COMPLEXITY CLASSES

ALEX THOLEN

1. Preliminaries

Definition 1.1. A *problem* is described by giving a description of all of its parameters and what properties the solution is supposed to satisfy.

For example, Sudoku is defined as giving the rules for sudoku (where each number appears once in each square, row, and column) and the size of the grid - 9 x 9 for example.

Definition 1.2. An *instance* of a problem is described by describing the details for some (or all, depending on the problem) of the parameters.

For example, an instance of Sudoku could be like this:

1	2		
	3		
2		4	
			1

Definition 1.3. A *solution* of a problem to an instance is something that satisfies all of the properties a solution of that problem must solve with the parameters described in the instance.

For example, a solution of the 4x4 instance

1	2		
	3		
2		4	
			1

would be

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

Definition 1.4. An *algorithm* for a problem is a step-by-step procedure for solving problems. It is said to solve a problem Π if, given any instance of Π, the algorithm is guaranteed to provide a solution for said instance.

Definition 1.5. A *function problem* Π is a triple (D, S, σ) that satisfies the following:

- D is the set of all instances of a problem.
- S is the set of all possible solutions to the problem.
- σ is a mapping from D to 2^S.

Date: December 8, 2019.

For example, D might be the set of all Sudoku boards, with empty squares and even boards with no solution, such as one with multiple of a number in the same row. S is the set of all filled out Sudoku boards, and then σ is a mapping from D to 2^S that given an element of D outputs a set of elements of S that satisfy the problem.

Definition 1.6. If f and g are function N to N or R to R, then we say that $f(x) = O(g(x))$ if there exists $c, L > 0$ such that for all $x > L$ $f(x) < c \cdot g(x)$.

There is no requirement for this to be the smallest $g(x)$ that works. For example $7x^4+3x^2 = O(e^x)$, but it is best described as $O(x^4)$.

Definition 1.7. The time complexity of an algorithm to solve a problem is the function where $f(n)$ is the maximum amount of steps that algorithm could take to solve an instance of size n. It is typically described in O notation.

Note that this definition doesn't include what is considered a step and how to encode an instance into a string to measure the size of. However, neither of these factors matter for NP-completeness.

Definition 1.8. For an algorithm to solve a problem in polynomial time means that its time complexity is a polynomial in terms of time.

Definition 1.9. A non-deterministic algorithm is an algorithm allowed to go down multiple probability paths to find the answer. In other words, checking an answer, as then the algorithm runs down the path of each possible solution and then checks if they are correct.

Definition 1.10. A problem is in NP if there is a non-deterministic algorithm that solves it in polynomial time. In other words, if it takes polynomial time to check if an answer is correct.

Definition 1.11. A problem Π is an NP-Complete problem if Π is in NP and if any NP problem there is a polynomial time reduction from Π to that problem. In other words, if given an oracle that could solve Π, could then be solved in polynomial time.

2. Another Solution Problem

Definition 2.1. Given a function problem $\Pi = (D, S, \sigma)$ and an integer n, the Another Solution Problem (ASP) is the function problem $\Pi_{[n]} = (D_{[n]}, S, \sigma_{[n]})$ where $D_{[n]} = \{(x, S_x) | x \in D, S_x \in \sigma(x), |S_x| = n\}$ and $\sigma_{[n]}(x, S_x) = \sigma(x) - S_x$.

To put this in regular English, what this means is given n solutions to a problem, the n-ASP is to find another solution.

Definition 2.2. The class FNP consists of function problems $\Pi = (D, S, \sigma)$ such that the following holds:

- There exists a polynomial p such that $|s| \leq p(|x|)$ holds for any $x \in D$ and any $s \in \sigma$.
- For any $x \in D$ and $y \in S$, the proposition $y \in \sigma(x)$ can be determined in polynomial time.

Definition 2.3. Let $\Pi_1 = (D_1, S_1, \sigma_1)$ and $\Pi_2 = (D_2, S_2, \sigma_2)$ be function problems. We say that Π_1 is polynomial-time ASP reducible to Π_2 (denoted by $\Pi_1 \preceq \Pi_2$) if there exists ψ_D, ψ_S such that

- ψ_D is a polynomial-time computable mapping from D_1 to D_2.

- For any $x \in D_1$, ψ_S is a polynomial-time computable bijection from $\sigma_1(x)$ to $\sigma_2(\psi_D(x))$.

From this definition we can see that this property is invariant with respect to taking an ASP. In other words, the following proposition:

Proposition 2.4. *$\Pi_1 \preceq_{ASP} \Pi_2$ then $\Pi_{1[n]} \preceq_{ASP} \Pi_{2[n]}$ for any nonnegative integer n.*

Proposition 2.5. *For any function problem $\Pi = (D, S, \sigma)$ and nonnegative integers m, n, then $(\Pi_{[m]})_{[n]} \preceq_{ASP} \Pi_{[m+n]}$.*

Proof. An instance of $(\Pi_{[m]})_{[n]}$ is of the form $((x, \{s_1, \ldots, s_m\}), \{t_1, \ldots, t_n\})$ where $x \in D, s_1, \ldots, s_m, t_1, \ldots, t_n \in \sigma(x)$. Now we can set $\psi_D(\overline{x})$ to be $(x, \{s_1, \ldots, s_m, t_1, \ldots, t_n\})$. Then the solutions of $\overline{x}$ are the same as the solutions of $\psi_D(\overline{x})$ and so with ψ_S to be the identity function this proposition holds. ■

From these two propositions we can see the following result.

Theorem 2.6. *Let Π be a function problem. If $\Pi \preceq_{ASP} \Pi_{[1]}$, then for any nonnegative integers $n < m$ we have $\Pi_{[n]} \preceq_{ASP} \Pi_{[m]}$.*

Proof. From 2.4 we can see that $\Pi_{[n]} \preceq_{ASP} (\Pi_{[1]})_{[n]}$, and from 2.5 we get that $\Pi_{[n]} \preceq_{ASP} \Pi_{[n+1]}$. The rest comes from induction. ■

3. ASP-Completeness

Definition 3.1. A function problem Π is *ASP-complete* if and only if $\Pi \in$FNP, and $\Pi' \preceq_{ASP} \Pi$ for any $\Pi' \in$FNP.

Proposition 3.2. *Let Π and Π' be function problems. If Π is ASP-complete, $\Pi' \in$FNP and $\Pi \preceq_{ASP} \Pi'$, then Π' is ASP-complete.*

The ASP-complete problem we begin with is SAT. SAT represents the function problem of satisfiability. In other words, given a first order logic statement the problem is either to find a set of truth values that satisfies it or detect if there is one. We are dealing with the first one. When [1] proved that SAT was NP-complete, he used an ASP reduction . That implies the following theorem:

Theorem 3.3. *SAT is ASP-complete*

One important property of ASP-completeness is that it implies NP-completeness. Let's begin by showing that this is true for SAT.

Theorem 3.4. *First that $SAT \preceq_{ASP} SAT_{[1]}$, and then using this for any nonnegative integer n, $SAT_{[n]}$ is NP-complete*

Proof. We will construct a polynomial time ASP reduction ψ_D, ψ_S from SAT to $\text{SAT}_{[n]}$. To construct ψ_D, we will construct a ψ' from a ψ. They are both CNF formulas - a big string of statements consisting of logical nots and logical ands. We make a new variable w, and for each clause $l_1 \vee l_2 \vee \cdots \vee l_r$ in ψ, we make the clause $l_1 \vee l_2 \vee \cdots \vee l_r \vee w$ to ψ' . Then we also add the clauses $x \vee \overline{w}$ for each variable x. We define ψ_D to be $(\psi', \{g\})$ where g is the assignment in which all variables are true. For the second part to be true, using the first part and 2.6 it becomes clear that $\text{SAT} \preceq_{ASP} \text{SAT}_{[n]}$ and thus $\text{SAT}_{[n]}$ is NP-complete. ■

With this we can see the following theorem.

Theorem 3.5. *For any ASP-complete function problem* Π *and any nonnegative integer* n, $\Pi_{[n]}$ *is NP-complete.*

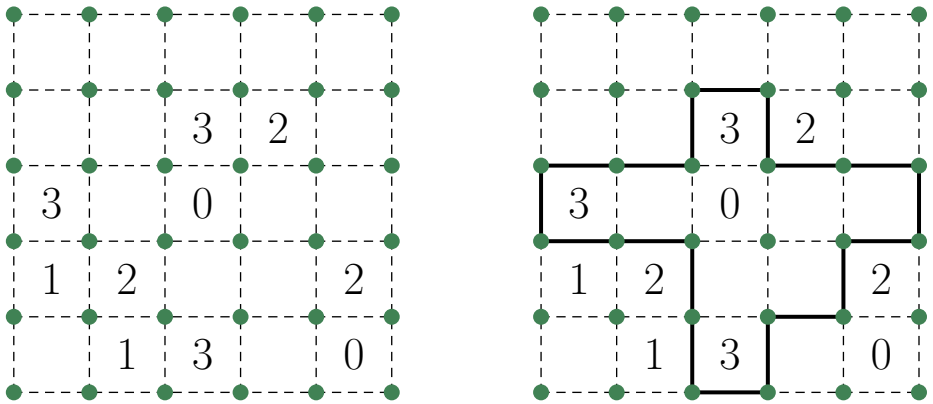
Proof. Since Π is ASP-complete, we know that SAT $\preceq_{ASP}$ Π. We can take the nth ASP of both sides because of 2.4, and so we have $\text{SAT}_{[n]} \preceq_{ASP} \Pi_{[n]}$. Since we just proved that $\text{SAT}_{[n]}$ is NP-complete in 3.4, we get that $\Pi_{[n]}$ is NP-complete. Note that this also applies for Π, so this is relevant even with just the original problem. ■

4. NP-complete Puzzles

4.1. **Slither Link.** The rules of Slither Link is as follows:

- Each problem is given a rectangular lattice. The length of sides of the rectangle (as to the unit length of lattice) is called the size of the problem.
- A 1x1 square surrounded by four points is called a cell. A cell may have a number out of 0,1,2, or 3.
- The goal is to make a loop which does not intersect or branch by connecting adjacent dots with lines, so that a number on the cell is equal to the number of lines drawn around it.

The following is an example of Slither Link.



To prove that Slither Link is ASP-complete (and so NP-complete) we will use two known facts:

Lemma 4.1. *To find a Hamiltonian circuit for a given planar graph with degree at most* 3 *is ASP-complete.*

Lemma 4.2. *Any planar graph with degree at most* 3 *with* n *vertices can be embedded in an* $O(n) \cdot O(n)$ *grid in polynomial time in* n.

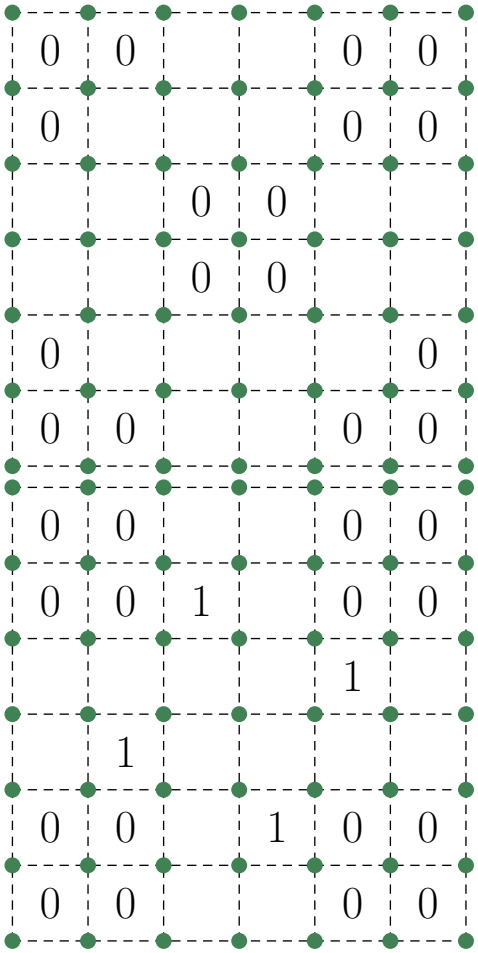
Now the proof.

Theorem 4.3. *Finding a solution to a given instance of Slither Link is ASP-complete*

Proof. The membership in FNP is easy to see. Now we will construct a polynomial time ASP reduction from the restricted Hamiltonian circuit problem to the Slither Link problem.

Using 4.2, we can transform a graph G of the restricted Hamiltonian circuit problem into a graph G' on the grid. Now some points on this graph G' have lattice points which don't correspond to any vertices of G. Those are points that don't need to be visited when considering Hamiltonian circuits of G'.

The strategy consists of turning points that need to be visited into one 6 x 6 grid, and points that don't need to be visited into another 6 x 6 grid, and that can become a Slither Link grid. We turn a lattice point which doesn't need to be visited into the top 6 x 6 grid, and one that does into the bottom.



We can join two gadgets by having a 3 x 6 border with either 0's blocking it or 0's allowing passage. This means we can turn any restricted Hamiltonian circuit into a Slither Link problem, and so Slither Link is ASP-complete and NP-complete. ■

4.2. **Sudoku.** So the 9x9 sudoku isn't quite general enough (after all, with just finite board states a program could just test for all of them, becoming quick). So this is the general form:

- A problem is given as an n^2 x n^2 grid, which is divided into n x n squares. The value n is called order.
- Some cells are filled with an integer from 1 through n^2.
- The goal is to fill in all the blank cells so that each row, column, and n x n square has each of the integers from 1 through n^2 exactly once.

To show that this is ASP-complete, we will use Latin squares.

Definition 4.4. A Latin square of order n is a matrix such that each row and column contains each integer from 1 through n exactly once. A partial Latin square is a matrix with

some blank entries such that each row and column contains each integer from 1 through n at most once.

Theorem 4.5. *The problem of partial Latin square completion is ASP-complete.*

Proof. Coulbourn has proven that partial Latin square completion is NP-complete. His reduction was what we defined as ASP reduction. ■

Now let's link these two problems together, with the following lemma.

Lemma 4.6. *Let S be a Sudoku problem of order n such that*

$$S(i,j) = ((i \mod n)n + \left\lfloor \frac{i}{n} \right\rfloor + j) \mod n^2$$

when $(i,j) \notin B$, where $B = \{(i,j) | \left\lfloor \frac{i}{n} \right\rfloor = 0$ and $(j \mod n) = 0\}$, and is empty otherwise. Then a square S' obtained by filling in the blanks of S is a solution to S if and only if

- *For any $(i,j) \in B$, $S'(i,j) \mod n$ equals 0.*
- *A square L defined by $L(i, \frac{j}{n}) = \frac{S(i,j)}{n}$ for all $(i,j) \in B$ is a Latin square*

Proof. What this theorem means is shown below, and as such follows from the definition of Sudoku. ■

While what the previous theorem says may seem complicated, what it actually means is that solving the Sudoku grid shown below is the same as the Latin square shown a bit farther down (with this being the $n = 2$ case)

A0	01	C0	11
B0	11	D0	01
01	10	11	00
11	00	01	10

A	C
B	D

Now we can prove that Sudoku is ASP-complete.

Theorem 4.7. *To find a solution to a given instance of Sudoku is ASP-complete.*

Proof. The membership in FNP is immediate. It isn't hard to check if a Sudoku board satisfies all the conditions. Now we need to show a polynomial time ASP reduction from the problem of partial Latin square completion to Sudoku.

For a given partial Latin square L of order n, we construct a Number Place problem S as follows:

$$S(i,j) = \begin{cases} L(i, \frac{j}{n}) \cdot n & \text{if } ((i,j) \in B, L(i, \frac{j}{n}) \neq \text{nothing} \\ \text{nothing} & \text{if } ((i,j) \in B, L(i, \frac{j}{n}) = \text{nothing} \\ ((i \mod n)n + \lfloor \frac{i}{n} \rfloor + j) \mod n^2 & \text{(otherwise)} \end{cases}$$

We can see that this can be done quickly. Now we can also see from Lemma 4.6 that this can be done in the other way. So, despite the fact that this doesn't care about the vast majority

of Sudoku games, we can see that being ale to solve all of them would mean also being able a partial Latin square, which is NP-complete. ■

So, we can see that Sudoku is NP-complete and ASP-complete. That also means that if you are given some amount of solutions to one layout, it is NP-complete to find another one. [3] [2] [1]

References

[1] Cook. The complexity of theorem-proving procedures, 1971.
[2] Coulborn. The computational complexity of recognizing critical sets, 1983.
[3] Yato and Seta. Complexity and completeness of finding another solution and its application to puzzles. 2003.