anyone even tried to solve them. I only say that because as shown above, the proof that *P = NP* only requires *four* lines. In any case, it was believed that if you could solve a problem in the highest hierarchal level of NP, then you would be able to solve all problems beneath it.

This is obviously true, however, what was not so obvious, was that the solution to the simplest NP problems, is what actually led the path to the very complex ones. In terms of math, it is simply the difference between having an extra loop and not having not having one. There is a set of rules that governs every problem. Certain sets of rules govern many problems, but all problems can be written out in standard algorithms. The NP solution works for a large variety of questions, but does not take into consideration anything other than not including something.

The *NP-Complete* solution has to understand that you need to do something, and based on the rules it has to figure out what the required next move is. If you think about sudoku, wherever you add a new piece to the puzzle, you may create certain requirements, and then you might score an easy piece because of that. You might also run out of requirements in which case you guess randomly and then delete and replace the guesses that were wrong. Don’t make the same guesses twice.

**NP Solution**

Items: A = { x ∈ A | x = an }

Whitelist: B ⊆ A = { y ∈ A | y = ( xm , xn ) }

Random List: C ⊆ A = { z ∈ A | z = ( xn , xlen(B) - n ) }

Sorted List: D = D ⊆ C = { d ∈ C | any z ≠ any z and any ∈ zn ≠ any other ∈ zn }

**NP-Complete Solution**

Items & Locations: A = { a ∈ A | a = {( x1 , l1), ( xn , ln), …}}

New I & Ls: Bn ⊆ A = { bn ∈ A | bn = the new solution given the ruleset of the problem given A }

Next I & Ls Bn+1 ⊆ Bn = {bn+1 ∈ A | bn = the next required solution given the:rulesetof the problem given bn }

.

Final I & L Sorted: C ⊆ Bn = { c ∈ Bn | c = the final solution c given the ruleset of the problem given bn }

**Therefore P = NP = NP-Complete = NP-Hard**

The statement, P = { L | L = L(M) for some Turing machine *M* that runs in polynomial time} [2],

is **true**for all computationally possible problems. The following pages contain proof of a solution to the *‘Halting Problem.’* As it turns out, to solve this crazy conundrum, we don’t just guess from all the solutions, we just look for the ones which are possible.

**NP Solution**

Items: A = { x ∈ A | x = an }

Whitelist: B ⊆ A = { y ∈ A | y = ( xm , xn ) }

Random List: C ⊆ A = { z ∈ A | z = ( xn , xlen(B) - n ) }

Sorted List: D = D ⊆ C = { d ∈ C | any z ≠ any z and any ∈ zn ≠ any other ∈ zn }

**Explanation**

In this explanation I am going to talk about the math and how it can be written programmatically. To do so, we need to solve a problem, and we will solve the problem proposed by the Clay Mathematics Institute,

*‘Suppose that you are organizing housing accommodations for a group of four hundred university students. Space is limited and only one hundred of the students will receive places in the dormitory. To complicate matters, the Dean has provided you with a list of pairs of incompatible students, and requested that no pair from this list appear in your final choice. This is an example of what*

*computer scientists call an NP-problem, since it is easy to check if a given choice of one hundred students proposed by a coworker is satisfactory (i.e., no pair taken from your coworker's list also appears on the list from the Dean's office), however the task of generating such a list from scratch seems to be so hard as to be completely impractical.’* [3]

*The first statement | Items: A = { x ∈ A | x = an }*

explains that the list A is equal to a set of values where x is an element of the set A given that x is equal to some variable an. To solve our student housing crisis, the elements of set A are equal to all of the names of our students to be housed.

*The second statement | Whitelist: B = { y ∈ B | y = ( xm , xn ) }*

is the set B where y is an element of B such that y is equal to some pair-set of x values. In our housing problem, we can see this as our set of pairs of students which cannot be housed together Those students who cannot be housed together must be on the housing list, and one student cannot room with him or herself.

*The third statement | Random List: C = { z ∈ C | z = ( xn , xlen(B) - n ) }*

states that our set C contains elements z such that elements z are equal to set-pairs of xn, and xlen(B) - n. To solve the housing crisis, all we need to do is generate a random set of students, technically based on the order of the student list, where one student is paired with his or her corresponding opposite within the list. The first person in the list is paired with the last, and the second with the second last, and so on. When we do this, if the number of students is 200, we will create 100 pairings of students, which is obviously too many, but hey the more the merrier.

*The final statement: Sorted List: D = { n ∈ C| any n ≠ any y and any ∈ of any n = any other ∈ any n }*

Shows that the list D contains elements of the list C, given that those elements do not equal each other, and the elements within those elements do not equal each other either. This is the solution step to our problem. We have a list of all of our students paired together semi-randomly. From that list, we take all the take all the student-pairs which are not contained in the whitelist, and don’t include duplicates of any names. This will leave us with 100 unique pairs every time.

The reason this solution solves P = NP is because it can be used in the exact same form to solve the following question:

Given a graph with 400 vertices, can you pick 100 vertices such that no two of them are adjacent?

The solution does not change. A simple python program proving this solution works can be found below and at [zackary.seger.us](file:///C:\Users\zak\AppData\Roaming\Microsoft\Word\zackary.seger.us) along with the corresponding text file.

Program

|  |
| --- |
| ## Zackary Seger |
| ## Begin Date & Time: 2/20/2020 at about 9:00PM |
| ## Completion Date & Time: 2/21/2020 at around 1:00AM |
| ## First Draft was written in word and saved that way, lol. |
|  |
|  |
| import time ## For this program we require both |
| import random ## the time and random libraries. |
|  |
| start\_time = time.time() ## Creates a base time |
|  |
| file = open("names.txt", "r") ## This is a list of 400 names |
| lines = file.readlines() |
|  |
| ## This is how we add the names to the first list, All Students. We just |
| ## navigate the text file line by line, and add each word to All Students |
| ## list. |
| ## Time Complexity = O(n) |
|  |
| x = 0 |
| All\_Students = [] |
| for line in lines: |
| x += 1 |
| All\_Students.append(line) |
|  |
| ## The Whitelist is all the pairs that aren't allowed. |
| ## This is a list of 20 not allowed pairs, or 40% of the |
| ## total maximum pairs. With thes sizes the program |
| ## completes itself in about 1 minute total. As you add |
| ## more exceptions to this program the literal time it takes |
| ## to complete will grow quickly. With only 10 Whitelist |
| ## entries the program will complete itself in about 15 seconds. |
|  |
| Whitelist = [[ "WILLIAMS\n", "MOODY"],[ "BROWN\n", "COBB\n" ], |
| [ "MILLER\n", "FLOWERS\n"], ["JONES\n", "MOODY"], |
| ['CHAVEZ\n', 'MONTGOMERY\n'], ['BAKER\n', 'TODD\n'], |
| ['GONZALEZ\n', 'NEWTON\n'], ['NELSON\n', 'ROBBINS\n'], |
| ['CARTER\n', 'RODGERS\n'], ['MITCHELL\n', 'HARMON\n'], |
| ['PEREZ\n', 'COHEN\n'], ['ROBERTS\n', 'MANNING\n'], |
| ['TURNER\n', 'GLOVER\n'], ['PHILLIPS\n', 'VEGA\n'], |
| ['CAMPBELL\n', 'AGUILAR\n'], ['PARKER\n', 'DELGADO\n'], |
| ['EVANS\n', 'FARMER\n'], ['EDWARDS\n', 'MCGEE\n'], |
| ['COLLINS\n', 'DENNIS\n'], ['PALMER\n', 'SILVA\n'], |
| ['MILLS\n', 'CARLSON\n'], ['NICHOLS\n', 'HOFFMAN\n'], |
| ['GRANT\n', 'BREWER\n'], ['KNIGHT\n', 'FOWLER\n'], |
| ['FERGUSON\n', 'MEDINA\n'], ['ROSE\n', 'BOWMAN\n'], |
| ['STONE\n', 'MORENO\n'], ['HAWKINS\n', 'MENDOZA\n'], |
| ['DUNN\n', 'DAY\n'], ['PERKINS\n', 'HANSON\n'], |
| ['HUDSON\n', 'BURKE\n'], ['SPENCER\n', 'FRAZIER\n'], |
| ['GARDNER\n', 'LARSON\n'], ['STEPHENS\n', 'WELCH\n'], |
| ['PAYNE\n', 'ROMERO\n'], ['PIERCE\n', 'GARRETT\n'], |
| ['BERRY\n', 'GILBERT\n'], ['MATTHEWS\n', 'DEAN\n'], |
| ['ARNOLD\n', 'LYNCH\n'], ['MATTHEWS\n', 'LYNCH\n']] |
|  |
| Housing\_List = [] |
| del\_list = [] |
| new\_list = [] |
|  |
| ## The following is a for loop that iterates through each student in |
| ## the list we created, and it inserts all of our data into pairs by |
| ## traversing the list from outside in. This is a random distribution |
| ## of our data, which is also in a random order originally, and allows |
| ## us to create many initial sets that will end up qualifying as |
| ## allowed sets. For the purpose of solving the problem, it does |
| ## not matter whether or not this data is sorted. |
| ## Time Complexity: O(n) |
|  |
| x = 0 |
| for student in All\_Students: |
|  |
| lastS = int(len(All\_Students) - 1) |
|  |
| if student == All\_Students[0]: |
| Housing\_List.append([student, All\_Students[lastS]]) |
|  |
| elif x > int(lastS / 2) : |
| Housing\_List.append([student, 0]) |
| break |
|  |
| else: |
| Housing\_List.append([student, All\_Students[lastS - x]]) |
| x += 1 |
|  |
| print("first loop complete \n\n") |
| print(Housing\_List) |
|  |
| ## In this next loop we will be comparing each sublist in our Housing List, |
| ## with each sublist in our Whitelist. We say for every sublist in in our |
| ## randomly generated Housing List, check every sublist in our Whitelist |
| ## to see whether or not it should be there. If the match is not allowed, |
| ## then delete it from the Housing List, and add it to another list called |
| ## |
|  |
|  |
| n = 0 |
| for set1 in Housing\_List: |
| for set2 in Whitelist: |
| Backset = [[set2[0], set2[1]]] |
| if set1 == set2 or set1 == Backset: |
| del\_list.append(set1) |
| Housing\_List.remove(set1) |
|  |
| n += 1 |
|  |
|  |
| print("second loop complete \n") |
|  |
| y = 0 |
| for set1 in del\_list: |
| new\_list.append(set1[0]) |
| new\_list.append(set1[1]) |
|  |
| del del\_list[y] |
| y += 1 |
| del\_list = [] |
|  |
|  |
| print("third loop complete \n\nHousing List: ", Housing\_List, |
| "\n\nWhitelist: ", Whitelist,"\n\nNew List: ", new\_list) |
|  |
| time\_now = time.time() |
| q = 0 |
| while q < 500: |
|  |
| new\_list = list(dict.fromkeys(new\_list)) |
|  |
|  |
| if len(Housing\_List) < 200: |
| print(q) |
| print("Housing List Length:", len(Housing\_List)) |
| print("New List Length:", len(new\_list)) |
| random.shuffle(new\_list) |
|  |
| q += 1 |
| n = 0 |
| for set1 in new\_list: |
| for set2 in Whitelist: |
| Backset = [[set2[0], set2[1]]] |
| if set1 == set2 or set1 == Backset: |
| del\_list.append(set1) |
| new\_list.remove(set1) |
| n += 1 |
|  |
| for set1 in Housing\_List: |
| for set2 in Whitelist: |
| Backset = [[set2[0], set2[1]]] |
| if set1 == set2 or set1 == Backset: |
| del\_list.append(set1) |
| Housing\_List.remove(set1) |
| n += 1 |
|  |
|  |
| y = 0 |
| for set1 in del\_list: |
| new\_list.append(set1[1]) |
| new\_list.append(set1[0]) |
|  |
| del del\_list[y] |
| y += 1 |
| del\_list = [] |
|  |
| x = 0 |
| for student in new\_list: |
|  |
| lastS = int(len(new\_list) - 1) |
|  |
| if student == new\_list[0]: |
| Housing\_List.append([student, new\_list[lastS]]) |
|  |
| elif x > int(lastS / 2) : |
| Housing\_List.append([student, 0]) |
| break |
|  |
| else: |
| Housing\_List.append([student, new\_list[lastS - x]]) |
| x += 1 |
|  |
| for set1 in Housing\_List: |
| for set2 in Whitelist: |
| Backset = [[set2[0], set2[1]]] |
| if set1 == set2 or set1 == Backset: |
| del\_list.append(set1) |
| Housing\_List.remove(set1) |
|  |
| for setN in Housing\_List: |
| if setN[0] == setN[1]: |
| Housing\_List.remove(setN) |
|  |
| for setN in Housing\_List: |
| if setN[1] == 0: |
| Housing\_List.remove(setN) |
|  |
| for student in All\_Students: |
| s = 0 |
| for setN in Housing\_List: |
| if student == setN[0] or student == setN[1]: |
| s += 1 |
| if s == 2: |
| Housing\_List.remove(setN) |
|  |
| ## |
|  |
| for set1 in Housing\_List: |
| for set2 in Whitelist: |
| Backset = [[set2[0], set2[1]]] |
| if set1 == set2 or set1 == Backset: |
| del\_list.append(set1) |
| Housing\_List.remove(set1) |
|  |
| for setN in Housing\_List: |
| if setN[0] == setN[1]: |
| Housing\_List.remove(setN) |
|  |
| for setN in Housing\_List: |
| if setN[1] == 0: |
| Housing\_List.remove(setN) |
|  |
| for student in All\_Students: |
| s = 0 |
| for setN in Housing\_List: |
| if student == setN[0] or student == setN[1]: |
| s += 1 |
| if s == 2: |
| Housing\_List.remove(setN) |
|  |
| print("Complete \n\nHousing List: ", Housing\_List, |
| "\n\nWhitelist: ", Whitelist) |
|  |
| print(len(Housing\_List)) |
| print("Seconds to completion: ", time.time() - start\_time) |

**NP-Complete Solution**

Items & Locations: A = { a ∈ A | a = {( x1 , l1), ( xn , ln), …}}

New I & Ls: Bn ⊆ A = { bn ∈ A | bn = the new solution given the ruleset of the problem given A }

Next I & Ls Bn+1 ⊆ Bn = {bn+1 ∈ A | bn = the next required solution given the:rulesetof the problem given bn }

.

Final I & L Sorted: C ⊆ Bn = { c ∈ Bn | c = the final solution c given the ruleset of the problem given bn }

**Explanation**

The solution here shows that set A is a set of elements a such that the elements are sets of some data xn and some location/ weight. The solution is formed by creating a logical ruleset that governs an amount of iterations required to solve the problem. Each set is a solution set that has been given appended value-sets based on the previous verified value sets. All problems begin with a certain set of initial values. Based on those values, you have to find out what values are required, and take your best guess based on the ruleset at what the next values will be. They will be within some range every time. If the values you have do not create any requirements, then you have to guess at elements that are allowed. When you do this, you find that you can be right very quickly simply by taking guesses and eliminating the incorrect guesses. You will eventually create requirements and things will be even easier. Or you will break the program, need to start from scratch, and avoid the path that leads to nowhere. This program takes a lot more effort to write, but does work. Just put it in terms of sudoku. There is a n x n board where the board is split into n cubes, n rows, and n columns. Each row, column, and cube cannot contain a value greater than n, cannot contain a 0 or negative, and cannot contain the same value more than once. Knowing those rules, you can check logically if you there is a requirement at each location. If the numbers in the row, and the numbers in the column, and the numbers in cube are only missing one number within n, that number is required.

The code for the grids required is written out for Python 3.7 below. Grids created from dictionaries and nested lists are clunky, and hard to navigate. The grid below works at n x n size, and works by the same principles that govern linked lists.

Program

|  |
| --- |
| class Node: |
| def \_\_init\_\_(self, data): |
| self.data = data |
| self.east = None |
| self.west = None |
| self.south = None |
| self.north = None |
| self.southeast = None |
| self.southwest = None |
| self.northeast = None |
| self.northwest = None |
|  |
| class LinkedGrid: |
| def \_\_init\_\_(self): |
| self.head = None |
|  |
| def link\_from\_top(self, data): |
| cur = self.head |
| cur2 = data.head |
| cur3 = data.head.east |
|  |
| n = 0 |
| a = 0 |
| z = 0 |
|  |
| cur.south = cur2 |
|  |
| while cur.east: |
| a += 1 |
| n = 0 |
| cur = cur.east |
| cur2 = data.head |
|  |
| while cur2.east: |
| cur2 = cur2.east |
| south = cur2 |
| n += 1 |
| if n is a: |
| cur.south = south |
|  |
| def link\_se\_from\_top(self, data): |
| cur = self.head |
| cur2 = data.head |
|  |
| n = 0 |
| a = 0 |
| z = 0 |
|  |
| cur.southeast = cur2.east |
|  |
| while cur.east: |
| a += 1 |
| n = 0 |
| cur = cur.east |
| cur2 = data.head |
|  |
| while cur2.east: |
| cur2 = cur2.east |
| southeast = cur2.east |
| n += 1 |
| if n is a: |
| cur.southeast = southeast |
|  |
| def link\_sw\_from\_top(self, data): |
| cur = self.head |
| cur2 = data.head |
|  |
| n = 0 |
| a = 0 |
| z = 0 |
|  |
| cur.southwest = cur2.west |
|  |
| while cur.east: |
| a += 1 |
| n = 0 |
| cur = cur.east |
| cur2 = data.head |
|  |
| while cur2.east: |
| cur2 = cur2.east |
| southwest = cur2.west |
| n += 1 |
| if n is a: |
| cur.southwest = southwest |
|  |
| def link\_ne\_from\_top(self, data): |
| cur = self.head |
| cur2 = data.head |
|  |
| n = 0 |
| a = 0 |
| z = 0 |
|  |
| cur.northeast = cur2.east |
|  |
| while cur.east: |
| a += 1 |
| n = 0 |
| cur = cur.east |
| cur2 = data.head |
|  |
| while cur2.east: |
| cur2 = cur2.east |
| northeast = cur2.east |
| n += 1 |
| if n is a: |
| cur.northeast = northeast |
|  |
| def link\_nw\_from\_top(self, data): |
| cur = self.head |
| cur2 = data.head |
|  |
| n = 0 |
| a = 0 |
| z = 0 |
|  |
| cur.northwest = cur2.west |
|  |
| while cur.east: |
| a += 1 |
| n = 0 |
| cur = cur.east |
| cur2 = data.head |
|  |
| while cur2.east: |
| cur2 = cur2.east |
| northwest = cur2.west |
| n += 1 |
| if n is a: |
| cur.northwest = northwest |
|  |
| def append(self, data): |
| if self.head is None: |
| new\_node = Node(data) |
| new\_node.west = None |
| new\_node.south = None |
| new\_node.north = None |
| self.head = new\_node |
| else: |
| new\_node = Node(data) |
| cur = self.head |
| while cur.east: |
| cur = cur.east |
| cur.east = new\_node |
| new\_node.west = cur |
| new\_node.east = None |
| new\_node.south = None |
| new\_node.north = None |
|  |
| def prepend(self, data): |
| if self.head is None: |
| new\_node = Node(data) |
| new\_node.west = None |
| self.head = new\_node |
| else: |
| new\_node = Node(data) |
| self.head.west = new\_node |
| new\_node.east = self.head |
| self.head = new\_node |
| new\_node.west = None |
|  |
| def add\_after\_node(self, key, data): |
| cur = self.head |
| while cur: |
| if cur.east is None and cur.data == key: |
| self.append(data) |
| elif cur.data == key: |
| new\_node = Node(data) |
| nxt = cur.east |
| cur.east = new\_node |
| new\_node.east = nxt |
| new\_node.west = cur |
| nxt.west = new\_node |
| cur = cur.east |
|  |
| def add\_before\_node(self, key, data): |
| cur = self.head |
| while cur: |
| if cur.west is None and cur.data == key: |
| self.prepend(data) |
| elif cur.data == key: |
| new\_node = Node(data) |
| prev = cur.west |
| prev.east = new\_node |
| cur.west = new\_node |
| new\_node.east = cur |
| new\_node.west = prev |
| cur = cur.east |
|  |
| def print\_list(self): |
| cur = self.head |
| while cur: |
| print('node data: ', cur.data) |
| print('node location: ', cur) |
| cur = cur.east |
|  |
| def print\_list\_south(self): |
| cur = self.head |
| while cur: |
| print('node data: ', cur.data) |
| print('node southern link: ',cur.south) |
| cur = cur.east |
|  |
| def print\_list\_southeast(self): |
| cur = self.head |
| while cur: |
| print('node data: ', cur.data) |
| print('node southeastern link: ',cur.southeast) |
| cur = cur.east |
|  |
| def print\_list\_southwest(self): |
| cur = self.head |
| while cur: |
| print('node data: ', cur.data) |
| print('node southwestern link: ',cur.southwest) |
| cur = cur.east |
|  |
| def print\_list\_northeast(self): |
| cur = self.head |
| while cur: |
| print('node data: ', cur.data) |
| print('node northeastern link: ',cur.northeast) |
| cur = cur.east |
|  |
| def print\_list\_northwest(self): |
| cur = self.head |
| while cur: |
| print('node data: ', cur.data) |
| print('node northwestern link: ',cur.northwest) |
| cur = cur.east |
|  |
| dllist = LinkedGrid() |
|  |
| dllist.append(1) |
| dllist.append(2) |
| dllist.append(3) |
| dllist.append(4) |
| dllist.append(5) |
|  |
| dllist2 = LinkedGrid() |
|  |
| dllist2.append(6) |
| dllist2.append(7) |
| dllist2.append(8) |
| dllist2.append(9) |
| dllist2.append(10) |
|  |
| dllist3 = LinkedGrid() |
|  |
| dllist3.append(11) |
| dllist3.append(12) |
| dllist3.append(13) |
| dllist3.append(14) |
| dllist3.append(15) |
|  |
| dllist4 = LinkedGrid() |
|  |
| dllist4.append(16) |
| dllist4.append(17) |
| dllist4.append(18) |
| dllist4.append(19) |
| dllist4.append(20) |
|  |
| dllist5 = LinkedGrid() |
|  |
| dllist5.append(21) |
| dllist5.append(22) |
| dllist5.append(23) |
| dllist5.append(24) |
| dllist5.append(25) |
|  |
|  |
| dllist.link\_from\_top(dllist2) |
| dllist2.link\_from\_top(dllist3) |
| dllist3.link\_from\_top(dllist4) |
| dllist4.link\_from\_top(dllist5) |
|  |
| dllist.link\_se\_from\_top(dllist2) |
| dllist.link\_sw\_from\_top(dllist2) |
| dllist2.link\_ne\_from\_top(dllist) |
| dllist2.link\_nw\_from\_top(dllist) |
|  |
| dllist2.link\_se\_from\_top(dllist3) |
| dllist2.link\_sw\_from\_top(dllist3) |
| dllist3.link\_ne\_from\_top(dllist2) |
| dllist3.link\_nw\_from\_top(dllist2) |
|  |
| dllist3.link\_se\_from\_top(dllist4) |
| dllist3.link\_sw\_from\_top(dllist4) |
| dllist4.link\_ne\_from\_top(dllist3) |
| dllist4.link\_nw\_from\_top(dllist3) |
|  |
| dllist4.link\_se\_from\_top(dllist5) |
| dllist4.link\_sw\_from\_top(dllist5) |
| dllist5.link\_ne\_from\_top(dllist4) |
| dllist5.link\_nw\_from\_top(dllist4) |
|  |
| dllist2.print\_list() |
| print('\n') |
| dllist.print\_list\_south() |
| print('\n') |
| dllist.print\_list\_southeast() |
| print('\n') |
| dllist.print\_list\_southwest() |
| print('\n') |
| dllist2.print\_list\_northeast() |
| print('\n') |
| dllist2.print\_list\_northwest() |
| print('\n') |
| dllist.print\_list() |
|  |

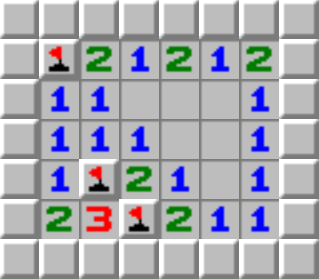
Once your grid is built out, you can decide how you want to traverse it by looking at the problem you wish to solve. For example, in the game Minesweeper, the player must reveal tiles which he/she believes are not covering bombs, and mark away the tiles that are. Each time a tile is revealed, beneath it is either a number, a blank, or a bomb. If a bomb is revealed, the game is over. If a blank is revealed, all adjacent blanks and each new adjacent blank is revealed, until there is a perimeter of numbers is revealed. When a number is revealed two pieces of information are revealed, one: there is at least one bomb adjacent to that tile; and two: the number is the of bombs adjacent to that tile. The problem with writing a program to *‘solve’* Minesweeper is that you cannot *‘solve’* Minesweeper. Each turn does not give a required turn. In the opposite corner, every Sudoku puzzle can be solved without guessing. This is where the difference in the NP-Completeness lies. You can see that in Minesweeper, given the first turn, none of the board is revealed to you. With that in mind, the probability of a first turn loss is equal to the sum total of bombs over n x n.

Then, nothing is guaranteed for your second turn. There a large number of special cases where you may have marked bombs and can rule out certain spaces as either definitely a bomb or definitely not a bomb. Especially important though, is the case of the outlying 1. In figure 1 below, we see two



Figure

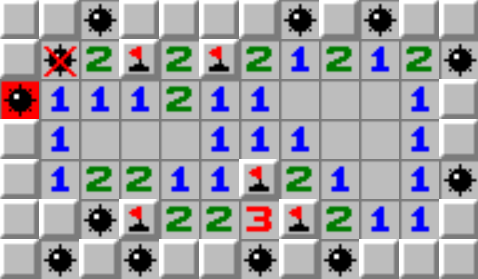
flags which are adjacent to a 1, which only touches one unknown. We can say we know that there are bombs under the flags because we know that 4 of the adjacent tiles are revealed and contain numbers, and three of them are revealed and blank. In Figure 2, we see that



Figure

since we knew that certain tiles had to be bombs, we could also determine that certain other tiles could not possibly be bombs. The tile to the left of the tile which is above the upper flag in figure one cannot be a bomb because there is already another bomb that is adjacent to it. The tile to the left and above of the tile which is above the upper flag in figure one also cannot be a bomb because there is already another bomb that is adjacent to it. Then we can look at the tiles to the left of the flags. The tile to the left of the top flag cannot be a bomb because the 1 located above it is already touching a bomb. That tile reveals another 1, so the two tiles to the left of the bottom flag, which are adjacent to the 1 we just spoke of, also cannot possibly contain bombs. We can also then see another outlying 1 , and can mark that tile as a bomb as well.

We can keep doing this, but at some point, we will be forced to take our best guess. There is also another issue inherent with playing this game today. If you play and pay attention, you will work on getting yourself to the point of guessing, and then something like this will happen,



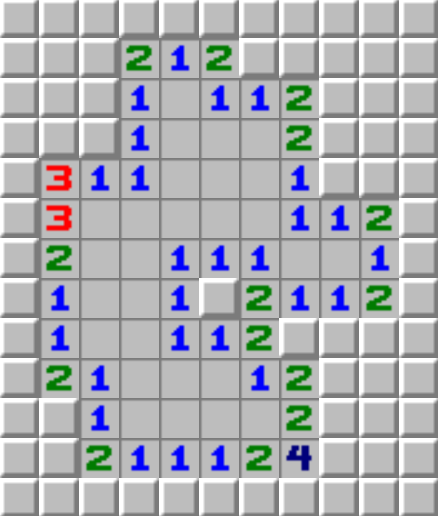
Figure

and all you will be able to do is get frustrated that the top google links will give you nothing except crummy copies of the original game that don’t truly work. In figure 3 you can see that the red tile with the bomb is the bomb which I clicked on, and the x on the bomb where I had a flag. You can clearly see a one next to two bombs. Anyhow you cannot trust these games for any sort of test, but sometimes you can get them to a point that is a pure guess to show what it looks like. Figure 4 below, shows what one of



Figure

situations. In the above scenario, we can see that there are no requirements which can possibly be filled. Each number has a variety of unknown adjacent tiles, and there are no known bombs to work off of. In cases like this one, the only available option is to take the best probable guess. To do that, we need to consider our options. In the case of Minesweeper, given standard rules, the highest number of adjacent bombs is 5. For each number tile *n*, the probability that an adjacent tile *T* is a bomb is *n* over the sum of adjacent unknown tiles *u*.



Figure

You can complete this computation for each number tile, and determine which tile has the lowest probability of being a bomb, and use that as you best choice. Given that you are making your best choice, you have should have a probability like 1/5, or 20%, or lower. The case of the outlying 1 is the case where a 1 is adjacent to the minimum number of tiles. The opposite case, where a one 1 is located on an inside corner, gives you that probability of 1/5. That leaves you with an 80% chance of choosing a space which is not a bomb, and guarantees you at least one new tile to work with. The following game board in Figure 5 shows a lot of special case examples which work great to clean up this explanation.

We can see two 3’s of the leftmost side of the board. The lower three is the first tile we will focus on. The first thing to notice is that this 3 is only adjacent to 3 unknowns. This is a perfect situation for us, because we can confirm that those 3 adjacent unknowns must be bombs. We can even confirm this with our probability formula for a bomb at *T.* Looking at the board, we said there are 3 adjacent unknown tiles, *u,* for the lower three tile*.* We also know that our *n* is equal to 3. Plugging the numbers into our formula, using any tile *T* adjacent to the tile *u,* we end up with this:

or a 100% probability that there is a bomb located at each adjacent tile T.

Now the 4 in the bottom right corner of the board is an example of a worst-case scenario. In advanced levels of the game, you may experience high numbers up to 7, but for the case of this explanation our 4 will show worst case for a high number you are more likely to see. Here we see a tile 4, *n*, which has 5 adjacent unknown tiles *u.* The probability that any adjacent tile *T* is a bomb is then:

or there is 75% that T is a bomb. You can see that what we are really doing in our program is that at each step, we are simply making choices based on probability. If there is a 100% chance that a tile is a bomb, we mark the tile as a bomb, and it affects all the following probabilities of its adjacent tiles.

The ideal program for solving problems like these contains a grid like the previous code in this section, as represented in Figure 6 In that code we create a grid which can be represented as n x n in size, and where each node within the grid is linked to a corresponding north, south, east, and west node, respectively. The grid is created by building linked lists out of node objects, and then linking those linked lists one to the next. No linking or search algorithm for a grid should ever have a time complexity higher than O(n2). In fact, at each node, you have access to up to 8 adjacent nodes. The grid structure is modeled in figure 6. The figure shows connections to and from each adjacent node to the next. Grids like these can be traversed not only with binary search, i.e. search each link list from front and back respectively,

A picture containing window, building, fence, door

Description automatically generated

Figure 6

[xn , xlen(B) - n ], at each node in the grid we can check each of the adjacent nodes. This allows for a diagonal search where every second and third diagonal is skipped, eliminating the need to traverse half of any dataset. You can for a diagonal traversal, you pass a link for every node in each parallel. Starting at your 1st head node, you can see your east node, and your south node. Then, moving southeast by one node brings us to our first node containing locations of 8 adjacent nodes. Technically, we can search a 3x3 node grid in O(2) time complexity. Once we have reached our southeast node, we can look at the data of the next southeast node, the southern node, the southwestern node, and then even the northeastern node.

If you consider much larger grids, say 1000 x 1000, our first diagonal search would cover diagonal line 0, -1, and 1, if we say that diagonal line 0 is the diagonal line which begins at the uppermost left node and intersects all connected southeastern nodes. Diagonal line -1 is then the line parallel to the left, and diagonal line 1 is the line parallel to the right. The time complexity of such a traversal through the entire structure then turns out to be, O, considering that we can still run binary search in the same fashion as before, using each first node to quickly locate the last.

Implementing the data structure of a linked grid is the least of issues though in these problems. The problem with the problems in the *NP-Complete* set is that they do not all adhere to the same rules. Looking at our first line in the *NP-Complete* solution statement,

*Items & Locations: A = { a ∈ A | a = {( x1 , l1), ( xn , l­n), …}}*

we can see that we have a list of *n tuples*, containing elements *a*. Each element *a* contains a *tuple*, (xn, ln), which contains the node data, and the location of that data. Our linked grid contains not only that, but also the data and locations of the

surrounding nodes. This is to say that, technically, you can create a structure of nested lists to solve the same problems, however the efficiency of your program will suffer greatly. The second statement,

*New I & L’s: Bn ⊆ A = {* *bn ∈ A | bn = a given the new solution given the ruleset of the problem given A }*

states that our list *B* is a subset of list *A,* given that an element *bn* is equal to a corresponding element *a,* given that *a* has been checked for necessary manipulation, and manipulated if required. If we’re looking at Sudoku here, the first step is to create our grid with our given numbers as our data, at there corresponding locations. Then, here in step two, we make our first set of moves. We say for each empty location of the board, check the associated row, column, and box, and see if any number is required, if not, move on to the next empty location. If a number is required, place it, and do change it again. You will have to iterate through this step many times before the entire board is complete, which brings us to our step 3, which is simply a loop of step 2.

*Next I & Ls Bn+1 ⊆ Bn = {bn+1 ∈ A | bn = the next required solution given the ruleset . of the problem given bn }*

Then, finally, when we reach a state where all requirements have been filled, we can say that we have reached our final step,

*Final I & L Sorted: C ⊆ Bn = { c ∈ Bn  | c = the final solution c given the ruleset of the problem given bn }*

which simply states that all of our lists have been subsets of their prior, and the final list is that of a complete solution. In problems like Minesweeper, you may need to restart from scratch, or the last correct point you reached.

**NP-Hard Solution: The Halting Problem**

Given a set of instructions, will a program *finish* or *run forever*?

Why do *programs* not end?

Programs do not finish when:

1. There is not a satisfactory answer no matter what solution is processed.
2. The Boolean state of the program remains unchanged.
3. The total number of solutions is infinite

Ways to know if your *program* might end:

1. If you are waiting for an external force to cause the program to halt, the program will never end unless the external force acts.
2. If you have checked all the possible solutions to the problem, and you do not have a satisfiable solution, the program will never end if it is instructed to end only with an answer.
3. If you ask the program to run every solution of an algorithm that has more solutions than there is time for in a thousand years, it will never stop giving solutions

The solution to the Halting Problem is located on the next page. Once again, this solution is not complicated, and in pseudocode makes perfect sense. The following are the *Laws of Data* we are basing our program on.

***Law 1:*** If your total iterations is larger than the total possible solutions, then your program is creating replicate data and will continue doing so forever.

***Law 2:*** Based on average time, if the estimated time to complete the program, is greater than the time allowed, the program is determined to run forever.

***Law 3:*** If your program does not halt until specific external data is sent to it there is no way to estimate the time it will take to receive the correct data, and therefore may run forever, but may also end at any given time.

***Law 3.1:*** If you can parse the data of the external file, Law 3 is false.

The NP-Hard solution on the following page is an example of the NP complete solution where only variables are required, rather than sets.

**Solution**

Program

|  |
| --- |
| m = number of external stimuli |
| n = total iterations |
| x = total possible iterations |
| y = average time in seconds to check each solution |
| z = False |
| a = estimated time to completion in hours = ( ((n \* y) / 60) / 60) |
|  |
| if a > hours /lifetime: |
| z = True |
| else: |
| z = False |
| if ( n > x): |
| z = True |
| else: |
| z = False |
| if m > 0: |
| It is impossible to determine when or if the program will finish |
| if m = 0: |
| The Program might finish |
| if z = False: |
| Program Will Finish |
| Print(‘Time to completion: ’, a) |
| Else: |
| Program Will Not Finish |
| Halt Program |

Everything in the above code however is in simplest form. The true code required to write this program to check a whole program is substantial, but entirely possible. It would essentially run as a dummy AI that parsed the lines of the code and created all of the variables required. When a program has to follow files, or most of the time, you then also have to parse those lines for data.

The solution is created by what we did on the previous page. When you break a problem down into its parts, and create rules based on what you know, you can find that certain answers become required. You don’t guess. There are only so many ways a program can go on forever. Based on those ways, the question becomes*, ‘how can we know if and when this is going to happen?’* That question is much easier to solve.

As stated before, the program on the following page is pseudocode, and will not get the job done. The logic should make sense however the variables might not. Tracking the total iterations in a program is as simple as creating a counter. Tracking the total possible iterations of a program, however, is not as simple. To do so we have to analyze the program. For that, we can do something like this.

Program

|  |
| --- |
| Initialize Read of program file |
| Line List = [ ] |
| Words in Lines List = [ [ ] ] |
| Last Line = [ ] |
| Current Line = [ ] |
| Tab List = [ ] |
| CurTab = 0 |
| PrevTab = 0 |
| Tabs = 0 |
| Loops = 0 |
| Nested Loops = 0 |
| Multiply Nested Loops = 0 |
| Total Loops = 0 |
| Lines in File = 0 |
| Largest Tab = 0 |
| Greatest Loop Tab = 0 |
| New Loop Tab = 0 |
| Last Line Loop = False |
| n = 0 |
|  |
| For line in program: |
| Cut each line into list of individual words |
|  |
| For line in Line List: |
| Count tabs before first word |
| Append total tabs to tab list |
| Carry Largest Tab and Return it |
|  |
| For word in Words in Lines List: |
| n += 1 |
| CurTab = Tab List[n] |
| If word in list == ‘for’ or word == ‘while’ and  CurTab > Greatest Loop Tab: |
| Loop += 1  Save Greatest Loop Tab |
| Save New Loop Tab |
| Else: |
| return |
|  |
| If word in list == for or word == while and last line loop = True: |
| Nested Loops += 1 |
| Save Greatest Loop Tab |
| Save PrevLoop Tab |
|  |
| If word in list == ‘for’ or word == ‘while’ and Tabs ==  Greatest Loop Tab and Last Line Loop = False: |
| Nested Loops += 1 |
| Save Greatest Loop Tab |
| Save PrevLoop Tab |
|  |
| If CurTab < PrevLoop Tab and word in list == ‘for’ or word ==  ‘while’ or word == ‘do’: |
| Loops += 1 |
|  |

**Conclusions**

From that code we can at least know if we have lists, and if we have nested lists. Checking for multiply nested loops is also possible, but is outside the scope of this paper. Once you have all the loops you can check the length of whatever they are iterating through. Then you run the program through 1% of total operations, calculate the time of each iteration, and then you have every number you might ever need. At that point you can check to see whether or not the program you wrote will end in a time you can afford. If you do not have 1000 years to complete the program you would like to run, you can consider the following option. The table on the following page is a description of the question statement, *‘if a program is going to take 1000 years to complete,*

*how many days will it take to complete if run in parallel on 10,000 Cores?’*

This question is important to answer so that we can understand what is practical today versus what is not. Or what is practical for you versus what is practical for them. If you want to build something largely scalable, like a social network or a news publication, you are going to need to access data connecting one account out of x million to another, and you have 0 seconds to do it. The only way to reasonably search/sort everything is to distribute sorting packets across networks, search/sort each packet, and then send back and reassemble the sorted packets, or just send back the correct connection. Even if you have a program that would take the equivalent of 1000 years to complete, you can still split it up and run it in parallel out in the cloud for a reasonable price. The cost is not so reasonable when you start adding any more 0’s after 1000.

Solving N = P is one thing, but solving the computability issue of running any program in polynomial time that requires you to search through any more than O(n(100 Million)), at 100 iterations per second is another. That program alone would take about 11 days to compute on one machine. All the questions can be answered using these methods, but no one knows how long it will take to find them, or how much it will cost.

**Other Relevant Things**

1. A tree is a set of nodes, but a set of nodes is not necessarily a tree. The term, *‘tree,’* is only used to describe the configuration of that set of nodes.
2. Since a tree is a set of nodes, we then define a node as an object of a class containing, a *self*-parameter, *n* data parameters, and *n* links.
3. All data structures contain a *‘head’* node.
4. Any shape of data structure imaginable can be created from a head node.
5. Given the prior knowledge, it is easy to see how you could create a linked list network of categorized data.
6. The Following code proves this. You can run it in the latest update to date of Python 3.7

Program

|  |
| --- |
| import time |
|  |
| class Node: |
| def \_\_init\_\_(self, data1, data2): |
| self.data1 = data1 |
| self.data2 = data2 |
| self.next = None |
| self.prev = None |
| self.arm = None |
|  |
| class DoubleNodeDoubly: |
| def \_\_init\_\_(self): |
| self.head = None |
|  |
| def split(self): |
| return list(self) |
|  |
| def convert(self): |
| new = "" |
|  |
| for x in self: |
| new += x |
|  |
| return new |
|  |
| def append(self, data1, data2): |
| if self.head is None: |
| double\_node = Node(data1, data2) |
| double\_node.prev = None |
| self.head = double\_node |
| else: |
| double\_node = Node(data1, data2) |
| cur = self.head |
| while cur.next: |
| cur = cur.next |
| cur.next = double\_node |
| double\_node.prev = cur |
| double\_node.next = None |
|  |
| def build\_layer(self, dllist): |
| if self.head is None: |
| return |
| else: |
| cur = self.head |
|  |
| while cur is not None: |
| if cur.next is not None: |
| cut = cur.data1 |
|  |
| wList = [ ] |
|  |
| for char in cut: |
| wList.append(char) |
|  |
| last = len(wList) - 1 |
|  |
| if last > 1: |
| wList.pop(last) ## Delete \n |
| wList.pop(last - 1) ## Delete Last Character |
|  |
| new = "" |
| for x in wList: |
| new += x |
|  |
| new\_word = new |
|  |
| dllist.append(new\_word, None) |
|  |
| cur = cur.next |
| else: |
| cur = self.head |
|  |
| while cur is not None: |
| cut = cur.data1 |
|  |
| wList = [ ] |
|  |
| for char in cut: |
| wList.append(char) |
|  |
| last = len(wList) - 1 |
|  |
| if last > 1: |
| wList.pop(last) ## Delete Last Character |
|  |
| new = "" |
| for x in wList: |
| new += x |
|  |
| new\_word = new |
|  |
| dllist.append(new\_word, None) |
|  |
| cur = cur.next |
|  |
|  |
| def build\_layer\_n(self, dllist): |
| if self.head is None: |
| return |
| else: |
| cur = self.head |
|  |
| while cur is not None: |
| cut = cur.data1 |
|  |
| wList = [ ] |
|  |
| for char in cut: |
| wList.append(char) |
|  |
| last = len(wList) - 1 |
|  |
| if last > 1: |
| wList.pop(last) ## Delete Last Character |
|  |
| new = "" |
| for x in wList: |
| new += x |
|  |
| new\_word = new |
|  |
| dllist.append(new\_word, None) |
|  |
| cur = cur.next |
|  |
|  |
| def link\_lists(self, L2): |
| first\_node2 = L2.head |
|  |
| if self.head is None: |
| return |
| else: |
| cur = self.head |
| while cur.next: |
| cur.arm = first\_node2 |
| cur = cur.next |
| cur.arm = first\_node2 |
|  |
| def print\_list(self): |
| cur = self.head |
| while cur: |
| print(cur.data1, cur.data2) |
| cur = cur.next |
|  |
| def print\_list\_arms(self): |
| cur = self.head |
| while cur: |
| print(cur.data1, cur.data2, cur.arm) |
| cur = cur.next |
|  |
|  |
|  |
| print("Linked List Started...") |
|  |
| new\_time = time.time() |
| full\_time = time.time() |
|  |
| print("for loop beginning now...") |
|  |
| dllist1 = DoubleNodeDoubly() |
|  |
| dllist1.append('a',None) |
| dllist1.append('b',None) |
| dllist1.append('c',None) |
| dllist1.append('d',None) |
| dllist1.append('e',None) |
| dllist1.append('f',None) |
| dllist1.append('g',None) |
| dllist1.append('h',None) |
| dllist1.append('h',None) |
| dllist1.append('j',None) |
| dllist1.append('k',None) |
| dllist1.append('l',None) |
| dllist1.append('m',None) |
| dllist1.append('n',None) |
| dllist1.append('o',None) |
| dllist1.append('p',None) |
| dllist1.append('q',None) |
| dllist1.append('r',None) |
| dllist1.append('s',None) |
| dllist1.append('t',None) |
| dllist1.append('u',None) |
| dllist1.append('v',None) |
| dllist1.append('w',None) |
| dllist1.append('x',None) |
| dllist1.append('y',None) |
| dllist1.append('z',None) |
|  |
| dllist2 = DoubleNodeDoubly() |
|  |
| file = open("dictWords.txt", "r") |
| lines = file.readlines() |
|  |
| x = 0 |
|  |
| for line in lines: |
| x += 1 |
| dllist2.append(line, None) |
| print(x," --- %s seconds ---" % (time.time() - new\_time),line) |
|  |
| print("for loop complete") |
|  |
| print("Dictionary Words List Built in:") |
| print("--- %s seconds ---" % (time.time() - new\_time), "\n") |
|  |
| dllist3 = DoubleNodeDoubly() |
| dllist4 = DoubleNodeDoubly() |
| dllist5 = DoubleNodeDoubly() |
| dllist6 = DoubleNodeDoubly() |
| dllist7 = DoubleNodeDoubly() |
| dllist8 = DoubleNodeDoubly() |
| dllist9 = DoubleNodeDoubly() |
| dllist10 = DoubleNodeDoubly() |
| dllist11 = DoubleNodeDoubly() |
| dllist12 = DoubleNodeDoubly() |
| dllist13 = DoubleNodeDoubly() |
| dllist14 = DoubleNodeDoubly() |
| dllist15 = DoubleNodeDoubly() |
| dllist16 = DoubleNodeDoubly() |
| dllist17 = DoubleNodeDoubly() |
| dllist18 = DoubleNodeDoubly() |
| dllist19 = DoubleNodeDoubly() |
| dllist20 = DoubleNodeDoubly() |
|  |
| new\_time = time.time() |
|  |
| dllist2.build\_layer(dllist3) |
| dllist3.build\_layer\_n(dllist4) |
| dllist4.build\_layer\_n(dllist5) |
| dllist5.build\_layer\_n(dllist6) |
| dllist6.build\_layer\_n(dllist7) |
| dllist7.build\_layer\_n(dllist8) |
| dllist8.build\_layer\_n(dllist9) |
| dllist9.build\_layer\_n(dllist10) |
| dllist10.build\_layer\_n(dllist11) |
| dllist11.build\_layer\_n(dllist12) |
| dllist12.build\_layer\_n(dllist13) |
| dllist13.build\_layer\_n(dllist14) |
| dllist14.build\_layer\_n(dllist15) |
| dllist15.build\_layer\_n(dllist16) |
| dllist16.build\_layer\_n(dllist17) |
| dllist17.build\_layer\_n(dllist18) |
| dllist18.build\_layer\_n(dllist19) |
|  |
| print("Layers Completed in: --- %s seconds ---" % (time.time() - new\_time)) |
|  |
| print(dllist16.print\_list(), "\n") |
| print(dllist17.print\_list(), "\n") |
| print(dllist18.print\_list(), "\n") |
|  |
| print(dllist2.head) |
| print(dllist3.head) |
|  |
| print("Program Completed in: --- %s seconds ---" % (time.time() - full\_time)) |

It is recommended to use a list like this one when you need to move through data quickly, and when you need to compare multiple variables quickly, and have the ability to get data from the previous and next node from each link. Then when you have found the right piece of data in that section you can move to the next layer and check for that data, and keep doing so through categorized layers until you arrive at a final node, with data you need. The previous code was an example of a linked list network that creates a dictionary that can be accessed for any word by parsing the word and moving through layers letter by letter.

Further,

LOGSPACE ⊆ P⊆ NP⊆ PSPACE

A simple diagonal argument shows that the first is a proper subset of the last, but

we cannot prove any particular adjacent inclusion is proper.

…

The first statement given in the proposed question written by Stephen Cook. The statement can be solved by looking at the NP solution next to a solution in PSACE.

Items: A = { x ∈ A | x = an }

Whitelist: B ⊆ A = { y ∈ A | y = ( xm , xn ) }

Random List: C ⊆ A = { z ∈ A | z = ( xn , xlen(B) - n ) }

Sorted List: D = D ⊆ C = { d ∈ C | any z ≠ any z and any ∈ zn ≠ any other ∈ zn }

***Is a subset of,***

Items & Locations: A = { a ∈ A | a = {( x1 , l1), ( xn , ln), …}}

New I & Ls: Bn ⊆ A = { bn ∈ A | bn = the new solution given the ruleset of the problem given A }

Next I & Ls Bn+1 ⊆ Bn = {bn+1 ∈ A | bn = the next required solution given the:rulesetof the problem given bn }

.

Final I & L Sorted: C ⊆ Bn = { c ∈ Bn | c = the final solution c given the ruleset of the problem given bn }

References

1. Stephen Cook, The P Versus NP Problem, Available at, http://www.claymath.org/sites/default/files/pvsnp.pdf, Accessed in March 2020.

2. Stephen Cook, The P Versus NP Problem, Available at, http://www.claymath.org/sites/default/files/pvsnp.pdf, Accessed in March 2020.

3. Stephen Cook, The P Versus NP Problem, Available at, http://www.claymath.org/sites/default/files/pvsnp.pdf, Accessed in March 2020.

4. Stephen Cook, The P Versus NP Problem, Available at, http://www.claymath.org/sites/default/files/pvsnp.pdf, A­­ccessed in March 2020.