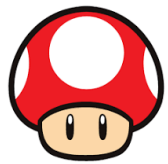
Mario is Hard

An Introduction to Complexity and Reductions

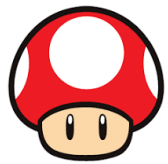
Gautham Viswanathan

CMI Student Seminar



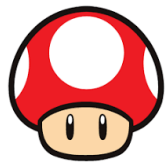
Mario is Hard

- Mario is hard!



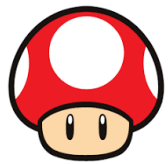
Mario is Hard

- Mario is hard!
- What does that even mean?



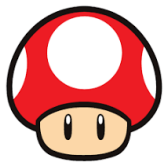
Mario is Hard

- Mario is hard!
- What does that even mean?
- ~~git gud lol~~
- One CS-y way to measure this is how much a computer struggles to solve the problem
- If a computer takes a few seconds to solve a problem, the problem is probably much easier than one which takes the computer a few years



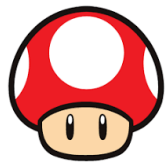
Idealized Difficulty

- Beefy gaming rigs can do things faster than beaten up old machines



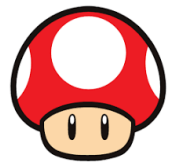
Idealized Difficulty

- Beefy gaming rigs can do things faster than beaten up old machines
- So we want a theoretical measure of difficulty



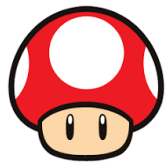
Idealized Difficulty

- Beefy gaming rigs can do things faster than beaten up old machines
- So we want a theoretical measure of difficulty
- One idea along this line is to look at the number of steps needed to solve the problem



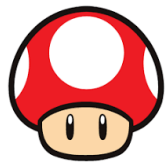
Idealized Difficulty

- Beefy gaming rigs can do things faster than beaten up old machines
- So we want a theoretical measure of difficulty
- One idea along this line is to look at the number of steps needed to solve the problem
- This won't depend on the computer we use



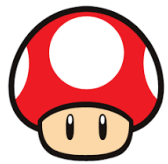
Idealized Difficulty (contd.)

- Big problems need big solutions
- Easy problem with a large input needs more time than a hard problem with a small input
- So we'll only compare inputs of the same size, and think of the difficulty as a function of the input size



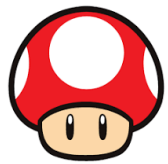
Complexity

- Now that we have a measure of how complex a problem is, how should we measure hardness?
- The obvious way to do this is to just look at the number of steps we need
- *That depends on the algorithm*
- Algorithm designers can be really really inefficient
- We want *inherent* hardness



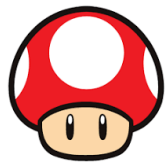
Inherent Hardness

- Is there a good algorithm?
- Is there *no* good algorithm?
- This is the measure of hardness that we'll use
- For various reasons, 'good' means that the function is polynomial in the size of the input
- Bad is anything bigger



Complexity Classes

- Summing up, if our function for number of steps is good (a polynomial), then the problem that our algorithm solves is easy
- The class of all easy problems is called P (for polynomial)
- In other words, this is the class of all problems for which there is an algorithm running in polynomially many steps
- Again, the number of steps is considered as a function of the input size

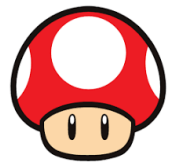


Problems!

We'll now switch gears a bit and look at some examples of problems.

Problem

Alice is playing a game in which she writes down numbers on a blackboard. She initially starts out with the number 42 on the board. On each of her turns, if the number x is on the board, she erases it, and writes either $3x$, or $\lfloor \frac{x}{2} \rfloor$. Is it possible for her to eventually write down, say, 1000?

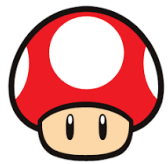


Problems! (contd.)

Problem

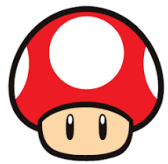
Alice is playing a game in which she writes down numbers on a blackboard. She initially starts out with the number 42 on the board. On each of her turns, if the number x is on the board, she erases it, and writes either $3x$, or $\lfloor \frac{x}{2} \rfloor$. Is it possible for her to eventually write down, say, 1000?

- If the numbers were smaller, we could try out every 'path' by hand and see if we reach 1000
- Can't do this (by hand) for bigger numbers



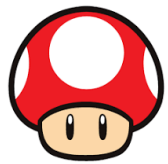
Nondeterminism

- We are very lazy and have access to computers, so we can still just bash this out
- This takes too much time though, our 'path tree' gets very big



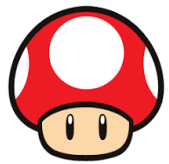
Nondeterminism (contd.)

- Another way to think about this:
- At every step, we have two choices
- So if we start at 37, we could end up at 111 or 18
- After one more step: if we were at 111, can go to 333 or 55, and if we were at 18, then we can go to 54 or 9
- Put our fingers on all of them at once and just store the list $[333, 55, 54, 9]$
- Essentially computing an entire layer of the tree at once!



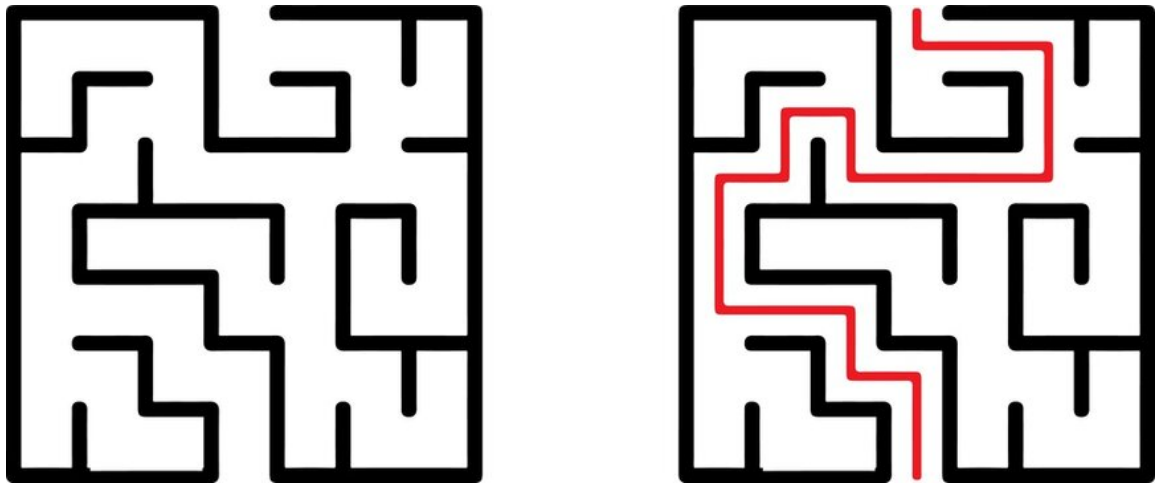
Nondeterminism (contd.)

- This problem inherently has choices
- All we care about is whether or not we get to some good state
- Such problems are well modeled by so called 'nondeterministic' programs, programs which can 'make choices' as they go along

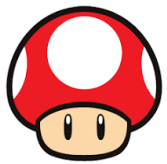


Mazes

- Another example of this is maze solving - given a maze, is there a solution?

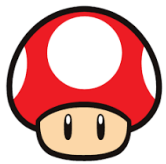


- In this case, the solution is 'L R L R R L R R L L L R'



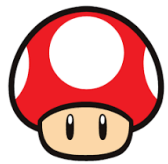
Maze Solving

- This problem again inherently has choices in it
- One way to solve this is to pick a path, explore it fully, backtrack, pick another path, and so on
- Pretty slow and boring
- We can use the same idea as the previous problem and store all nodes that we get to in one single list
- At the end all we care about is whether or not the exit was in the list



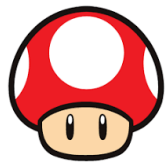
Nondeterministic Programs

- A program which, at each step, arbitrarily picks an instruction to execute, is called a nondeterministic program
- Maze solving: instructions are 'go left' and 'go right'
- Code: `while not at the exit: go left, go right`
- The program gets to pick whenever it comes to an intersection
- The program gets as lucky as it can
- If there is *some* good execution path, the program is guaranteed to take it
- Models in general the kind of situation we saw in our examples



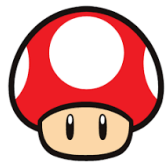
NP

- We can apply the same idea of 'difficulty' to these strange new programs
- If the number of the steps (depending on the input size) is a polynomial, then the algorithm is good
- Bad otherwise
- A problem which has a good algorithm is in the set NP (nondeterministic polynomial)



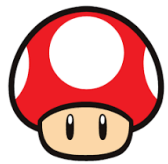
Aside: P vs NP

- The P vs NP problem asks whether or not the set P equals the set NP
- If I give you the 'lucky' execution path, then we have a regular program
- This is thought of as *verifying* a given solution
- 'Polynomial to solve' vs 'polynomial to verify'
- We'll see lots of examples of this today



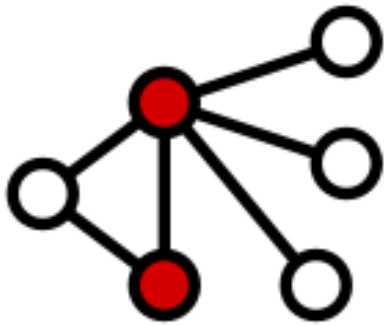
Comparisons

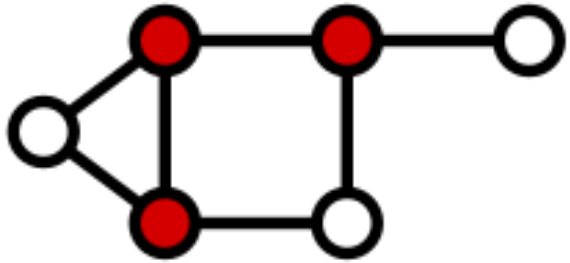
- The point of the talk:
- Problems have different difficulties, how do we compare them?
- Want to make sense of the statement 'problem A is at least as hard as problem B'
- Suppose I know how to solve problem B. If this lets me somehow solve problem A, then B *should* be easier than A
- That is, if we can reduce problem A to problem B, then A should be easier than B
- In some sense, we'd think of A as a special case of B



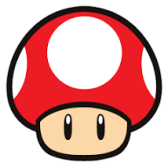
Examples!

- We'll talk about two problems, and look at this 'reduction' idea in a concrete setting
- The first problem: suppose that we have a graph. We want to check if there is a subset of the graph such that every edge is incident on a vertex in the chosen subset



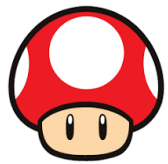


- To make the problem harder, we'll also specify how big this subset should be



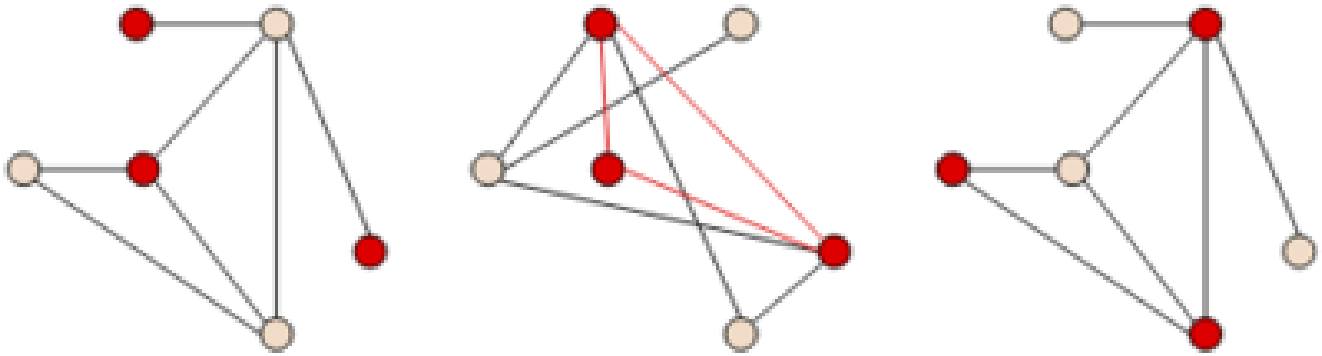
Vertex Cover

- One way to do this is to check every subset of the vertices
- Pretty slow - there are 2^V subsets to check
- If we can 'guess' the correct subset, then it's very easy to check!
- This problem is in NP
- Can think of this both as a lucky program and as 'verifying' a solution

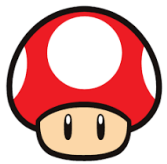


Independent Set

- Another problem: given a graph, does it have a subset such that no two nodes are adjacent?

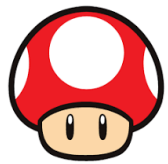


- Again, to make the problem harder, assume that we're looking for an independent set of a specified size



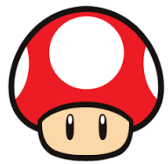
Independent Set

- Again, can check every subset of the vertices
- Slow once again
- Guessing again allows us to check very fast
- This problem is also in NP



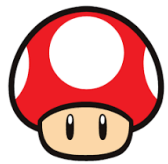
Compare

- Both of these feel similar; can we reduce one to the other?
- Pick a graph, suppose we're looking for a vertex cover of size k
- Observation: the complement of a vertex cover is an independent set!
- So if we can find independent sets, we can also find vertex covers
- In particular, given an input for vertex cover, we can *convert* it into an independent set problem
- So an algorithm solving independent set can also solve vertex cover



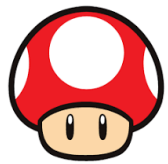
The Hardest Problems

- We have a bunch of problems in NP, and we can compare them
- What's the *hardest* problem in NP? What could this mean?



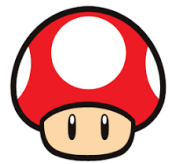
The Hardest Problems

- If A reduces to B, then B is 'stronger' than A
- So if some problem H is the hardest problem in NP, it should be stronger than *every* other problem in NP
- In our new language: every problem in NP should reduce to H
- Such problems H are called *NP-hard* (as hard as anything in NP)



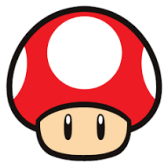
Our Final Prerequisite

- The problem that we are going to reduce to Mario is called SAT (short for satisfiability)
- Given a boolean formula in n variables, can it be made true?
- Example: the formula $(x_1 \vee x_2) \wedge (x_1 \vee \bar{x_3}) \wedge (\bar{x_2} \vee \bar{x_1} \vee x_4)$
- Need to set each variable to true or false
- This formula CAN be made true: set x_1 and x_4 to true.



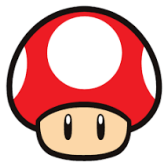
More on SAT

- This problem inherently has a lot of choice in it
- At every step, we want to make a choice between setting x_i to true or false
- Brute forcing this asks us to check 2^n cases if there were n variables
- If I tell you what the correct choices are, then checking if I'm lying or not is easy (polynomial)
- Therefore, this problem is in NP.



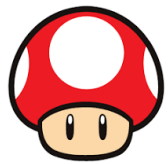
3-SAT

- A restriction on the types of formulas:
- Only take ANDs of '3-clauses'
- A 3-clause is something of the form $(x_i \vee x_j \vee x_k)$
- This is called 3-SAT



Completeness

- It turns out that 3-SAT is NP-hard (the Cook-Levin theorem)
- Again: this means that every problem in NP reduces to SAT
- Suppose 3-SAT reduces to Mario
- If a problem A in NP reduces to 3-SAT, and 3-SAT reduces to Mario, then A reduces to Mario
- Therefore Mario will be NP-hard
- (Aside: a problem which NP-hard and also in NP is called NP-complete)



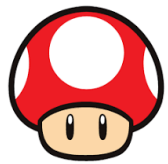
Back to Mario!

- The question we ask is the following:

Question

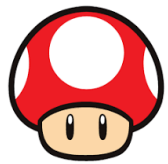
If we're given an $n \times n$ Mario level, is there a path that Mario can take from the starting point to the flag?

- The claim: this problem is NP-hard!
- Originally proven by Aloupis, Demaine, Guo, Viglietta, in 2014



Reductions from SAT

- What we want to do is make Mario look like SAT
- SAT has variables and clauses
- Each variable needs a choice
- Choices affect clauses
- To model Mario (or any problem) as 3-SAT, we have to model these in the language of our problem



General SAT Framework [ADGV '15]

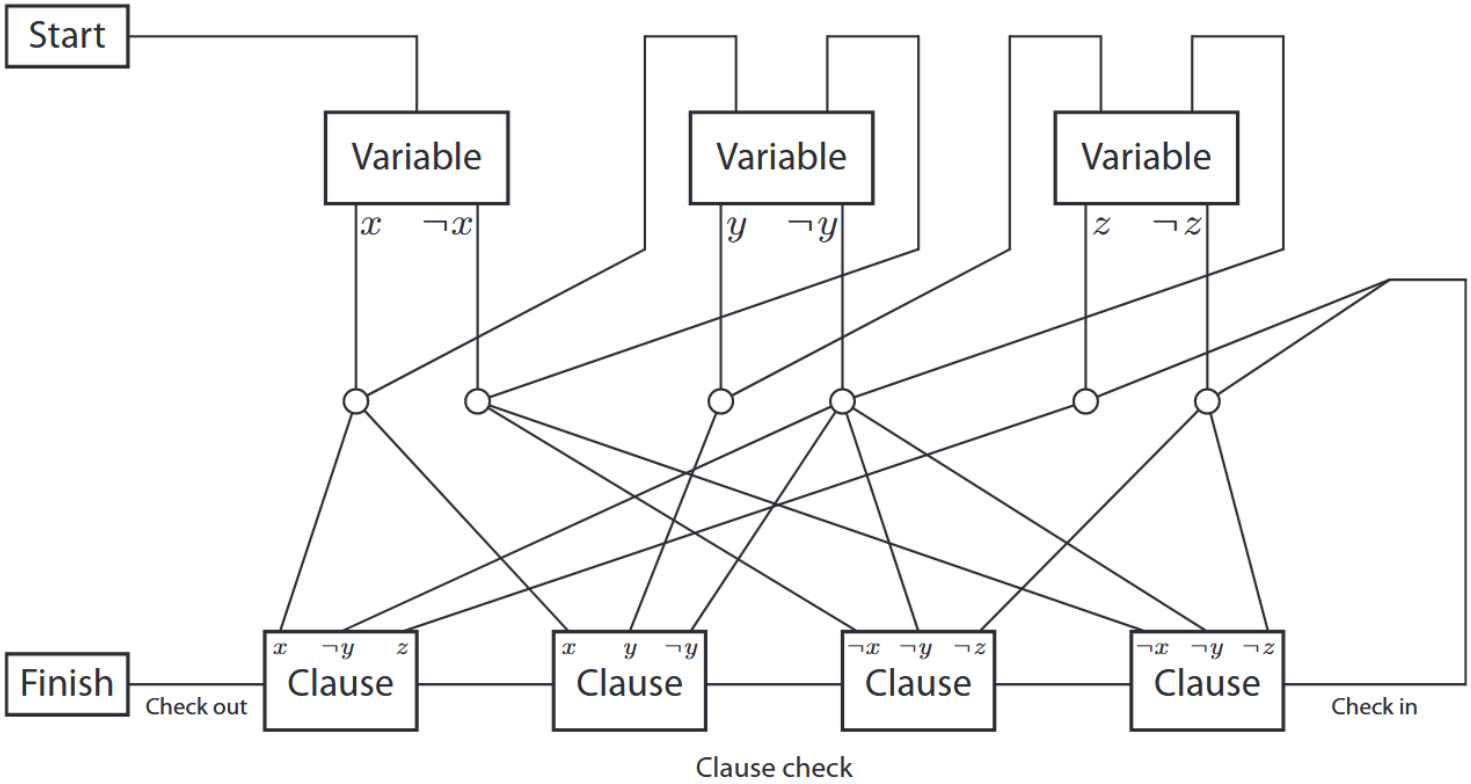
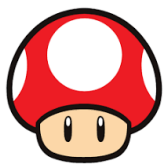


Figure 1: General framework for NP-hardness



Variable Gadgets

- Need to encode irreversible choices for each variable
- One operation for true, another for false

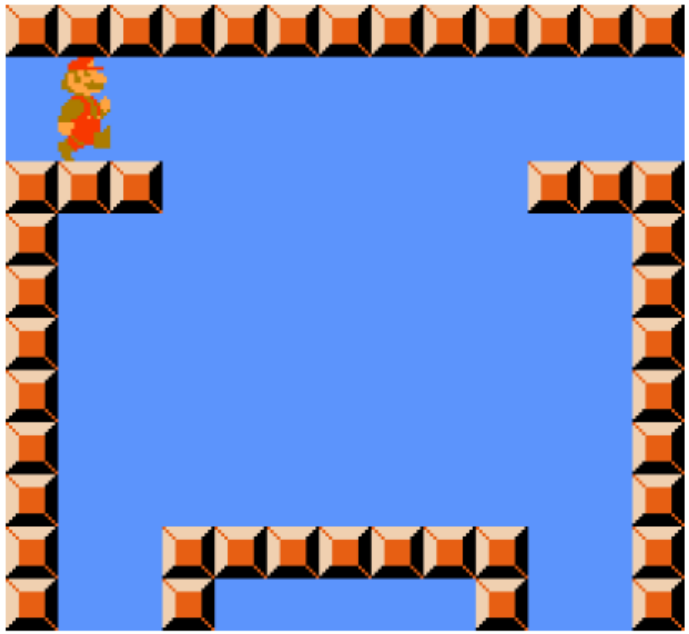
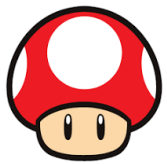


Figure 10: Variable gadget for Super Mario Bros.



Clause Gadgets

- Need to have something 'unlocked' by setting one of the clause variables to true

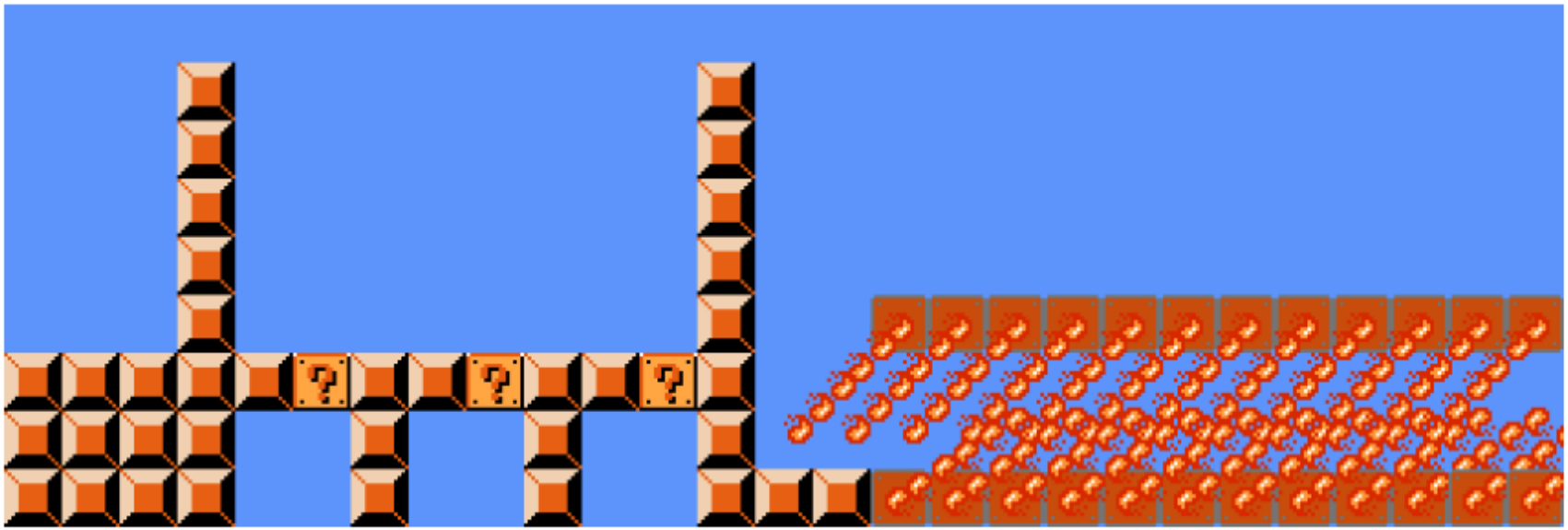
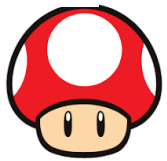


Figure 11: Clause gadget for Super Mario Bros.



Putting it all Together

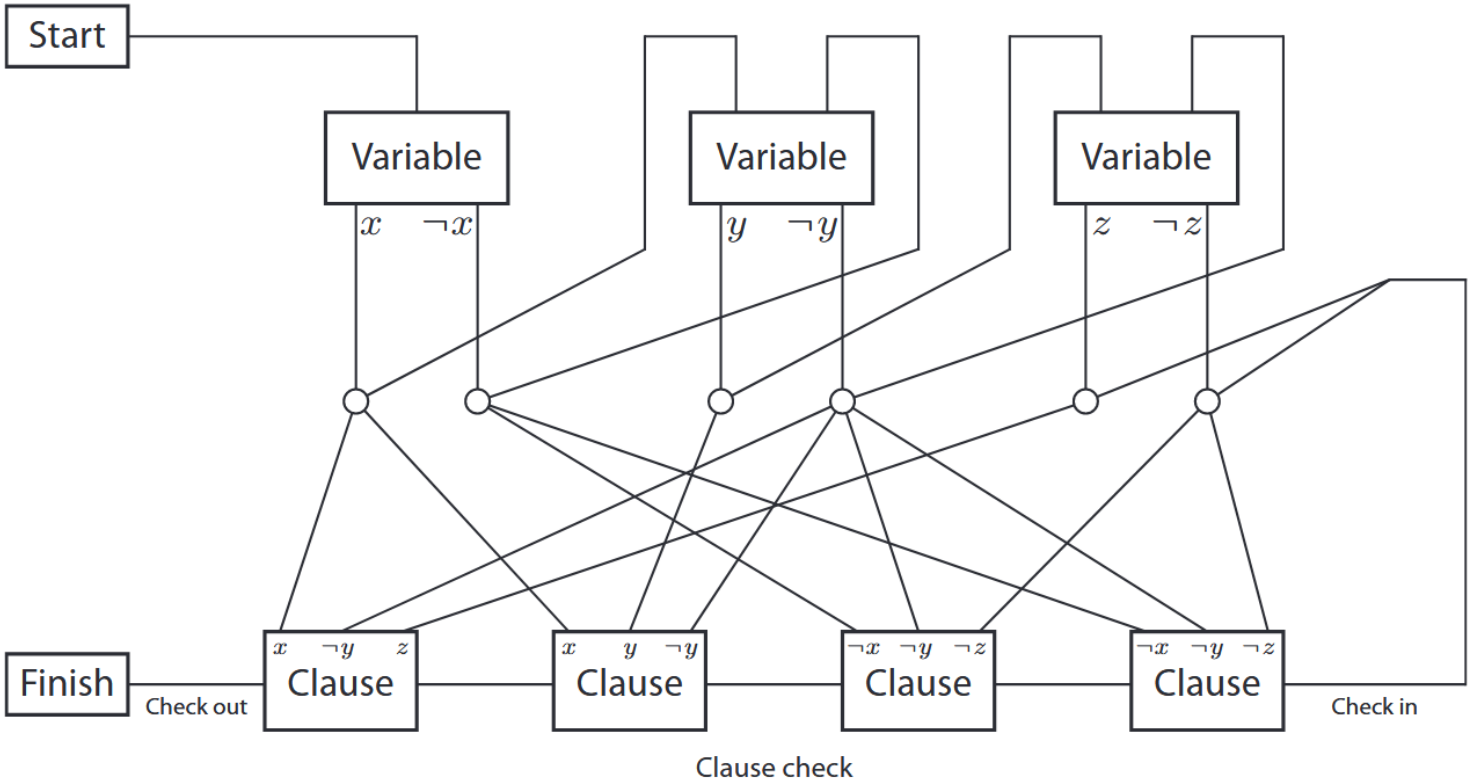
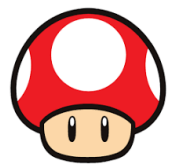


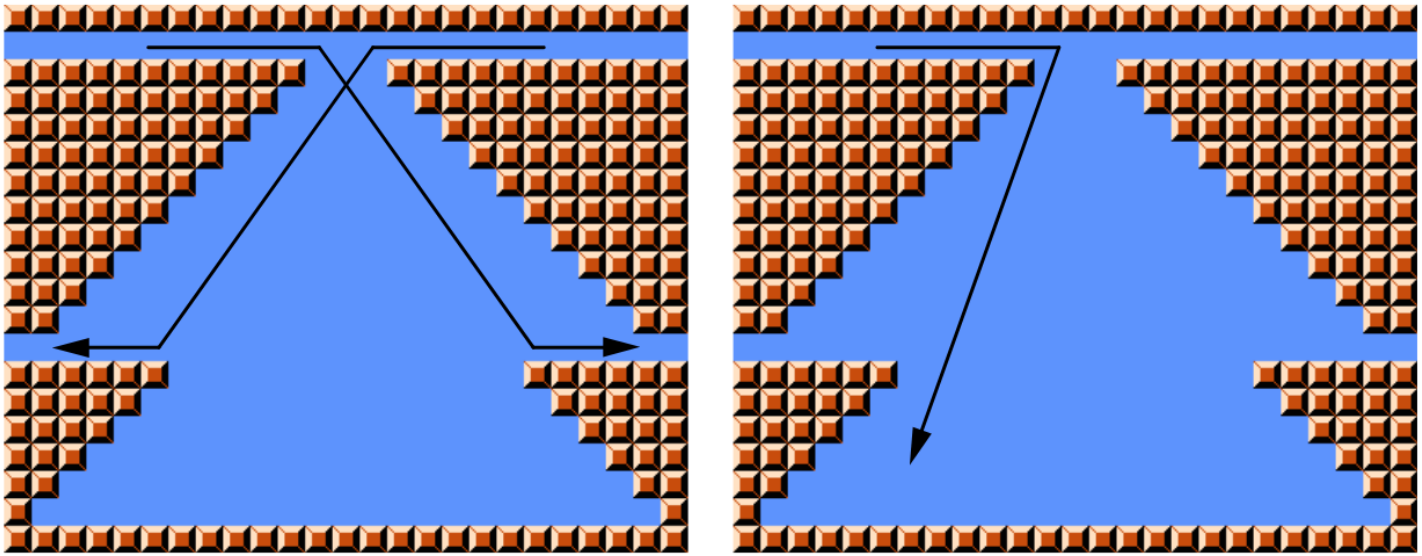
Figure 1: General framework for NP-hardness

- Aren't Mario levels planar graphs...?

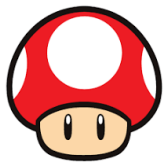


Crossovers

- We can convert a non-planar graph into a planar graph just by adding vertices at crossover points
- Need a crossover gadget purely for this purpose

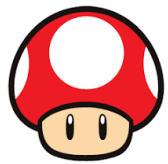


(a) (b)

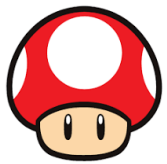


A Lot More Reductions

- A lot of other games can be proven NP-hard by very similar arguments
- In fact, a lot of classic Nintendo games (including Mario) are in PSPACE, and are actually PSPACE-complete!



Thank You!



References

- Classic Nintendo Games are (Computationally) Hard - Aloupis, Demaine, Guo, Viglietta. The reduction and pictures of the gadgets are all from this.
- The end picture is a frame from 'The Super Mario Bros. Movie'.