Introduction

How fast can things be computed on a computer? How difficult is it to play Pokémon? Are video games related to important problems in logic? These questions are addressed in the paper.

Complexity Theory

We can analyze time complexity of an algorithm. This is usually studied within the asymptotic framework. This means that we ask about how much the computation time increases with the length of the input, disregarding constant factors (both additive and multiplicative).

Complexity of a problem

Apart from determining time complexity of an algorithm (which is usually easy for algorithms use in practice), we can also be interested in complexity of a problem. By time complexity of a problem, we mean the complexity of the fastest algorithm that solves the problem.

Knowing an algorithm for the problem gives us an upper bound for the complexity of the problem. However, determining lower bounds for the complexity of the problem is usually more difficult. Typically, we use problem reduction for obtaining lower bounds. If A is efficiently reducible to B (which means that an algorithm for solving B can be used for solving A, after some modification with only minor time overhead), then problem B is at least as difficult as problem A.

Complexity classes

P (for Polynomial) is a class of all problems that can be solved deterministically in polynomial time. In simple words, P contains all problems that can be solved fast on a normal computer. Most programs that you run on your computer or smartphone use polynomial-time algorithms, thus they deal with problems from P. For instance, searching for the shortest path (or the fastest road trip) between two points on a road map is in P.

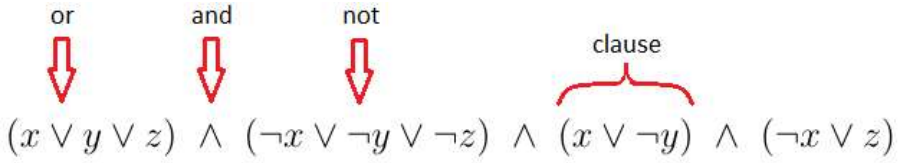
NP (for Nondeterministic Polynomial) is a class of problems such that when we are given a solution, we can efficiently (i.e. in polynomial time) check whether it is correct. For example, Sudoku (of general size $N^2 \times N^2$) is in NP, because we can easily check whether a given solution (a fully filled table) is correct, even though searching for a correct solution might be very difficult.

And if we have a problem X such that all problems from NP can be reduced to X, then problem X is NP-hard. The aforementioned Sudoku is NP-hard, because there exists a way how to encode any problem from NP into a Sudoku instance (a partially filled table). This might sound unbelievable, but yes, Sudoku is an example of such a “universal” problem.

The famous “P vs NP” problem asks whether classes P and NP are equal, which would mean that searching for solutions of problems is never significantly harder than verifying their solutions.

SAT

The most famous example of an NP-hard problem is SAT (a.k.a. Boolean Satisfiability Problem). We are given a logical formula (more accurately, a prepositional formula written in a conjunctive normal form), for example this formula.



In SAT we are asked to determine whether there exists an instantiation (assignment of either true or false to each variable) which makes the formula true.

first solution: $x = 0\ ,\ y = 0\ ,\ z = 1$

second solution: $x = 1\ ,\ y = 0\ ,\ z = 1$

3-SAT is a restricted version of SAT, in which each clause has at most 3 variables. The example above is an instance of 3-SAT. We are interested in 3-SAT, because it is the "smallest" version of SAT which is already NP-hard.

Pokémon games

Pokémon is a Japanese RPG game in which a player controls a fictional character who travels across a 2D world (square grid), collects items, catches monsters, battles other monsters and solves puzzles. Starting in early 1996, Pokémon has become the second most popular game series in the world.

Game mechanics

The mechanics of the game are complicated, but we will need only a small subset of them in our construction. In particular, we will move our game character on a map that contains only empty squares, inaccessible squares (rocks) and trainers. The character can move only in four directions (no diagonal movement) and only empty squares can be stepped on.

Trainers cannot walk through rocks and other trainers as well. Each trainer has a line of sight (a direction of looking and a distance he can see). If the character enters a trainer's line of sight, the trainer walks next to the character and a Pokémon battle is initiated automatically. The character can also initiate a battle, but to do so, the character must come next to the trainer (from any side) and speak to him.

If the character wins the battle, the trainer will stay on his spot forever and never battle again. If the adversary trainer wins the battle, the character will be transported to a Poké-center (located outside of our map), which is equivalent to "game over" (fail).

Our objective

We are interested in the "reachability problem" in Pokémon games. We are given a game map (with adversary trainers) and we are asked whether it is possibly for the player to win (reach the goal destination).

Because we study decision problems of general input size, we will have to generalize Pokémon games as well. In particular, the game map will have unlimited size. All other properties will be kept as they are in the original games. We want to show that this "reachability problem" is NP-hard.

Proof

Now we will describe a reduction from 3-SAT to Pokémon. That reduction will prove that Pokémon is at least as hard as 3-SAT, thus Pokémon is NP-hard. We will construct three gadgets made of obstacles and adversary trainers.

For simplicity, we will use only two kinds of trainers. Strong trainers (tall red figures) always defeat the player, thus stepping in their line of sight is like an instant death. Weak trainers (short blue figures) are so weak that they always lose against a player without depleting any resources. These weak trainers only act as an obstacle in the map or a blockage in the line of sight of strong trainers. Red and blue rectangles represent lines of sight belonging to strong and weak trainers (respectively).

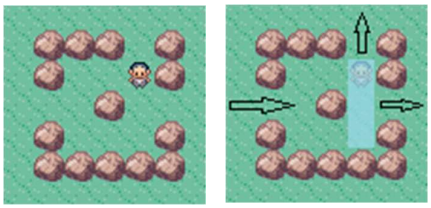


Figure 1: Variable gadget

The first gadget (Figure 1) represents a variable (fork). A player can either approach the trainer from the top to initiate a battle by speaking to the trainer, or he can walk by the bottom corridor and let the trainer walk one step down. Each of these actions will free a different exit and block the other exit forever. That is equivalent to assigning a value to a variable.

The second gadget (Figure 2) represents a clause (disjunction check). If any weak trainer is deactivated, he will stay in his spot and block the sight of the strong trainer on the right. On the contrary, if all of them are active (neither of them was visited before), they all will walk one step down and free the path for the strong trainer who is unavoidable for the player. This is equivalent to checking whether at least one of three elementary conditions (a variable or its negation) is true.

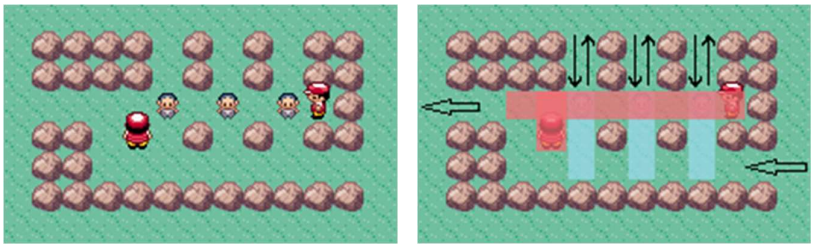


Figure 2: Clause gadget

The third gadget (Figure 3) represents a crossroad (one-way single-use crossing). The trainers are arranged in such a way, that entering from the top allows exiting at the bottom only; and entering from the left allows exiting at the right only.

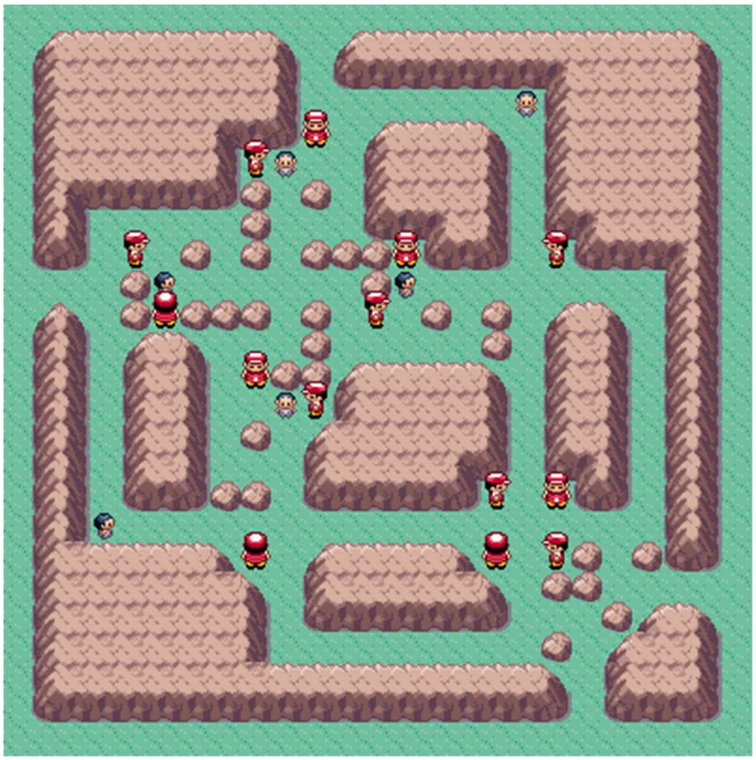


Figure 3a: Crossroad gadget

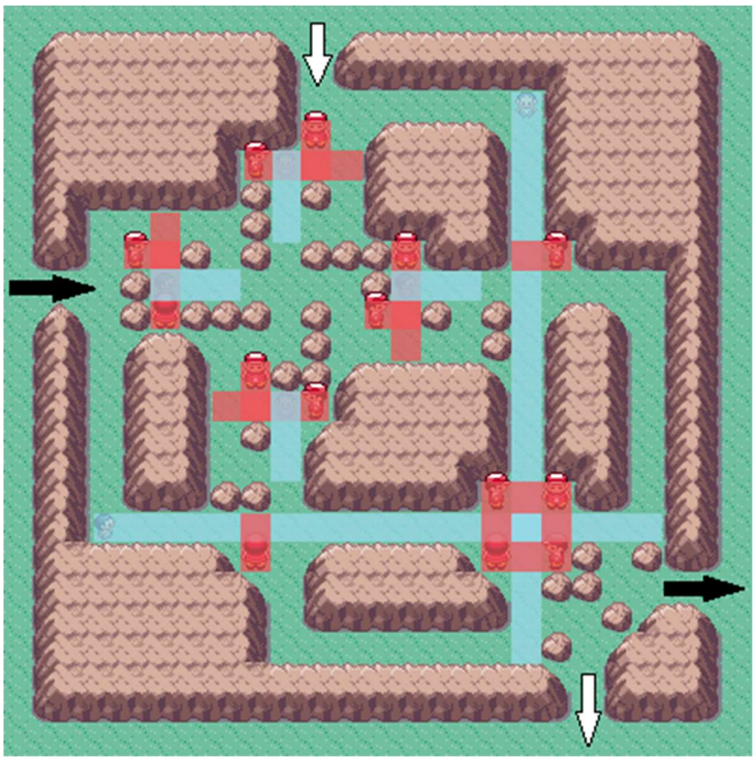


Figure 3b: Crossroad gadget

Now we must connect them together (Figure 4). In the first phase, the player is forced to go through all variable gadgets in sequence. After each true/false decision, the player is allowed to unlock all clause gadgets which contain the corresponding variable/negation. Crossroad gadgets are located at every intersection.

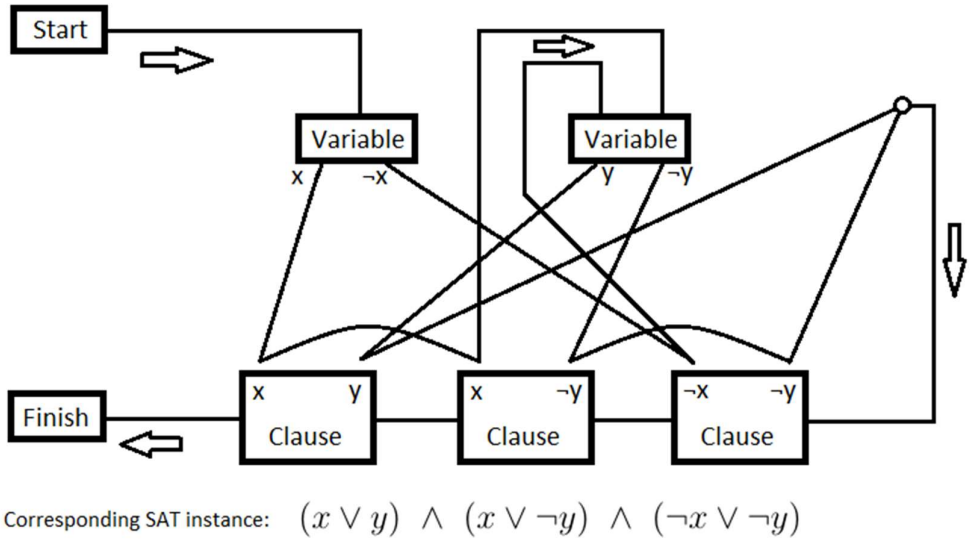


Figure 4: Connecting everything together (for simplicity, this example contains only two variables, thus only 2 out of 3 slots in each clause are used; but in general, there can be many variables, and clauses will often have all 3 slots connected to some variables)

In the second phase, the player is forced to pass through all clause gadgets (Figure 4, lower part). This verifies that all disjunctions have been made true. The player can succeed (reach the finish) if and only if the corresponding formula (SAT instance) is satisfiable.

Using this construction, we obtained a valid reduction from 3-SAT to Pokémon.

Conclusion

We proved that Pokémon is an NP-hard problem.

This implies that if a polynomial algorithm for Pokémon was discovered, that would prove P = NP and possibly spell chaos in our digitalized world (e. g. as a result of exploiting RSA cryptography by calculating private keys from public keys).

On the contrary, if we assume that P is a strict subset of NP, our proof gives hard limits for a performance of any artificial intelligence playing Pokémon. This “negative” result can be interesting e.g. for people who compete in tool-assisted speedrunning.

“Gaming is a hard job, but someone has to do it!” – Giovanni Viglietta (one of the authors)