Difficulty makes Candy Crush so addictive

March 13 2014, by Toby Walsh

Once you get a taste, it's hard to stop playing Candy Crush … but why? Credit: emi iemei/Flickr, CC BY-NC-SA

It's been said that in a city, you're never more than two metres away from a rat. But it seems more likely that you're never more than two metres from someone playing the puzzle game Candy Crush Saga.

One possible reason that Candy Crush is so addictive has been revealed in my recent proof which shows that it is a mathematically hard puzzle to solve, thus joining the ranks of puzzles such as Minesweeper, Sudoku and Tetris which have also been proved to be mathematically hard to solve.

Candy Crush is currently the most popular game on Facebook and it's

been downloaded and installed more than half a billion times. Its developer, King.com Ltd, is soon to list on the New York Stock Exchange in an initial public offering predicted to value the company at more than US$5 billion.

That's not bad for a simple game of swapping candies to form chains of three or more identical candies.

So how do we go about proving that a puzzle such as Candy Crush is hard to solve? What makes noughts and crosses easy, but Candy Crush hard?

Deduction from reduction

To prove that Candy Crush is hard, we need to call upon one of the most important and beautiful concepts in the whole of computer science: the idea of a problem reduction. This is the idea of mapping one problem into another. Or as computer scientists like to say, you reduce one problem into another.

Now, if the problem you started with was hard, then the problem you map into must be at least as hard. The second problem can't be easier as you can solve the first problem with a program that can solve the second. And if you can show the reverse, that the second problem can be reduced to the first, then the two problems are equally as hard.

In the case of Candy Crush, we started with the most famous class of computationally hard problems – problems in a class called "Non-deterministic Polynomial-time" (NP). This class contains all the problems that computer scientists believe are on the boundary between hard and easy.

Beneath NP, we have the problem class P that contains all the easy

problems such as sorting a list or finding a record in a database. The time it takes for an efficient computer program to solve such problems is small, even in the worst case.

Mathematically, the runtime (the length of time a program takes to run) is a polynomial (consisting just of terms multiplied together) of the size of the problem.

Above NP, we have really hard problems. There are even problems above NP for which there is no computer program which is guaranteed to stop and return an answer. You may wait for ever and still not have an answer.

NP, on the other hand, is right at the boundary between easy and hard. Within NP, we have many challenging problems such as the problem of routing trucks to deliver parcels, rostering staff in a hospital, or scheduling classes in a school. Any of these problems can be reduced to any other of these problems. They're all equally as hard as each other.

The best computer program that we have for any problem in NP has a runtime that grows dramatically as we increase the size of the problem.

On my desktop computer, I have a program that takes a few hours to find the optimal routing for ten trucks and demonstrate that this was the best that can be possibly achieved.

But for 100 trucks, the same program would take more than the lifetime of the universe. Mathematically, the runtime of my program is an exponential of the size of the problem.

Deciding between 'easy' and 'hard' is, um, hard

Surprisingly, while computer scientists believe problems in NP are on

the boundary between easy and hard, they don't actually know on which side they are.

The best computer programs we have take exponential time to solve problems in NP. But we don't know if there's some exotic algorithm out there that will solve problems in NP efficiently – and by efficiently, we mean in polynomial time.

In fact, this is one of the most important open problems in mathematics today, the famous P=NP question. The Clay Mathematics Institute has even offered a US$1 million prize for the answer to this question. The prize remains unclaimed since it was first offered in 2000.

The idea of problem reduction is central to the P=NP question. If we did find an algorithm that could solve any problem in NP efficiently then, by exploiting the idea of problem reduction, we could solve all problems in NP efficiently. The world would be a very different place if this ever happened.

On the plus side, we'd be able to go about our lives more efficiently, routing trucks, timetabling flights, and rostering staff to save money, but the absence of efficient algorithms to do various tasks such as crack codes is also required to keep our passwords and bank accounts secure.

Hard candy

Anyway, let's go back to Candy Crush. To show that Candy Crush is a hard problem, we could reduce from any problem in NP. But to make life simple, we start from the granddaddy of all problems in NP: finding a solution to a logical formula.

Mathematically, this is called the satisfiability problem. You will have solved such a problem if you ever tackled a logic puzzle. You have to

decide which propositions to make true, and which to make false, in order to satisfy some logical formulae.

We created a collection of candies where the player has choices as to which chains to create. And these choices have knock-on effects that mirror exactly the knock-on effects of the choices in deciding which propositions in a logical formula to make true.

We also showed the reverse – that is, you can reduce a Candy Crush problem to satisfying a logical formula. Hence, Candy Crush is no harder than any of the problems in NP. Candy Crush is equally as hard as solving all the other problems in NP.

If we had an efficient way to play Candy Crush, the we would have an efficient way to route trucks, roster staff or schedule classes. Equally, if we had an efficient way to way to route trucks, roster staff, or schedule classes then we would have an efficient way to play Candy Crush. That's the power of a problem reduction.

So, next time you fail to solve a Candy Crush board in the given number of swaps, you can console yourself with the knowledge that it was a mathematically hard problem to solve. Just as hard, in fact, as the staff rostering problem your boss actually wanted you to solve instead of playing Candy Crush.

Finally, there's an intriguing possibility that may appeal to Candy Crush addicts. Can we profit from the millions of hours humans spend solving Candy Crush problems? By exploiting the idea of a problem reduction, perhaps we can hide some practical computational problems within these puzzles?

Provided by The Conversation