Playing Games For "Research Purposes"

A Combined Effort on **Games and Computational Complexity.**

Part - I

In this part, we briefly talk about computation and complexity and mostly focus on our proof that solving a generalized level in the video game CELESTE is NP-complete. Other than this, we also show that if certain changes are introduced to the game, then this problem becomes PSPACE-complete.

Modelling Computation

Computation has a lot of models, that is there are many ways to describe computation. In general you can say that computation occurs in a well defined manner, we use `Algorithms` to describe a computation.

Algorithm

An algorithm is the **set of rules** to be followed to perform a computation, these steps take us from the initial state to the desired state. The set of rules is finite and well defined without any ambiguity.

Any computation can be well described in terms of an algorithm. For example, let's define an algorithm to check if a given string is a palindrome:

1. Measure the length of the input, name it `n`.
2. Set 2 pointers `start` and `end` at the 1st and the last character respectively.
3. Check the characters under the 2 pointers, if same then go to step 4, else go to step 6
4. Increment the `start` pointer position and decrement the `end` pointer position.
5. If the distance between the 2 pointers is less than 2, Go to step 7.
6. Since the characters did not match, the given string is not a palindrome. Stop computing.
7. Since we have successfully matched all the characters, the given string is a palindrome.

Issue with such Modelling

The algorithm mentioned above has an ambiguities that are overlooked by us due to certain valid assumptions. Hence there is a need for a better and more concrete definition of what an algorithm is for a required computation.

Turing Machines and Universality

A mathematical model of description with definite rules and definitions, Turing machines were developed in 1935 to accurately describe what computation is, It was developed independently from lambda-calculus which is equivalent to Turing machines.

These abstract machines even though simplistic, have the capacity to run any computation algorithm described to it, making it equivalent to any other form of computer.

This is the **Church-Turing Thesis.**

Description of a Turing Machine

Physical Description

A Turing machine can be described physically with 3 parts, those are:

- Finite state automaton
- Infinite tape(s) with discrete cells
- head(s) for each tape.

Formal Description

Formally, A Turing Machine is described as a 7 tuple $\langle Q, \Gamma, \square, \Sigma, \delta, q_{start}, q_{halt} \rangle$.

- Q is the set of all states in the finite state automaton embedded in the Turing Machine.
- Γ is the set of symbols which are allowed on the tape.
- $\square$ is the blank symbol which occurs on the tape when it has not been accessed. $\square \in \Gamma$
- Σ is the set of symbols which can be written on the tape by the head. $\Sigma \subset \Gamma$
- δ is the transition function which decides how the machine works.
- $\delta : (Q \backslash \{q_{acc}, q_{rej}\}) \times \Gamma \to (Q \times \Gamma \times \{L, R\})$
- q_{start} is the start state of the machine.
- q_{halt} is the halt state, when a Turing Machine reaches this state it halts with the output on the tape.

Working of a Turing Machine

A Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ computes as follows. Initially, M receives its input $w = w_1 w_2 ... w_n \in \Sigma^*$ on the leftmost n squares of the tape, and the rest of the tape is blank.

The head starts on the leftmost square of the tape. The computation proceeds according to the rules described by the transition function.

Transition Function

The transition function as described earlier are functions that take the current state and the symbol below the tape's head as the input. The output is the symbol written back onto the tape, the new state of the machine, and the direction of movement of the head.

The computation continues until it enters either the accept or reject states, at which point it halts. If neither occurs, M goes on forever.

As a Turing machine computes, changes occur in the current state, the current tape contents, and the current head location. A setting of these three items is called a **configuration** of the Turing machine. The configuration can be represented as uqv, where q if the current state, the current tape contents is uv and the head location is the first symbol of v.

We say that configuration C_1 **yields** configuration C_2 if the Turing machine can legally go from C_1 to C_2 in a single step.

The **start configuration** of M on input w is the configuration q_0w, which indicates that the machine is in the start state q_0 with its head at the leftmost position on the tape.

In an **accepting configuration**, the state of the configuration is q_{accept}. In a **rejecting configuration**, the state of the configuration is q_{reject}.

A Turing machine M accepts input w if a sequence of configurations $C_1, C_2, ..., C_k$ exists, where:

1. C_1 is the start configuration of M on input w,
2. each C_i yields C_{i+1}, and
3. C_k is an accepting configuration.

The collection of strings that M accepts is the language of M, denoted as L(M). We call a language Turing-recognizable if some Turing machine recognizes it.

Church Turing Thesis

The Church-Turing thesis (formerly commonly known simply as Church's thesis) says that any real-world computation can be translated into an equivalent computation involving a Turing machine.

Robustness of the Turing Machines

The equivalency of Turing Machines with respect to the power i.e the ability to solve problems stays same for a lot of variance in the design. This property is referred to as **robustness**.

There are many variation on the definition of a Turing machine. examples being Change in Tape alphabet, Multi-tape TM, Multiple heads TM, Non-deterministic TM, etc. But they all have equivalent power, i,e. they recognize same class of languages.

To show that two kinds of machines are equivalent, we show how to simulate the behavior of one using the other as shown below.

k-tape to 1-tape

We can design a machine (A) with k tapes ,each tape with one head. Where the input will come on first tape and all the tapes can be used for computation.

The transition function of this machine will be $\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R\}^k$.

It might seems that this machine is more powerful than our original machine i.e, can recognize a greater set of language. But we can show that it is equivalent to our original 1 tape machine (A) by simulating it on that in the given way.

1. Store the information of all the k tapes on a single tape separated by $\#$.
2. Mark the location of each tape head of the k-tape machine using dotted symbol on our single tape. this could be thought of as virtual heads.
3. To simulate one move of A, B moves its head left to right from the first $\#$ to the last $\#$ to get the symbols under the dotted positions.
4. Then A makes another pass to update then according to the transition function of A.
5. If B ends up in a $\#$ it replaces it with blank symbol and shifts the tape contents one unit to the right and then continues the computation.

Complexity Zoo

> Humanity may die, but P and PSPACE are forever.

Origins of Complexity Theory

In 1936, Alan Turing developed his model of computation which formally defined what an algorithm is. The Church Turing Thesis based on valid assumptions states that any real world computation can be computed on the Turing Machine.

But Turing Machines do not tell you the amount of time or the amount of space it consumed during its runtime. A lot of computational problems were being solved with newer algorithms, which resulted in same algorithms, There had to be a **measure of performance** so that the better one could be chosen. The amount of space and time the algorithm would consume would be the parameters.

Measure of algorithms

We define an algorithms performance based on the amount of space and time is consumes to compute the answer in terms of the input size provided. The resources it consumes defines the algorithms **complexity.** More the resources more complex the algorithm is.

Space Complexity

Turing machine has tape(s) which is used to write input, store information while performing operations and the write the output. This tape is split into **blocks** in which 1 *letter* can be written.

Blocks of the Turing Machine Tape.

Although Turing Machine is said to have infinite blocks, that is not the case when we have to use it in real life. The lesser blocks an algorithm needs the better space complexity it is said to have.

Since different alphabets will give different complexity, when comparing two algorithms, we compare them on the same alphabets.

The space complexity of an algorithm is described by a function on the **input size.**

Time Complexity

The amount of time consumed by an algorithm can not be measured in terms of the actual time it computes, since it will get ambiguous when multiple machines are brought into the picture.

To avoid such confusion, the time complexity is measured by the **number of steps** the Turing Machine takes to reach the halt state.

Each step is described as a transition in the finite automaton in the core of the Turing machine accompanied by either a read/write and tape head movement.

The time complexity of an algorithm is also described as a function on the input size.

Boundedness

To compare algorithms and their complexities, the notion of boundedness was brought into the picture. How better should an algorithm must be to tell them apart in terms of complexity? For this we use the **asymptotic notation**.

Applying bounds also made some problems unsolvable. To test this, **linear bounded automaton** was described as the machine in which the amount of space grows only linearly in terms of the input size.

For the time complexity, problems(given their best solutions) would be later on classified based on a hierarchy which will be later described.

The Need for Classification

We already have the tools required to measure the efficiency of an algorithm or the difficulty of a problem, why classify?

Efficiency of Solutions

We developed fast and efficient algorithms for certain problems, such as:

- Palindrome recognition
- String matching

- Calculating the GCD

But for some problems we could never find efficient solutions, some of them being:

- Traveling salesman problem
- Boolean Satisfiability
- Independent set in a graph.

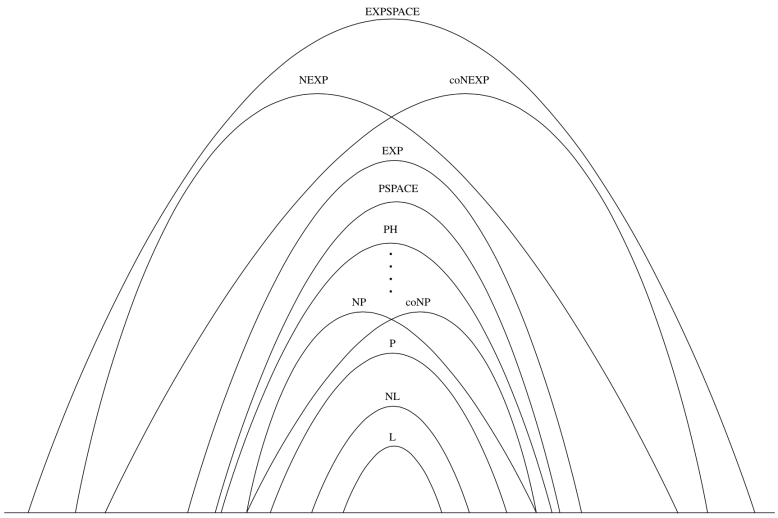
Mathematicians would generally wonder what made these questions harder than the rest, and was there something inherently common among all these problems that made them hard.

Different Models of Computations

Besides the simple single tape Turing Machine, there were other models using which there were other algorithms developed. How do they compare against each other?

To solve such issues, we classify problems and solutions based on different models together. When a new model of computation is developed, there is no need to destroy pre-existing classification and start from the scratch. Instead the new model can be added among the rest, the most recent example being the **quantum computers** getting their own class among the **hierarchy.**

Hierarchy



Hierarchy of classes from Wikipedia

Some problems by nature can be solved in lesser resources than the others, and a model with more resources can solve any problem solvable with lesser amounts of resources.

This sort of nature results into a hierarchy of models and the classification of problems. Before we get into the details of the hierarchy, lets describe the structure and the rules behind it.

Hierarchy Theorems

One of the facts leading to the hierarchy of the models and problems is that **More resources result in more computational capacity**. Resources here being Time and Space, A Turing Machine

with more resources can accomplish everything a Turing Machine with lesser resources can.

To prove the strictness of this hierarchy, we need to show that the models with more capacity can recognize more languages that the ones with lesser cant. This is accomplished by the **Time and Space Hierarchy theorem**.

Describing the terminology

Time constructible functions: A function which when given 1^n as input, will stop in exactly $f(n)$ steps.

Space constructible functions: A function which when given 1^n as input, will stop after using exactly $f(n)$ blocks on the tape.

TIME: A Turing Machine is said to be in $TIME(f(n))$ if it halts in less than $f(n)$ steps where n is the size of input. It has 2 variants, $DTIME$ which means the model has to be deterministic and $NTIME$ being nondeterministic.

Similarly we have **SPACE** with its 2 variants.

Time Hierarchy theorem

The strict hierarchy was proven for deterministic Turing Machines by Richard E. Stearns and Juris Hartmanis. One of the implications of this theorem was the strict containment of Polynomial time problems in exponential time class.

Statement:

Let $f(n)$ and $g(n)$ be 2 time constructible functions such that $g(n)log(g(n)) = O(f(n))$ Then

$$DTIME(g(n)) \subsetneq DTIME(f(n))$$

Similar result was proven by Stephen Cook for nondeterministic Turing Machines.

Space Hierarchy theorem

Similar to the Time Hierarchy theorem stating that asymptotically more space allows the model to recognize more languages than the one with the lesser space.

Statement:

Let $f(n)$ 1 space constructible Then:

$$SPACE(o(f(n)) \subsetneq SPACE(f(n))$$

Unlike time hierarchy, this theorem holds for both deterministic and nondeterministic models.

Space equivalency

For nondeterministic machines, the space consumed is considered to be the maximum space consumed among all the branches it takes for the given input.

Savitch's theorem showed that whatever a nondeterministic machine can solve in $f(n)$ space, a deterministic Turing Machine can in $(f(n))^2$.

Since the result of squaring a polynomial will still be a polynomial it implies **PSPACE = NPSPACE.**

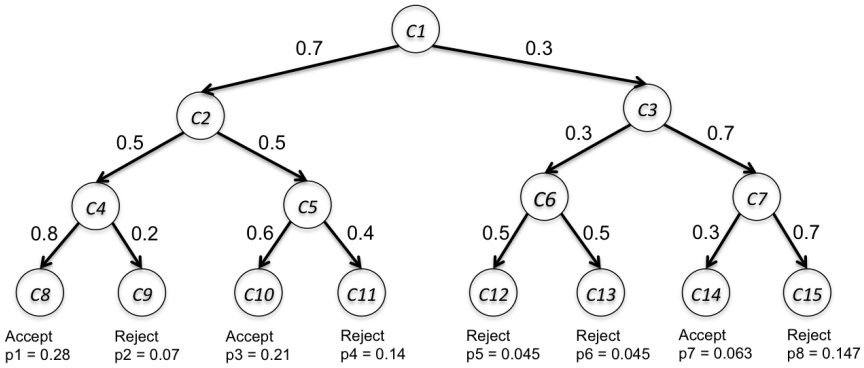
Place for other Computational Models

Besides deterministic and non-deterministic models of Turing machines, there are other computational models, those to have their own classes among the hierarchy.

Although there are plenty of them, lets look at **Probabilistic Turing Machines**

Probabilistic Turing Machine

Briefly speaking, Probabilistic Turing Machines are nondeterministic Turing Machines which choose the branch based on a probability distribution.



Probabilistic Turing Machine going to configurations based on probability

It can also be described as a Turing machine with **2 transition functions** among which it chooses one of the corresponding transitions based on the result of a coin flip(representation of a probability distribution).

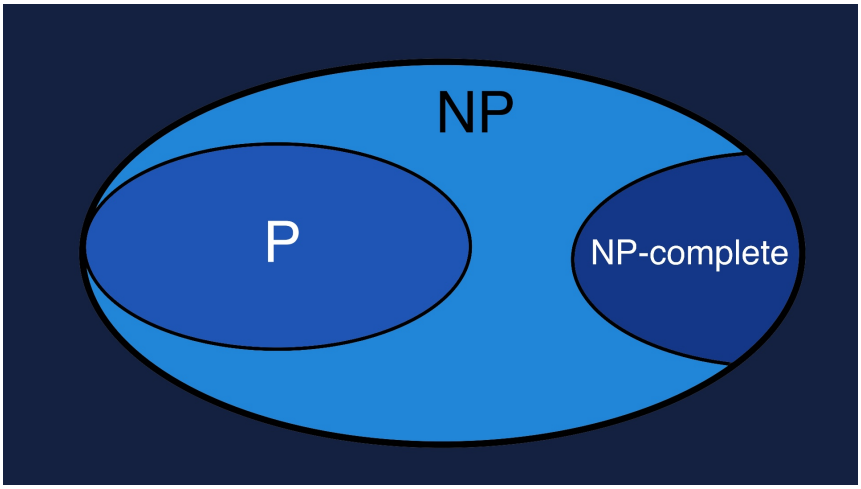
This idea of PTM came up when Solovay and Strassen made an algorithm to **primality check** with the help of coin-flipping. Although there are much better algorithms now, this was a setting stone for probabilistic computational models.

Since the models are not 100% accurate, we need to decide an upper bound of the chances that they will **fail**. This idea lead to a subclass of problems which can be solved by these model with certain accuracy in polynomial time.

Relations among the Classes

> As Grothendieck taught us, objects aren't of great importance; It is the relationship between them that are.

Since we have described the hierarchy, now we can look into some important relations these complexity classes have among them. Let's start off with 2 classes whose relation is a major unanswered question. **P and NP.**



P vs NP

An introduction of complexity zoo is not complete without a brief introduction on P vs NP. Given by Stephen Cook and now is a **Millennium prize problem**, whose solution will have consequences across all the fields. The problem is about proving the relation between the classes P and NP. is P a strict subset or is P = NP?

Deterministic Polynomial Time

Class P is the class containing problems whose solution's time complexity is a polynomial in the input size. Formally:

$$P = \bigcup_{k \in \mathbb{N}} DTIME(n^k)$$

Although practically some of them can not be solved, it is generally considered to be class of problems which can be solved efficiently.

Non Deterministic Polynomial Time

Class NP is the nondeterministic counterpart of the P. It is formally defined as:

$$NP = \bigcup_{k \in \mathbb{N}} NTIME(n^k)$$

Since the complexity taken by a nondeterministic machine to solve is equal to the complexity required to check the solution on a deterministic machine. NP can be considered as problems which have **efficient** algorithms to check the solution given a witness.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Solving a Sudoku is hard but once solved it is easy to verify

> NP problems can be thought of as **puzzles**, hard to solve, but easy to give away the answer.

Completeness

Solving one problem automatically gives us the answer for some other answer by applying some efficient conversions. One of the trivial example being converting squaring problem into multiplication, these conversions are called reductions, which will be further explained in the later sections. For now a problem is said to be complete if **all the problems in its class** can be reduced to it.

Continuing on the above definition on complete problems, we describe **NP-completeness.**

NP-complete

If any problem in NP can be reduced to a problem in polynomial time then that problem is said to be NP-complete. Cook-Levin theorem proved that 3SAT is NP complete. This was the 1st problem which was proven to be NP-complete.

Since the proof is quite lengthy, it won't be discussed here. But the intuition is provided below:

We convert the steps needed taken for the verification of the solution into intermediate configurations of the machine and check its validity using a Boolean formula. This Boolean formula can be reduced to its 3 conjunctive normal form using Boolean Algebra.

After a year of the publication of Cook-Levin's theorem, Richard Karp published a paper proving 21 problems to be NP complete. And since a lot of problems are being classified into NP-complete.

Intuitively a problem being NP complete means that it shares the resistance towards a polynomial solution as much as any other NP problem.

Having a polynomial solution to any of the NP-complete problem would give a polynomial solution to all of the NP problems.

The idea of NP-completeness and Karp's paper showed that P vs NP had implications beyond just computational problems.

There are many conjectures which when proven will result in the resolution of P vs NP problem, let us consider one of them, the Berman-Hartamanis conjecture.

Berman-Hartamanis Conjecture

The conjecture states that there is a bijective mapping between any 2 NP-complete languages which takes polynomial time to compute in any direction. This mapping is also called **p-isomorphism** (p for polynomial and isomorphism for bijection).

Since NP-complete problems have a polynomial-time reduction from all the NP problems, including the NP-complete ones, isn't this already true? No since the reduction which is used to prove NP-completeness is generally Karp's reduction, which is **polynomial many-time reduction**.

How is this related to P vs NP?

The bijection mapping implies that the size of the 2 chosen languages is same for any size of the input. That means that number of "yes" for given size of input should be same.

What happens when this becomes true?

If this conjecture were to be true then that means there can be no NP-Complete languages with linear growth with respect to the size of input. Because if this were true then we cant have P = NP, since P = NP would imply that even the languages in P where this growth is linear are NP complete, such as the strings with all 0s, which is a contradiction.

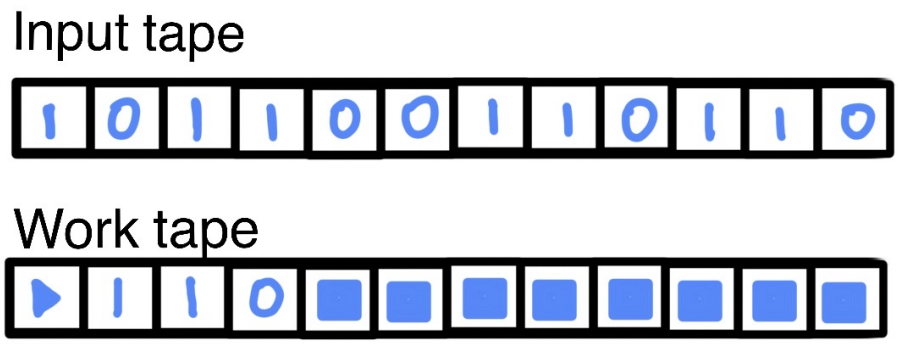
Even though P vs NP remains an unsolved problem, most of the experts already believe that P is a strict subset of NP.

> If P=NP, then the world would be a profoundly different place than we usually assume it to be. There would be no special value in “creative leaps,” no fundamental gap between solving a problem and recognizing the solution once it’s found. Everyone who could appreciate a symphony would be Mozart; everyone who could follow a step-by-step argument would be Gauss.
>
> - **Scott Aaronson**

Other Classes

Besides NP and P, there are plenty of classes in the hierarchy (545 so far), Let us have a look at how is a class defined, how it is linked among the classes, and how it fits in the hierarchy.

Log-space class



Only the work tape blocks are counted

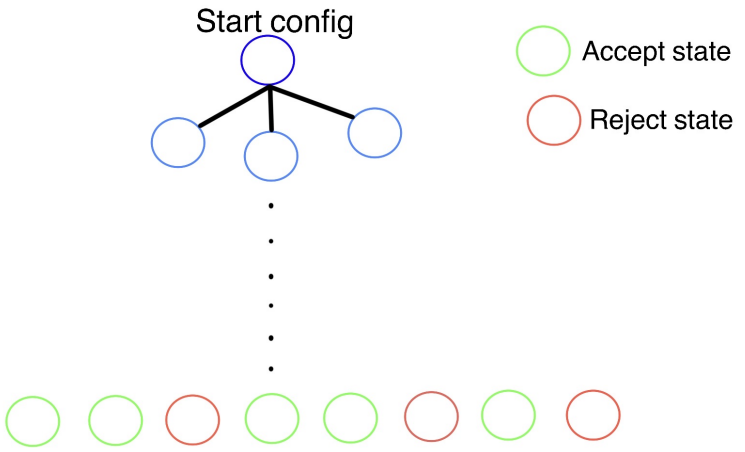
L is the class which uses log complexity of space. Since the input itself is more than this, we consider input to be written on a separate tape while considering the space complexity.

It has already been proven that Log-space is a subset of P, the proof is described below:

If a model takes $f(n)$ blocks, then the **number of configurations** it can have is $f(n)2^{O(f(n))}$.

To prevent looping, it can not come back to the same configuration. The number of configurations serve as an upper bound for the time complexity, so for log-space, the time complexity should be $O(nlog(n)) \subset \mathbf{P}$.

Counting Class: #P



Counting the number of accept states

A nondeterministic Turing Machine halts when it reaches at least one accept state. But what if we had to count the number of branches that lead to accept state for a given input? That is what **#P** is about.

Although a class usually has decision problems, that is 1 bit output. #P is a class function problems, formally:

#P is a set of functions such that, for each function there exists a nondeterministic Turing Machines **M**, with a verifier **V** which takes (x, w) as input where w is the witness, then:

$$\forall x \in \{0,1\}^* f(x) = |\{y \in \{0,1\}^{p(|x|)} : V(x,y) = 1\}|$$

It is clear that #P is as hard as NP, since a language in NP can be reduced to a language to a language in #P just by checking if the number of verifiers is greater than 0 in its #P counterpart.

More on Probabilistic Complexity

Based on the Probabilistic polynomial class described earlier, we will build other subclasses of this class, as an example of how classes are built for other models of computability.

As mentioned earlier that these models rely on probability distribution to make their decisions, and hence can not be always correct. This gives us another scale to measure their performance, **accuracy**. Based on this we have the following classes.

Bounded Error Probabilistic Polynomial Time

A class that contains languages that can be identified by a PTM with at least 2/3rd accuracy. Since the accuracy is more than 0.5, the probability of getting the correct result increases with the number of trails exponentially.

You can consider deterministic Turing Machines as a special case of BPP by setting the probability of picking one of the options as 1. Hence $P \subseteq BPP$ (equality is still an open problem).

Since quantum computing works on the probabilistic nature of qubits collapsing into one of the many superposed states, It is evident that the class of problems solvable by it (BQP) is a superset of BPP.

Since the main class has been defined, we can construct its sub-classes based on the details of how it makes decisions.

Probabilistic Algorithms

In general, an algorithm that may lead to incorrect results but always gives a result in polynomial time are known as **Monte Carlo** algorithms.

We can construct algorithms will return certain answer only if its certain about it by creating a bias towards the required side. But this comes with a trade, when the non-biased answer is returned, there is no guarantee if the answer was returned because it was found or because it was not very sure of the other answer. Creating such bias towards the true side gives us **RP.**

Randomized Polynomial Time

Output / Answer	1	0
1	$\geq 1/2$	$\leq 1/2$
0	0	1

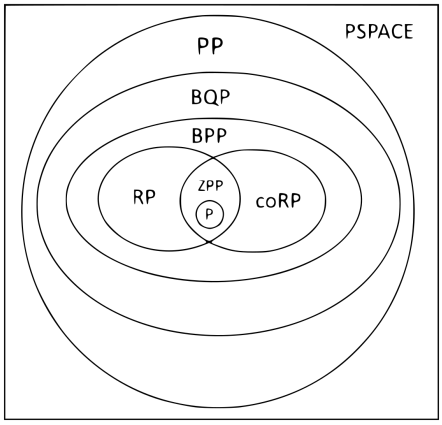
When output is yes, it is always correct.

A class of languages which can be decided by Turing Machines which return yes only if the probability of it being yes is ≥ 0.5, If the probability is ≤ 0.5, it returns a no.

Creating a Bias towards the false side gives us its counterpart **coRP**.

Now lets describe the intersection of **RP** and **coRP**.

Zero Probabilistic Polynomial



ZPP falls in the intersection of RP and coRP

Unlike **RP** and **coRP**, this class consists of languages that can be recognized by algorithms that can not give out wrong answers, since the model is still probabilistic, what separates it from a deterministic model? the computation time.

The above described algorithms are called Las Vegas algorithms, unlike other classes so far, the runtime for these algorithms is decided based on their **average runtime** and not the worst case runtime.

Constructing a Turing Machine which can run ZPP:

Since $ZPP \subset RP \cap coRP$ There exists an algorithm A in RP which decides it and an algorithm B in coRP.

We run both algorithms in parallel with alternating steps as follows:

- Run one step of A, if it returns true then stop. the answer is true, else go to next step.
- Run one step of B, if it returns false then stop. the answer is false, else repeat from above.

This guarantees that the answer is correct, naturally this means that the deterministic polynomial time algorithms (P) also fall in the same class. So, $P \subset ZPP$.

Why study probabilistic models?

A lot of problems have a much more efficient problem when randomness is brought into the picture, although this might be not fully true since a problems with better complexity on RTM are slowly being solved on DTM with the same complexity, one of the most famous one being the primality test whose deterministic solution was found in 2002 (known as he AKS primality test).

This leaves us with the question:

> Are probabilistic machines in reality more powerful than their deterministic counterparts?

Now that we understand probabilistic versions of Turing Machines, let us look at quantum computers which have the capacity to run Monte Carlo algorithms.

A Brief Introduction to Quantum Computing

In 1980, Richard Feynman Showed that Classical computers can not simulate quantum systems efficiently. He had hypothesized a quantum simulator which would not face an exponential slowdown compared to the classical computers, 5 years later David Duestch described a **universal quantum computer** taking Feynman's idea further.

Quantum Computers work on the ability of manipulating the qubits without measuring them, once they are measured, they collapse into one of the superposed with **certain probability**.

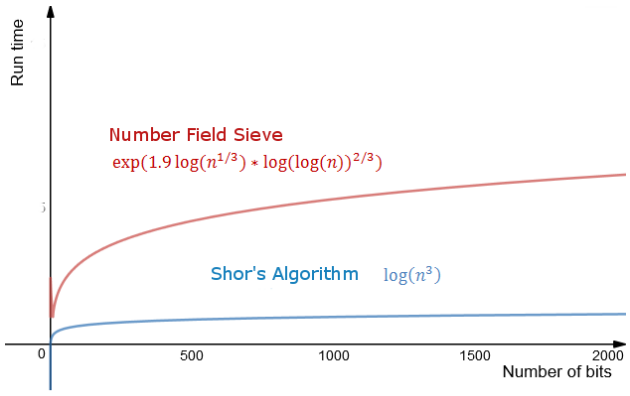
As you can notice, Since the qubit collapsing is equivalent to sampling a probability distribution, hence quantum computers have the ability to run bounded error polynomial probabilistic algorithms in polynomial time.

Now let us describe the class of problems solved by quantum computers, **BQP.**

Bounded error Quantum Polynomial Time

The quantum analogue of BPP, It is the class of languages that can be decided by a quantum computer in polynomial time with less than 1/3 probability of error.

Is quantum computer more superior to a classical computer? We don't know.



Shor's Algorithm and Grover's Algorithm are quantum algorithms that perform superior to their classical counter parts. But there has been no certain proof of strict superiority. **Shor's Algorithm** is of great importance since it factors a number in polynomial time, based on which cryptography stands.

Further Reading

What was described above is only an introduction to the complexity zoo. There are a lot of tools and ideas which haven't been covered. Below are listed some topics which were considered to be

included but rejected either due to their complexity or lack of purpose to include them.

There are a lot of classes and lots of problems in computational complexity, there is no reason to study all of them, but to know enough to explore any class when required.

Topics not covered

- Oracle Machines
- Alternating Turing Machines
- Interactive proof systems
- Probabilistic checkable proofs
- Derandomization
- Descriptive Complexity
- Finite models such as circuits

Reductions

A reduction from problem A to problem B is a polynomial-time algorithm that converts inputs to problem A into equivalent inputs to problem B. Equivalent means that both problem A and problem B must output the same YES or NO answer for the input and converted input.

Implications of Reductions

Reductions also tell us about the relative difficulty of problems. If we have a way of quickly reducing instances of A into instances of B, then solving B is theoretically at least as difficult as solving A. After all, if you can solve B, then you can solve A by using your reduction and the B solver. I believe this is why we notate the fact that A reduces to B with the notation $A < B$ or $A \leq_p B$ if the reduction is achievable in polynomial time.

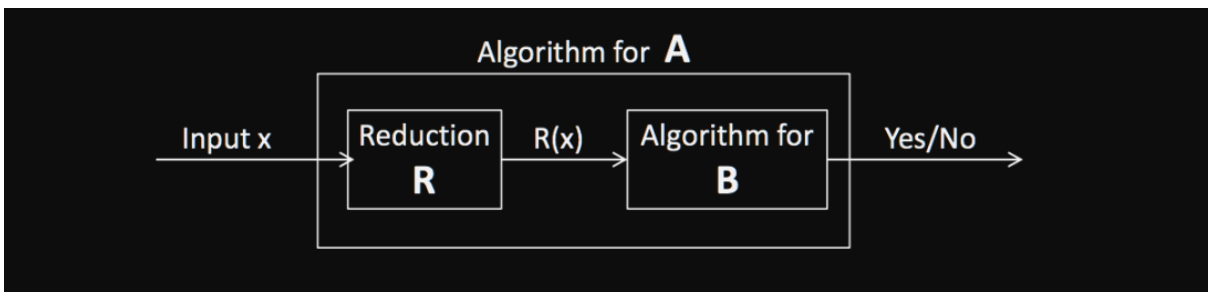
If A is NP-complete and $A \leq_p B$ then B is also NP-complete.

Karp Reduction

Let $A : X \to \{0,1\}$ and $B : Y \to \{0,1\}$ be decision problems.

A is a polytime reducible to B, or "A reduces to B" ($A \preceq B$), if there exists a function $R : X \to Y$ that transforms inputs to A into inputs to B such that $A(x) = B(R(x))$.

The following picture shows how this reduction leads to an algorithm for A which simply builds upon the algorithm for B.



Solving A is no harder than solving B. In other words, if solving B is "easy" (i.e. $B \in P$), then solving A is easy ($A \in P$). Equivalently, if A is "hard", then B is "hard". Given an algorithm for B, we can easily construct an algorithm for A.

How to prove problems are NP-complete?

We can show that problems are NP-complete via the following steps.

1. Show $X \in NP$. Show that $X \in NP$ by finding a nondeterministic algorithm, or giving a valid polynomial verifier for a certificate.
2. Show X is NP-hard. Reduce from a known NP-complete problem Y to X.

This is sufficient because $\forall Z \in NP$ can be reduced to Y, and the reduction demonstrates inputs to Y can be modified to become inputs to X, which implies X is NP-hard.

We must demonstrate the following properties for a complete reduction.

Give an polynomial-time conversion from Y inputs to X inputs.

- If Y's answer is YES, then X's answer is YES.
- If X's answer is YES, then Y's answer is YES.

Following is an example problem.

Reducing Hamiltonian Cycle to Hamiltonian Path

Hamiltonian Path: Given a directed graph G = (V, E), is there a path that visits every vertex exactly once?

Given that Hamiltonian Cycle is NP-Complete, we prove that Hamiltonian Path is NP-Complete.

1. **Show Hamiltonian Path $\in$ NP**
   To prove this, we need to prove that there exists a verifier $V(x, y)$. Let $x = G$ be a "yes" input. Let y be a path P that satisfies the condition.
   We can verify that the path traverses every vertex exactly once, then check the path to ensure that every edge in the path is an edge in the graph. Naively, it takes O(n^2) to check that every vertex is traversed exactly once and O(nm) to check that every edge in the path is in the graph.
2. **Show Hamiltonian Path $\in$ NP-Hard.**
   We prove this by giving a Karp-reduction of Hamiltonian Cycle to Hamiltonian Path.
   - Given an instance of Hamiltonian Cycle G, choose an arbitrary node v and split it into two nodes v' and v''. All directed edges into *v* now have *v'* as an endpoint, and all edges leaving v leave v'' instead. We call this new graph G''. The transformation takes at most $O(E)$.
   - If there is a Hamiltonian Cycle in G, there is a Hamiltonian Path on G''. We can use the edges of the cycle on G as the path on G'' but our path must begin on v'' and end on v'.
   - If there is a Hamiltonian Path on on G'', there is a Hamiltonian Cycle on G. The path must begin at v'' and end at v', since there are no edges into v'' or out of v'. Thus we can use the path on G'' as a cycle on G once v' and v'' are remerged.

3. This proves Hamiltonian Cycle reduces to Hamiltonian Path in polynomial time, which means that Hamiltonian Path is at least as hard as Hamiltonian Cycle, so Hamiltonian Path is NP-Complete.

Satisfiability Problem (SAT)

Basic definitions

- *literal* : propositional variable or the negation of a propositional variable.
- *clause* : dis-junction (or) of one or more literals.
- Boolean *formula* : is a conjunction (and) of one or more clauses.

$$\text{Boolean Formula}(\phi) : (\neg x \vee y) \wedge (\neg y \vee z) \wedge (x \vee \neg z \vee y)$$

Satisfiability

A formula ϕ is satisfiable if there exists an assignment of values to its variables that makes ϕ.

The satisfiability problem, usually called SAT, is the following language:

SAT = $\{\phi \mid \phi$ is a satisfiable clausal formula$\}$.

Thought of as a computational problem, the input to SAT is a formula ϕ and the problem is to determine whether ϕ is satisfiable.

Cook - Levin Theorem

Theorem : Determining if a Boolean formula ϕ is satisfiable or not is an NP-Complete problem.

Cook and Levin independently proved that SAT is NP-complete. Both did so before the term NP-complete was even in use. Their work led to the study of NP.

Proof

Main Ideas

1. SAT $\in$ NP since given a truth assignment for $x_1, \ldots x_n$, you can check if $\phi(x1 \ldots, xn) = 1$ in polynomial time by evaluating the formula on a given assignment.
2. We now need to show that there is a polynomial-time reduction for every A in NP.
3. $A \in$ NP means that there is a non-deterministic Turing machine **N** running in O(n^k) time that decides A. We will construct a Boolean formula ϕ that is satisfiable if and only if some branch of N's computation accepts a given input w.
4. A tableau for non-deterministic TM (N) is a table listing its configurations on some branch of its computation tree.
    1. So determining if $w \in A$ is equivalent to whether or not there is a tableau using encoding an accepting computation of N on input w.

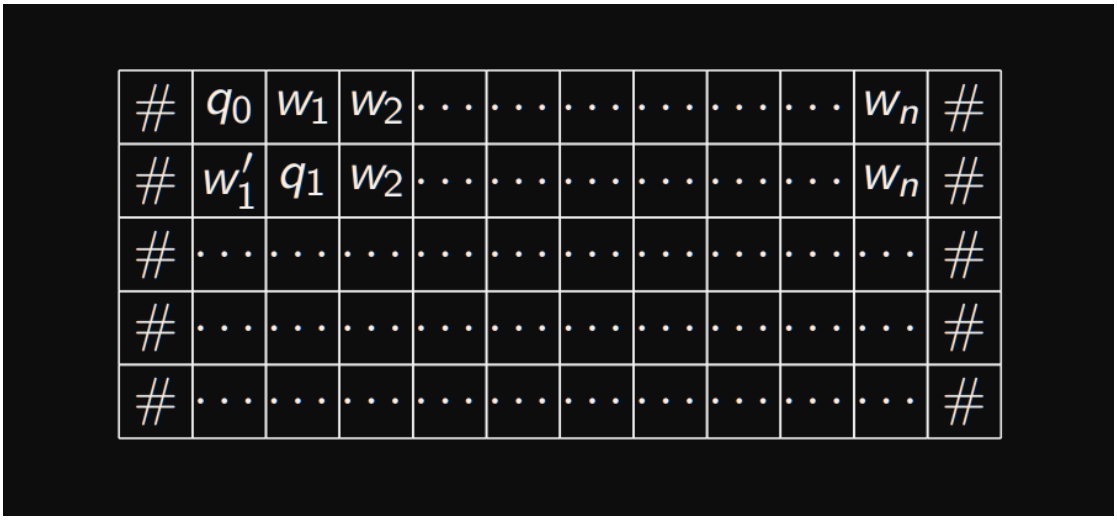


Figure : Part of a Tableau

Encoding The tableau as a Formula

- Each entry of a tableau T can be a state q_i of the TM (Q), an element of the tape alphabet Γ or $\#$. Let $C = Q \cup \Gamma \cup \{\#\}$. We define a propositional variable $x_{(x,j,s)}$ for every cell in row i, column j, and element $s \in C$.
- We interpret $x_{(x,j,s)}$ as true iff $T[i,j] = s$.
- N accepts w iff:
  1. Each cell is well-defined
  2. The first row is an initial configuration with w as the input.
  3. Each row follows from the previous row using the transition function given by N.
  4. Some row has a cell that includes an accepting state q_{accept}.

  We can express each of these conditions using propositional logic in the variables $x_{(x,j,s)}$.

Condition 1: Well defined Tableau

- A well-defined tableau means that every cell $T[i,j]$ in the tableau is filled with exactly one element (possibly the blank symbol).
- In propositional logic cell $T[i,j]$ being filled with exactly one element is equivalent to the proposition.

$$\phi_{i,j} = \left(\vee_{(s \epsilon C)} x_{i,j,s}\right) \wedge \left(\vee_{(s,t \epsilon C, s \neq t)} \left(\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}}\right)\right)$$

- We have a well-defined tableau if $\phi_{cell} = \wedge_{(i,j)} \phi_{i,j}$ is true.

Condition 2: Initial Configuration

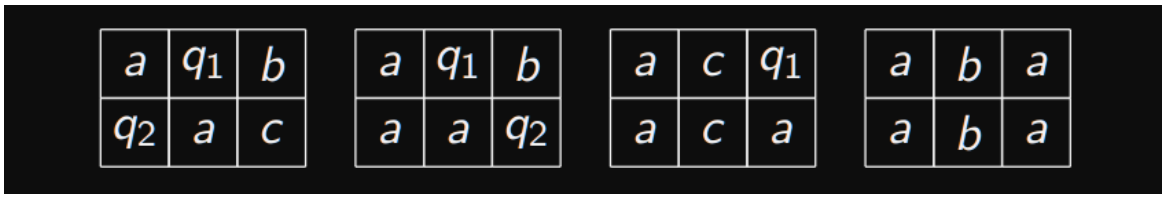
$$\phi_{start} = x_{1,1,\#} \wedge x_{1,2,q_0} \wedge x_{1,3,w_1} \wedge x_{1,4,w_2} \wedge ... x_{1,n+2,w_n} \wedge x_{1,n+3,\sqcup}, .. \wedge x_{1,O(n^k)-1,\sqcup} ... x_{1,O(n^k),\#}$$

is true iff $w = w_1 \ldots w_n$ is given as input.

Condition 3: Valid Transitions

- A window in the tableau is a 2×3 piece with adjacent rows and columns.
- A window is legal if it does not violate transition function of N. Determining which windows are legal can be done by case analysis.

  Example: suppose the TM N has a transition function $\delta(q_1, b) = \{(q_2, c, L), (q_2, a, R)\}$ then the following are legal windows for N's computation.



- **Observation 1**: Each row in the tableau is a configuration following the previous row according to N if and only if each window in the tableau is legal.
- **Observation 2** : The number of legal windows is finite (≤ $|C|^4$).
- Hence the condition that each row follows from the previous according to N can be expressed as the condition:

$$\phi_{move} = \wedge_{(1<=i,j<O(n^k))} \phi_{window,i,j}$$

Condition 4: Accepting Configuration

where $\phi_{window,i,j}$ expresses the condition that the window with cells $(a_1, \ldots, a_6)$ with top middle cell at (i, j) is legal.

The tableau is accepting iff some cell in the tableau contains an accepting state $\phi_{accept} = \vee_{(i,j)} x_{i,j,q_{accept}}$ iff the tableau is accepting.

Putting it all together

- Given a non-deterministic Turing machine N and some input w we have shown that there is a propositional formula ϕ defined by

$$\phi_{N,w} = \phi_{cell} \wedge \phi_{move} \wedge \phi_{start} \wedge \phi_{accept}$$

- that is satisfiable if and only N accepts w.
- The sub-formulas encode the 4 conditions needed there be an accepting tableau for the computation of N on input w.
- It remains to show that the reduction is computable in polynomial time.

Polynomial Time reduction :

- We assumed that the N runs in O(n^k) time on inputs of length n so the tableau has O(n^k) rows and O(n^k) columns.
- The formula constructed by the reduction has O(n^{2k}) literals,since there is a constant size formula for each cell of the tableau.

- The formula for each cell can be generated efficiently from a description of ND-TM N.
- All together this gives a reduction with runtime poly(n).
- This completes the reduction $A \leq$ SAT. We can produce a formula $\phi_{N,w}$ in polynomial time that, which is satisfiable iff $w \in A$.

Consequences

The proof shows that any problem in NP can be reduced in polynomial time (in fact, logarithmic space suffices) to an instance of the Boolean satisfiability problem. This means that if the Boolean satisfiability problem could be solved in polynomial time by a deterministic Turing machine, then all problems in NP could be solved in polynomial time, and so the complexity class NP would be equal to the complexity class P.

Karp's 21 Problems

- The significance of NP-completeness was made clear by the publication in 1972 of Richard Karp's landmark paper, "Reducibility among combinatorial problems." He showed that **21 diverse combinatorial and graph theoretical problems, each infamous for its intractability, are NP-complete.**

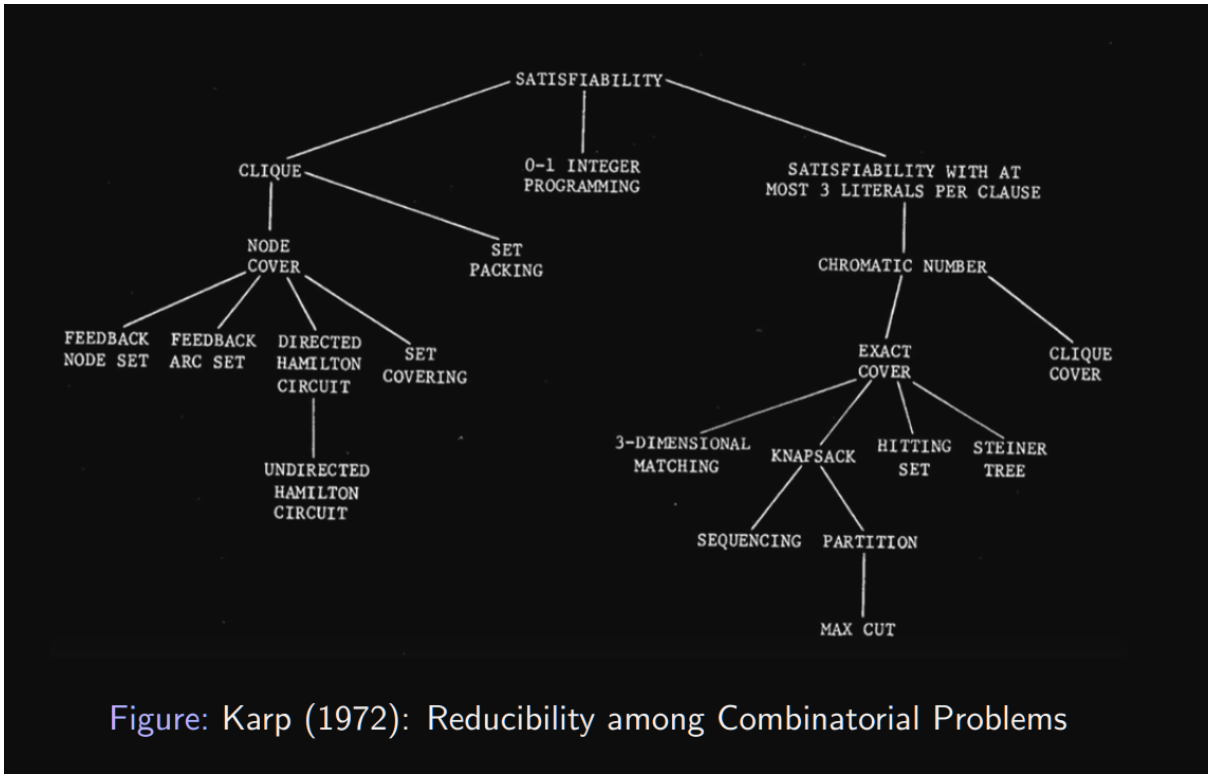


Figure: Karp (1972): Reducibility among Combinatorial Problems

- Karp showed each of his problems to be NP-complete by reducing another problem (already shown to be NP-complete) to that problem.
- For example, he showed the problem 3SAT (the Boolean satisfiability problem for expressions in conjunctive normal form with exactly three variables or negations of variables per clause) to be NP-complete by showing how to reduce (in polynomial time) any instance of SAT to an equivalent instance of 3SAT.

- First you modify the proof of the Cook–Levin theorem, so that the resulting formula is in conjunctive normal form, then you introduce new variables to split clauses with more than 3 atoms.
- For example, the clause $(A \vee B \vee C \vee D)$ can be replaced by the conjunction of clauses $(A \vee B \vee Z) \wedge (\neg Z \vee C \vee D)$, where Z is a new variable which will not be used anywhere else in the expression.
- Clauses with fewer than 3 atoms can be padded; for example, A can be replaced by $(A \vee A \vee A)$, and $(A \vee B)$ can be replaced by $(A \vee B \vee B)$.

Now that we understand Complexity classes and reductions, we prove Celeste to be NP-complete by reducing 3SAT into the Celeste.

Classifying Celeste

About Celeste

Celeste is a single-player 2D platformer developed by Maddy Thorson and Noel Berry. The game is about you being Madeline who is climbing to the peak of the mountain "Celeste". The goal of the game is to overcome obstacles on the way and make it to the end of the level.

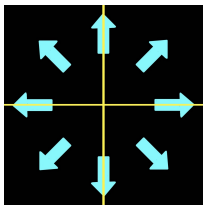
The game consists of 7 levels which consist of subparts. In each subpart, you have to reach an exit point without taking damage. Taking damage resets you back to the starting point.

About the Player

Madeline is the main character whose actions are governed by our controls. She is restricted in **8 directions of motion** and primarily has 3 special moves other than

basic left and right movement. those are:

Directions of movement



Madeline can move in 8 directions as described in the left diagram. These and the other moves have fixed button/button combinations assigned to them.

Jump

Allows her to go over gaps.

She has the ability to jump to a certain height, which can be then done again only when she hits the ground.

Dash

dashing gave her extra momentum to make to the platform.

When Madeline has the "charge" she can dash in any direction, this gives her extra momentum and the ability to dash into "space blocks" (described later). The charge gets used up when she dashes and is restored when she hits the ground or passes through the space block. (There are other objects which also recharge her, but those are not used in the proof)

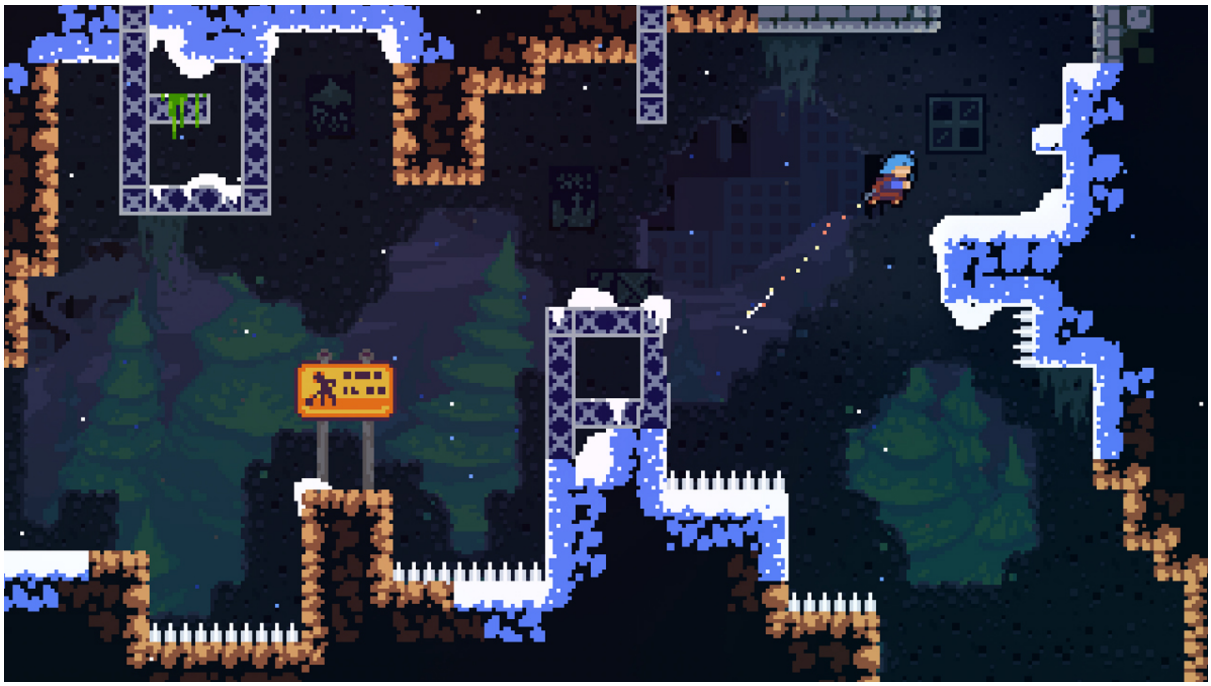
Grab

She is able to climb walls with this ability.

Since she is a mountain climber, she has the ability to climb walls and wedges. But she has stamina, which limits the amount of time she can grab onto a wall before sliding down. This stamina is restored when she makes contact with the ground

Level Implementation

Frames



bottom left corner is the entry point and the top right corner is the exit point.

What are frames?

Frames are areas of games which has a start and an endpoint. You have access to one frame at a time. Using the entries and the exits, you move from 1 frame to another. So Frames serve as the checkpoints in the game. Frames do not have a limit of size since your screen can scroll.

The game has been split into such frames, which are puzzles on their own, which require planning and reflexes to reach the endpoint. There are multiple ways to solve these puzzles due to the game's versatility and the mechanisms in it.

A graph of such frames connected together makes up a level of the game. To construct our proof, we will make frames which we will join together to make our level.

Objects

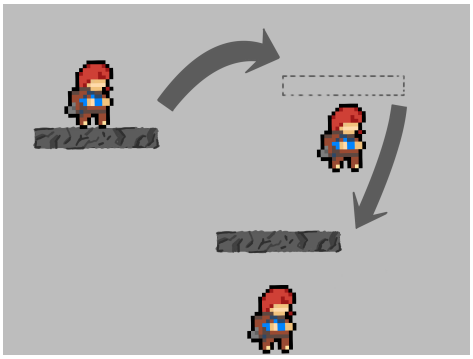
Revolving around the basic 3 operations the game has a lot of mechanisms that add to the fun and the difficulty of the game. These are added to the game as the player makes progress in the levels.

For our proof, we will mainly use 3 of the objects. They are:

- Unstable Platform
- Button door
- Space block

The purpose of these objects will be explained later.

Unstable platform



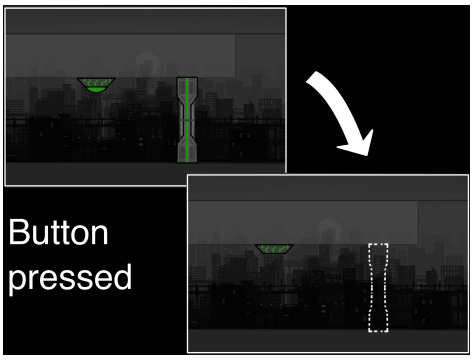
Forms back at the same position as before.

In slow motion

A stone platform that can float, this platform breaks when Madeline stands on this platform for more than a second. After the platform breaks, Madeline if not jumped will fall down.

This platform then forms back in the same place. But it can only be broken when stood upon and can not be broken from below, making it like a trap door if placed correctly.

Button door



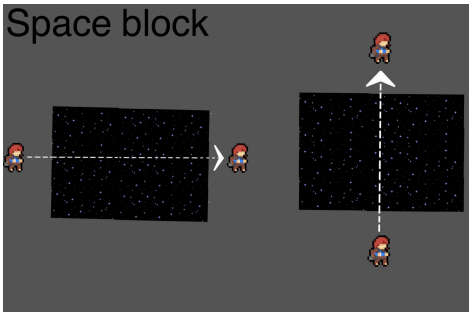
Multiple doors in a single frame are allowed.
But one button per door.

Visual representation

A door that can only be unlocked using its specific button. This button must be placed in the same frame as the door, but it has the freedom to be placed anywhere in the frame.

Once the door is opened it can not be closed again.

Space Block



Space blocks!

Visually pleasing space blocks

A block of celestial material which lets you float into and out in a straight line when you dash into it. Once Madeline dashes into the block, you can not stop her from reaching the other opening of the block in a straight line.

If the other side of the line is blocked with a wall, Madeline dies and respawns at the start of the frame.

Level put together

A level consists of many frames, but there is only 1 flag at the top of the level and there is only 1 initial start point of the level. For most of the levels, there is only 1 linear path from the start to end with a sequence of frames, but some other levels are more complex, which include sub-tasks and other detours.

Here is an example, this is the 1st level in the game, the frames have been arranged according to the order.

Bottom left being the start and the top right being the end.

Complexity Classification

Now that the game has been well defined, we work on classification of the game. Classification has been discussed broadly under the Complexity zoo section. Now we adapt one of the methods.

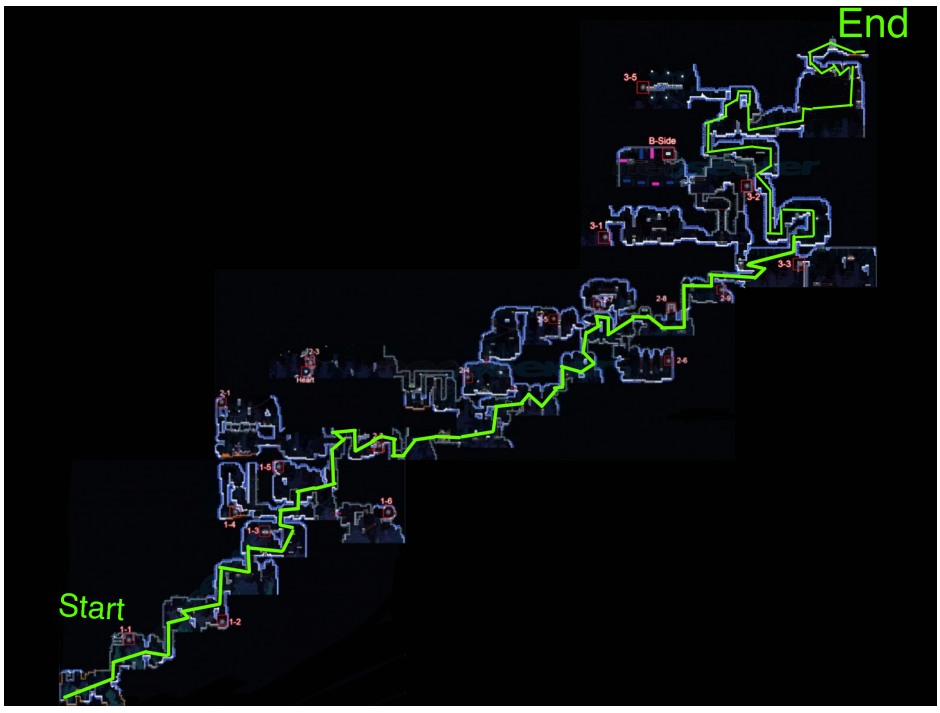
Whenever we say "Celeste belongs to X complexity class", we mean to say that the decision problem of deciding whether finish point is reachable from start point.

Basic Observation

Given a level and a path, that is the moves required to reach the endpoint, You can verify if the path is correct just by applying those moves.

The moves are polynomial, why? We repeat the sections only after visiting other sections. Since the number of sections themselves are polynomial, we can only have polynomial moves before we complete the level.

This clearly implies that the game is NP. For example, for the 1st level, we can map the path from start to end as seen below.



In this pattern we can always map from start to end.

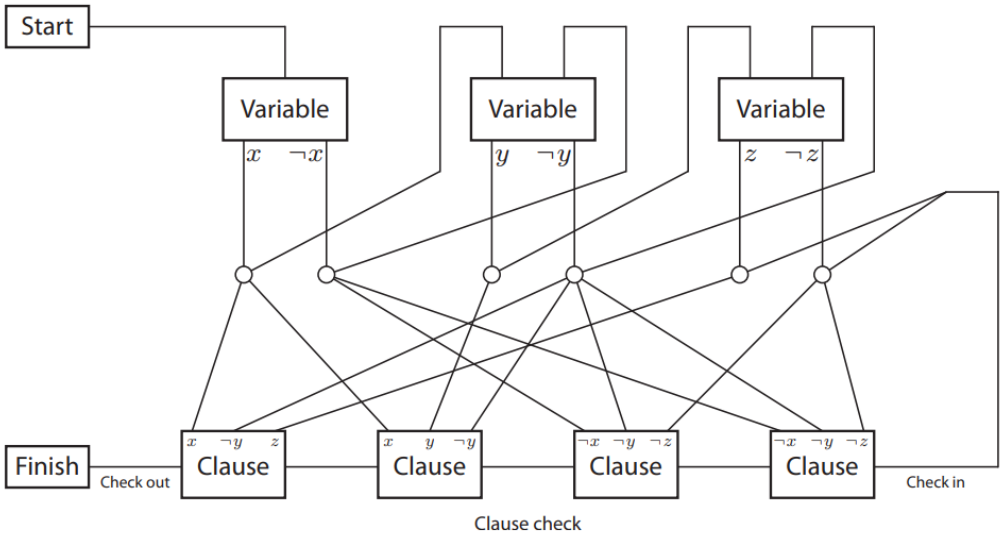
Now since we have proven that Celeste is NP. We can try to prove that it is NP-Hard, essentially proving that it is NP-Complete.

Due to many paths of the game which do not lead to the end, and certain mechanisms that lock us out which we will see in the future, We can not immediately tell that there exists a polynomial-time algorithm to solve the levels.

So we will try proving that this game is NP Hard.

Framework for NP-Hardness

To prove the NP-hardness of Celeste, we here describe a framework for reducing 3SAT to a 2-D platform game. This framework is based on (link source here). Using this framework in hand, we can prove the hardness of games by just constructing the necessary gadgets.



General Framework for NP-Hardness

The framework is a reduction of the 3SAT problem. We start from the Start gadget, and end at the Finish Gadget. At each Variable gadget, we make a choice and turn on the Clause gadgets according to the choice, which in essence is making a literal true or false.

At the end, we make a pass through all of the clauses sequentially and we are able to pass through them if all of them are satisfied. Since the game is 2D, we also need a Crossover gadget. The Crossover gadget makes sure that if there are two overlapping connections, we travel the connections one by one. Here are more details for the gadgets:

Start and Finish

The start and end gadgets contain the spawn point and the end goal respectively.

Variable

Each variable gadget must force the player to make a binary choice (select x or ~x). Once a choice is taken the other choice should not be accessible. Each variable gadget should be accessible from and only from the previous variable gadget is such a way that it is independent of the choice of the previous gadget and going back is not allowed.

Clause

Each literal in the clause must be connected to the corresponding variable. Also, when the player visits the clause, there should be a way to unlock the corresponding clause.

Check

After all the variables are passed through, all the clauses are run through sequentially. If the clause is unlocked, then the player moves on to the next clause else loses.

Crossover

The crossover gadget allows passage via two pathways that cross each other. The passage must be such that there is no leakage among them.

> If we can build these gadgets using a game, we can reduce 3SAT to that game using this framework and show that the game is NP-Hard.

Celeste is NP-Hard

To prove that Celeste is NP-hard, we will try to use the above framework to reduce 3SAT to Celeste.

The Variable Gadget

A Boolean Variable can take 2 values, True or False, and It might have multiple occurrences throughout the formula.

The verification of 3SAT is done by giving the satisfiable values to the variables, hence the values can not be changed in the middle of the substitution.

For now, we need to take care of the binary and the irreversible nature of boolean variables. We do this with the help of an **Unstable Platform**.

Exits are covered by Unstable platforms making them one way traps.

Gadget description

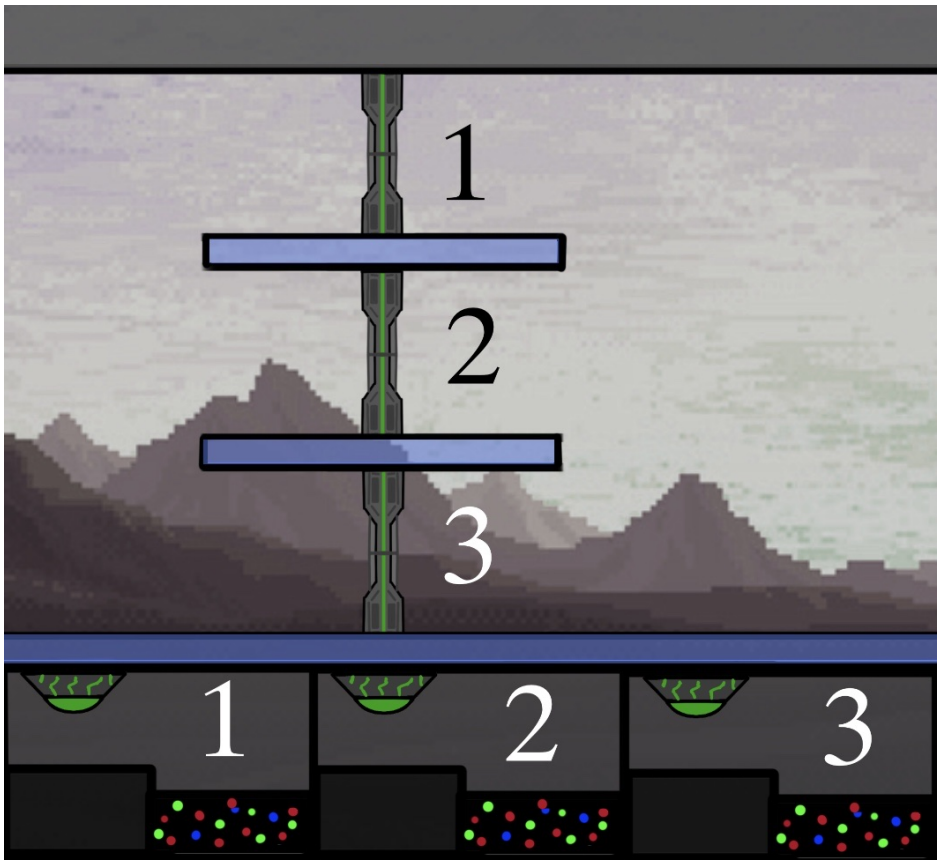
These exits have an Unstable platform covering them, these have to be broken before the exit can be used. Why the unstable platform?

Madeline falls from the top on the platform, that is the only entry to the Gadget. The Gadget has 2 exits on the sides of the floor, each leading to a tunnel.

The unstable platform makes Madeline seal her choice. Once the path is taken, there is no way to access this frame again other than restarting since the platform will reform blocking the entry.

Clause Gadget

Each Clause has 3 variables, out of which even if 1 were true the Clause would be true. To implement that in our frame, we use **The Button Door**.



The colorful dots represent space block and the hits are reachable by Madeline.

Gadget description

A Clause gadget consists of 3 Parallel Button Doors, the buttons are accessed through the variable tunnels. For now, do not worry about how the tunnels are connected.

The main idea is that even if 1 door opens it is sufficient for Madeline to pass through the region.

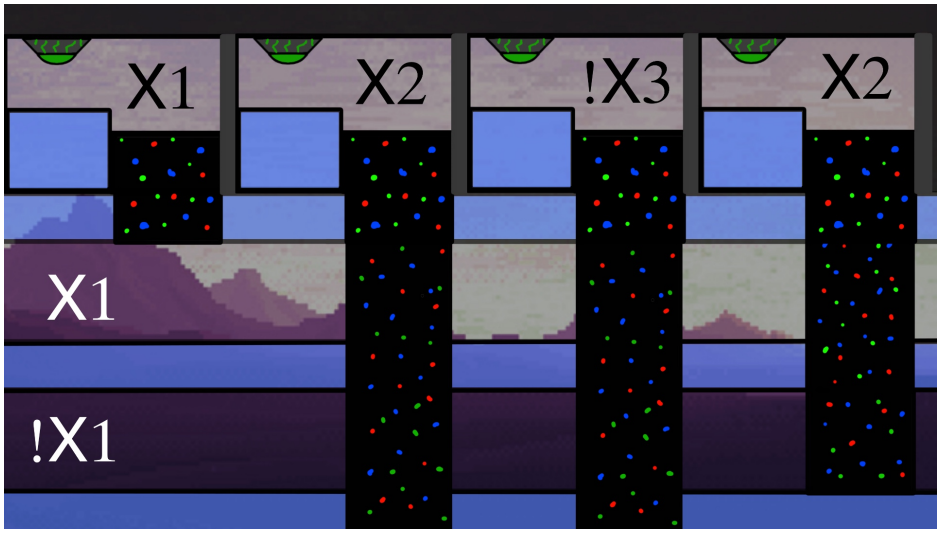
Madeline presses the buttons according to the values she took for the variables, these will unlock the doors, if the variables made a clause true, the clause would have at least 1 door open.

Support Gadgets

Now the Above gadgets must be connected, for that, we use our support gadgets that will be constructed as per the requirements.

The Tunnel

To connect the Variable exit to the Buttons of the Clause, we use a Tunnel gadget.



x1 Tunnel only has access to button which have x1 as their variable in their clause.

Gadget description

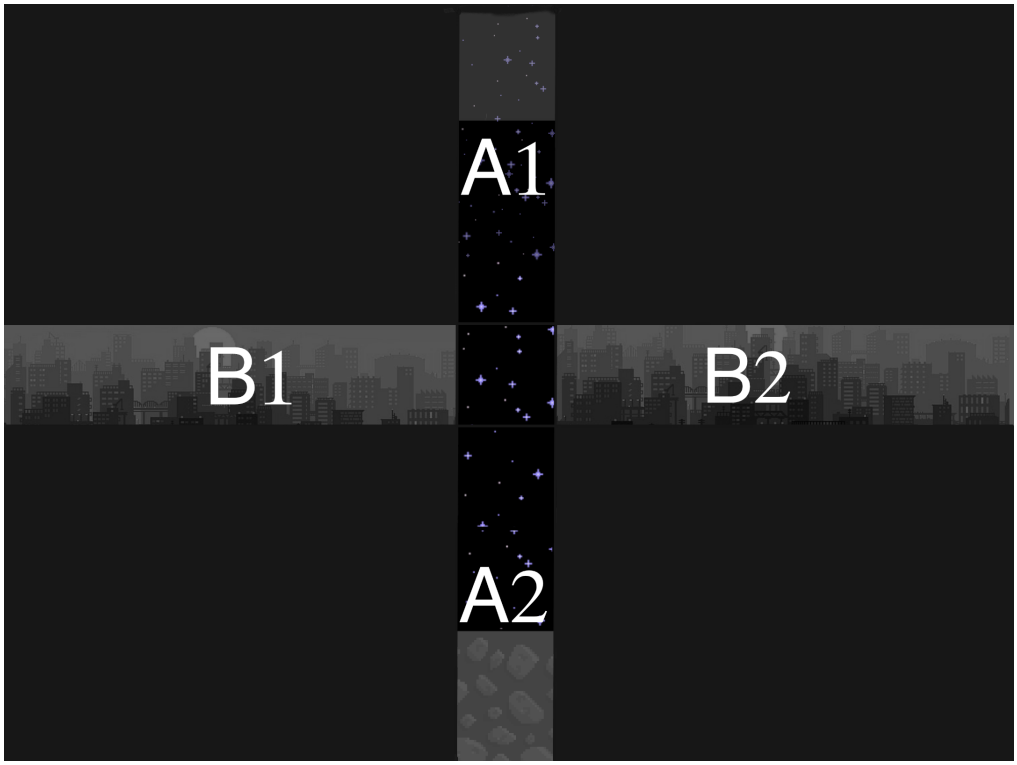
Below the buttons, there are Tunnels leading from the exits of the variables. The variables have access to the buttons they can set true according to the boolean expression.

For example, if x_1 is chosen to be true, then Madeline gets access to the x_1 tunnel and $\neg x_1$if she had chosen false. x_1 tunnel has access to buttons that open a door to the clause having x_1.

How do we block the variables from accessing the other doors? For that, we have constructed the **Crossover Gadget**. The space blocks that are displayed in the diagram are used in a specific manner described in the Crossing Frame.

Crossover Gadget

Since the game is 2D, you can not avoid paths from crossing each other during the construction of such a level. We can make sure that the intersection of the paths happens only in the form of a cross.



Gadget description

Suppose we want Madeline to go from A1 to A2 or B1 to B2 or the other direction. But she shouldn't be able to go from an A to a B or vice versa.

The Space block as described before teleports the player from 1 end to the other in a straight line without any interference. Encountering a wall will kill Madeline and she will respawn at the start of the frame.

So we put the space blocks in the intersection of the paths in such a way that there are no straight lines connecting the opening side of A to B.

Note: It might seem like you can draw a line from A to B but remember that Madeline can only move in 8 directions, so the lines can be parallel or 45 degrees inclined with the axis. So no such line will exist.

This means that the only way she can travel through the space block is in a straight line parallel to the axis, hence she can not access A from B or vice versa.

Sequence of Frames

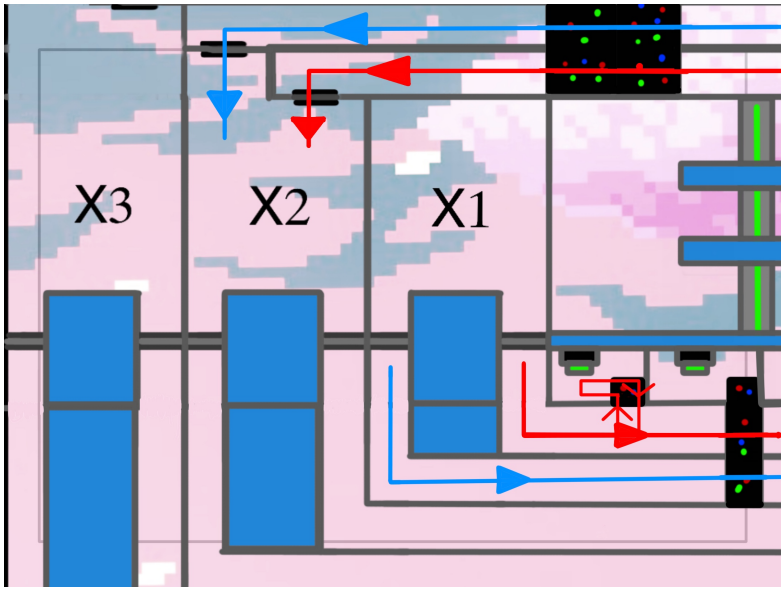
Now that all the frames have been constructed, we decide the sequence of the frames.

Let `n` be the number of variables in the Boolean expression distributed into `k` clauses.

Variable order

We will assume the order of the variables to be $x_1, x_2, x_3 \dots x_n$. We select values for these variables in the same order. So the **starting position** will be in the x_1 gadget since we pick its value first.

Transition between variables

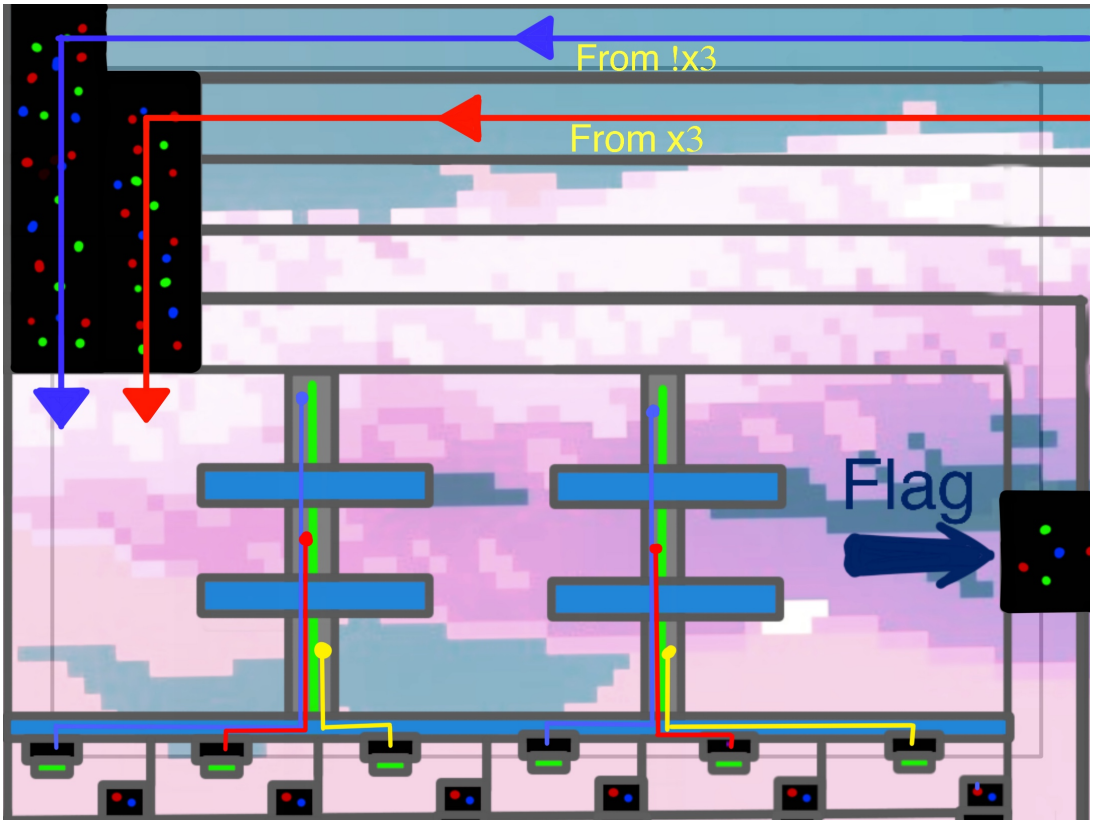


Transition from x1 to x2 after completing the path

From the variable gadget of x_1, we choose its value and go to the respective tunnel. In the tunnel, we press all the accessible buttons, After all the buttons, we reach the end of the tunnel. Now since the values of x_1 have been already picked and substituted, we have to pick a value for the next variable hence we must go to the x_2 variable gadget.

In such order we pick the value of all the variables, click the buttons which open the clause doors, and continue until we reach the end of the x_n variable. Since we ran out of variables, where do we go now?

Final passage



Example case where n = 3

At the end of the x_n passage, we should have an entrance to the Final passage containing the clauses. Madeline after choosing all the variables will now have to reach the flag.

The path to the flag lies on the other side of the passage. If at least one door from each Clause is open only then will she be able to reach to the flag, else she won't be able to complete the level.

This was the AND of all the OR clauses, leading to a normal form with 3 variables in each clause.

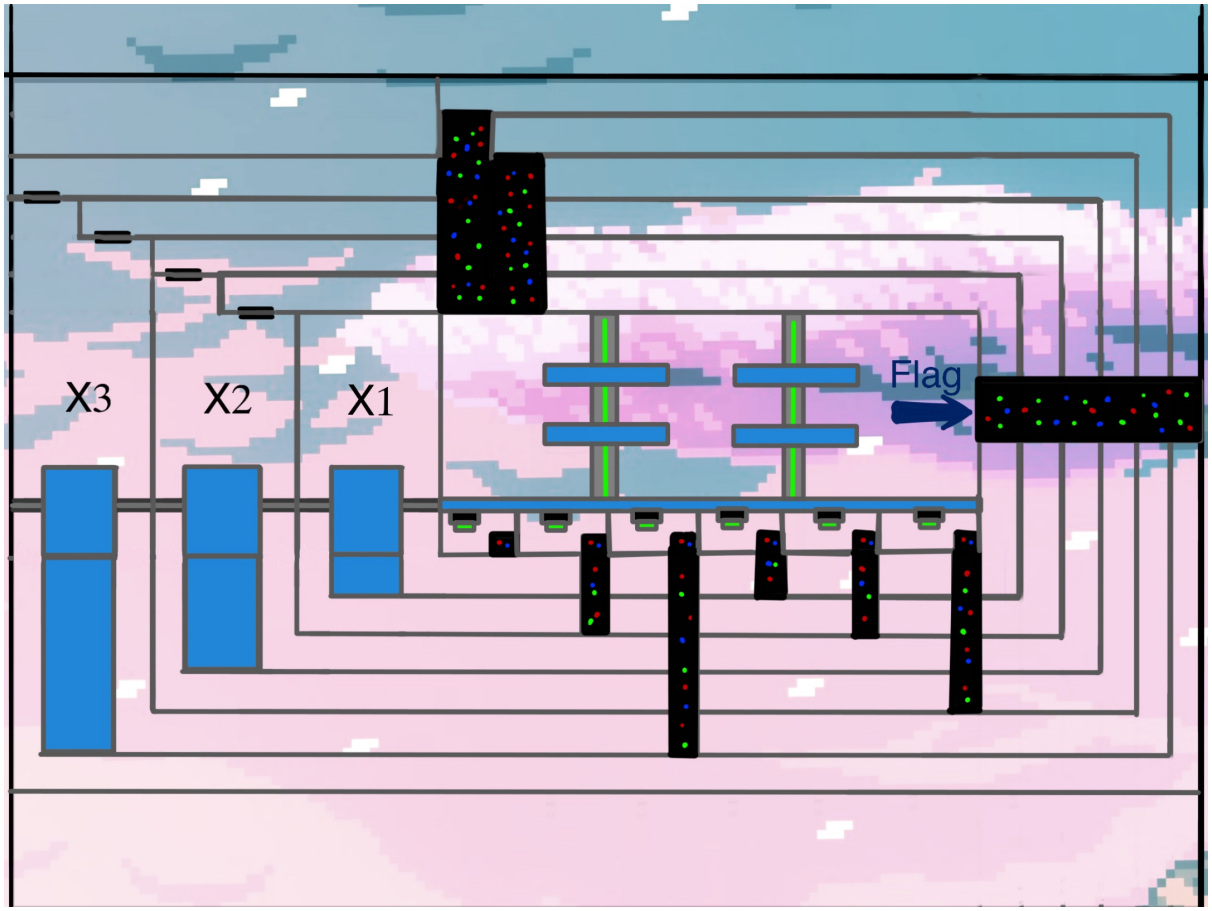
Why not put the flag at the end?

The flag is always at the top of the level. So we need to redirect the player from the end of the final passage to the flag. Since there are other Paths that come between it, we use crossover frame.

Final Level

Now that all the separate parts have been explained, we put together our final level. The Boolean expression for which the level has been implemented is:

$$(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3)$$

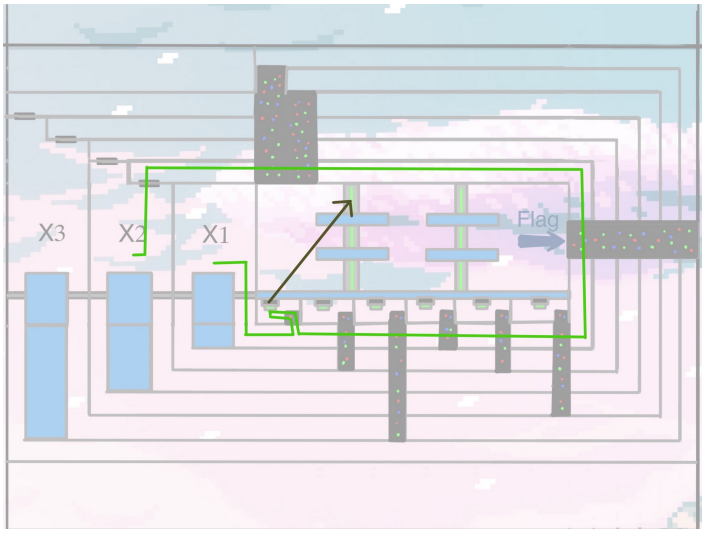


Example Substitution

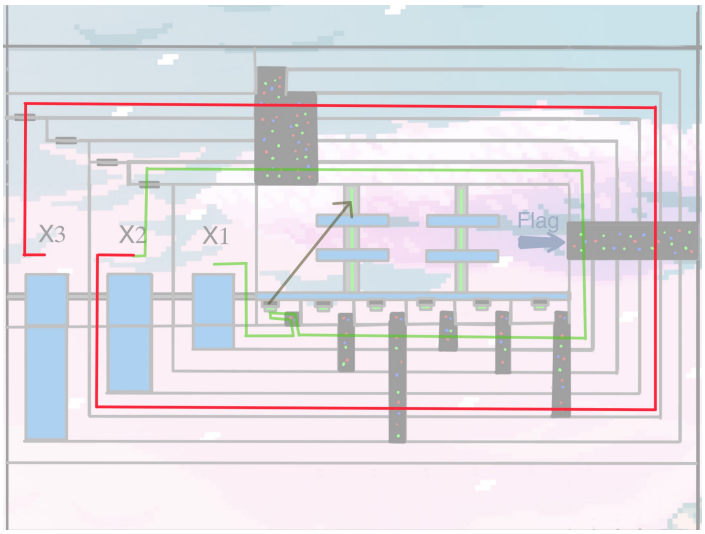
Let the substitution of the variables be:

- $x_1 = 1$
- $x_2 = 0$
- $x_3 = 1$

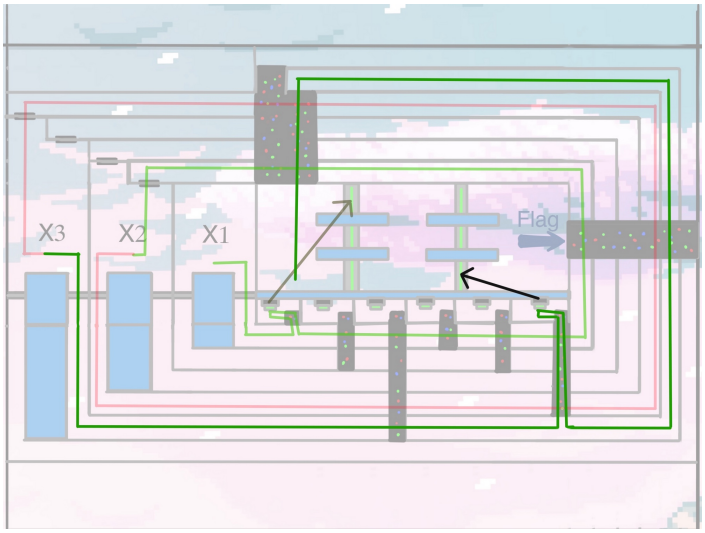
We start with the x_1 gadget, take the true tunnel, and activate the door. After which we end up with a tunnel with an exit leading to x_2 gadget.



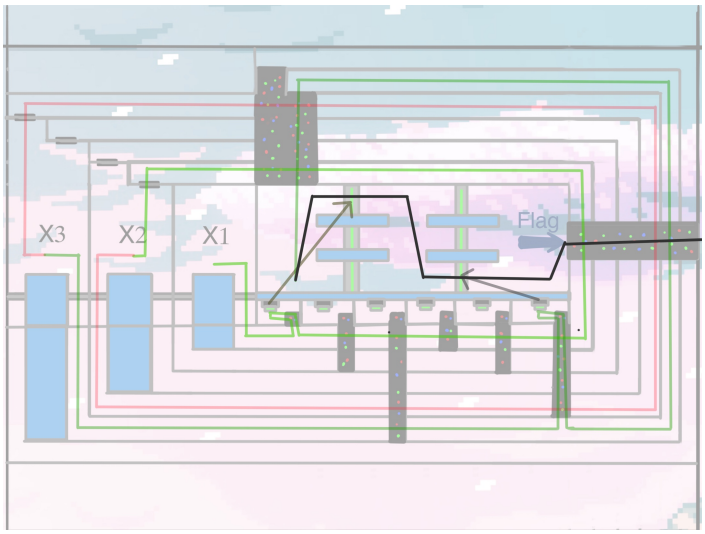
Now we are in the x_2 , we take the false tunnel, but since there is no x_2 in the expression, there is no door that can be opened from this tunnel. So we just continue and end up in the x_3 frame.



In the x_3 frame, we repeat the same procedure. We open a door in the 2nd OR clause and end up at the end of the tunnel which leads to a space block. This space block when used will take Madeline to the beginning of the Final passage.



Now one door of each clause has opened, Madeline can pass the final passage and make it to the flag.



Since Madeline was able to reach the flag. The level could be completed, hence the expression as expected is satisfied with the given values.

Conclusion

This proves that Celeste is at least as hard as 3SAT, making it NP-Hard. In conclusion, the game is both NP and NP-Hard, making it an NP-Complete puzzle.

Making Celeste PSPACE-Complete

In this section, we describe how making a small change to Celeste can make it PSPACE-Complete. In the proof the Celeste is NP-Complete, we used a button door which opened when we pressed the green button and then was obsolete.

We now make an addition to the game. **The door can now be close using a red button.** When the door is open, the red button is deflated and can be activated and when the door is closed the green button can be activated.

Celeste belongs to PSPACE

To prove that Celeste is a PSPACE-Complete puzzle, we have to first show that it belongs to PSPACE.

It is sufficient to show that Celeste belongs to NPSPACE since $NPSPACE \subseteq PSPACE$ by Savitch's Theorem.

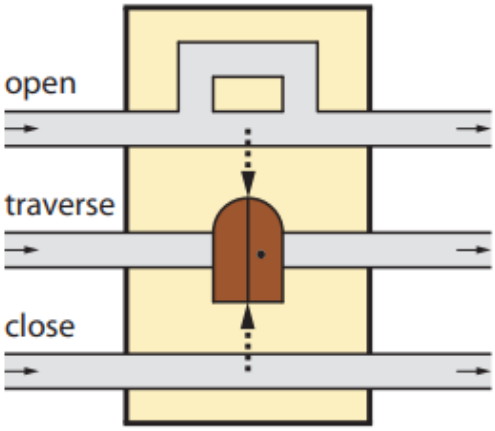
That means, for any given traversal on the level, it has to use polynomial space with respect to the size of the level.

Since the game's element behaviour is a simple (deterministic) function of the player's moves. Therefore, we can solve a level by making moves non-deterministically while maintaining the current game state (which is polynomial),

Framework for PSPACE-Hardness

To prove NP-Hardness of Celeste, we here describe a framework for reducing TBFQ problem to a 2-D platform game. This framework is based on (link source here). Using this framework in hand, we can prove hardness of games by just constructing the necessary gadgets.

For this framework we need one more gadget: **Pressure Button Door Gadget.**



Pressure Button Door Gadget

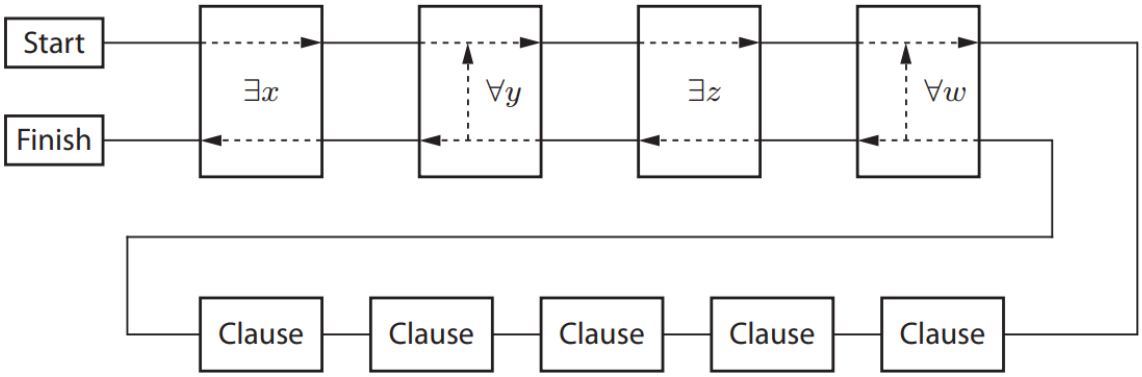
Gadget description

In Celeste, to press the button, Madeline has to dash into it. The button in the above frame is thick enough to prevent Madeline to pass through the tunnel without pressing the button. **Forcing her to press the button**.

- The **open path** has a button which the player is forced to press.
- The **traverse path** is the path containing the door which can be traversed if the door is opened.

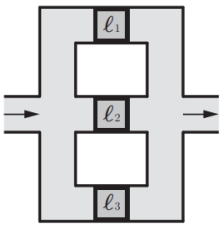
- The **close path** button forces the player to close the door.

The general framework for PSPACE-Hardness



Framework for PSPACE-Hardness

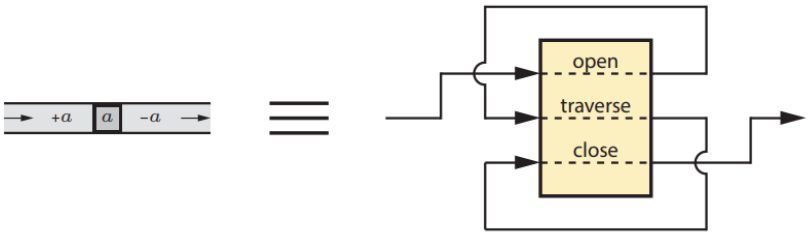
A given fully quantified boolean formula $\exists x \forall y \exists z \ldots \phi(x, y, z, \ldots)$, where ϕ is in 3-CNF is translated into a row of **Quantifier gadgets**, followed by a row of **Clause gadgets**, connected by several paths.



Clause gadget

The clause gadget is built using three gates whose pressure buttons are in quantifier gadgets.

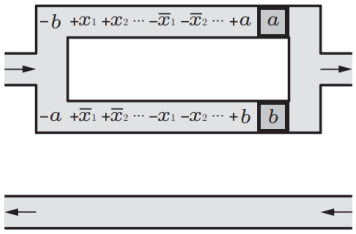
For building Quantifier gadgets we use a special notation in a tunnel as shown below:



Shorthand notation for tunnels

Here, +a opens the gate corresponding to variable and a and -a closes the gate corresponding to gate a. The player is forced to press these buttons as described before.

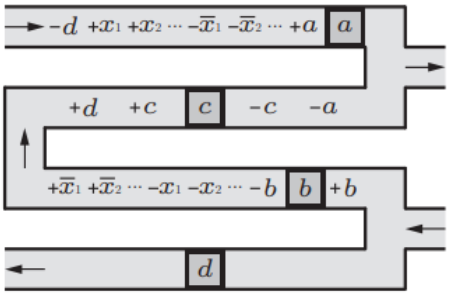
Existential quantifier



Existential Quantifier

The player can only select one of the path ways and once it is selected, the player can never change his choice.

Universal quantifier



Universal Quantifier

The player first proceeds to mark the variable as true and while backtracking has to traverse through the clauses again with the variable marked as false. After both possibilities are tried the player is able to move forward.

Condition for Traversal

Traversing a quantifier gadget sets the corresponding variable in the clauses. When passing through an existential quantifier gadget, the player can set it according to their choice. For the universal quantifier gadget, the variable is first set to true.

A clause can only be traversed if at least one of the variables is set in it. After traversing all the quantifier gadgets, the player does a clause check and is only able to pass if all the clauses are satisfied. If the player succeeds, they are routed to lower parts of the quantifier gadgets, where they are rerouted to the last universal quantifier in the sequence.

The corresponding variable is then set to False and the clauses are traversed again. This process continues and the player keeps backtracking and trying out all possibilities.

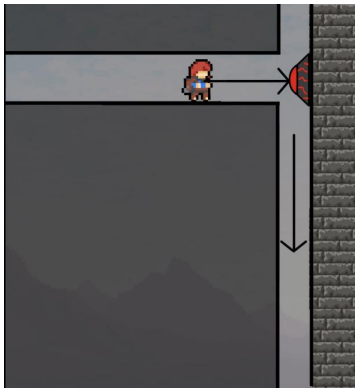
We will later show how to build these quantifier gadgets using Celeste game entities.

Modified Celeste is PSPACE-Hard

Using the previously described framework, we will build corresponding gadgets and thus show a reduction from the TQBF problem to Celeste, implying the Celeste is PSPACE-Hard.

Door Gadget

For creating a door gadget we first need to create a way to force the player to dash into a button to activate it. We do this using a narrow tunnel and **leaving only enough space to pass if the button is pressed.**



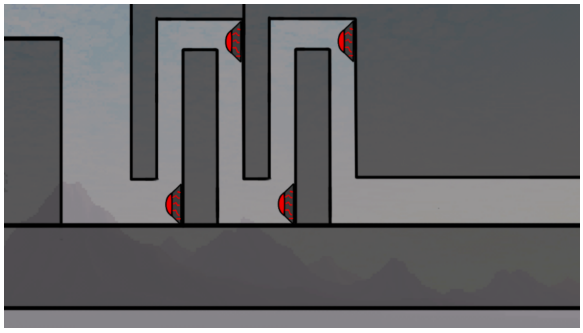
Forcing to press a button

Door Gadget

Now, using this force button, we create a door gadget.

The button above will open the door, and the button below will close the door, and since the path is thin, Madeline will not be able to pass through until the button is pressed.

Multi-Tunnel Gadget



Multi-Tunnel Gadget

For **clubbing** multiple open/close symbols together, we use a multi tunnel gadget. This is just a helpful gadget to make Quantifier Gadgets.

Putting it all together

Since we were able to make the gadgets described in the framework, we have a reduction from the TBFQ problem to Modified Celeste, thus Modified Celeste is PSPACE-Hard

Conclusion

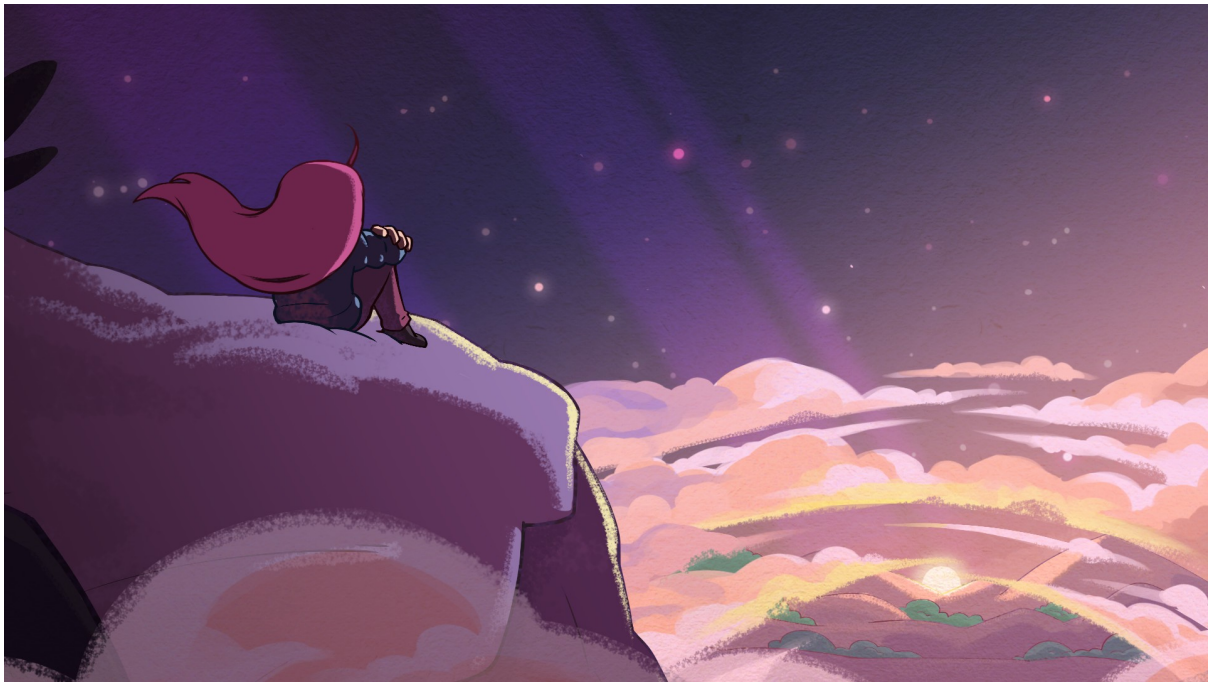
In conclusion, the modified game is both PSPACE and PSPACE-Hard, making it a PSPACE-Complete puzzle.

But what exactly changed?

On adding the close button, we added a requirement to keep track of all the doors that are open. Before once a door was open, it **always remained open.** Before we had to only keep track of the current state sequentially as all the doors will be opened in a sequence. Knowing that a door is openly implied that all the previous doors were opened so as to reach the current door.

But after adding the close button, at any point of the game, all the doors are independent and knowledge of the open state of a door gives us no info about the other doors. So, we have to keep track of all the other doors. This makes the game harder and makes its PSPACE-Hard instead of NP-Hard.

Lack of knowledge about the status of all the doors makes the game PSPACE-Complete .



Part - II

In Part-I we have learnt about Computation, Universal Turing Machines and Complexity Classes. We have seen how powerful the idea of Reduction can be in classifying "problems" in "certain classes" and have demonstrated the same in the shrewd proof stating that it is NP-Complete to win a generalized level in the Game Celeste.

Now in this part, we shall talk further about Games - how to model them, classify them, provided generalized theorems and discuss how and why having knowledge about such aspects of different games is important not only to our understanding of Complexity Theory but also to the world of Computation, in general.

Why Games?

Games involve the most vibrant (and in many cases even the oldest) set of computational problems. Chess, for example, is an ancient game believed to have originated in India around 1500 years ago and has been proven to be EXPTIME-Complete (in exclusion of the fifty-move rule) in 1981.

Games serve as models of computation which have been used quite prevalently used to mathematically model real-life scenarios. For example, classical game theory deals with several games involving rational decision making strategies and has found applications in computer science, biology and social sciences.

Games not only serve as a means of understanding computation but also decision making and behavioral relations. This is why they have been often found associated with numerous breakthroughs in artificial intelligence. It is quite astonishing as how games often are found to be embedded in deep computational problems and yet require incredibly less to none formal understanding to be played.

Moreover, puzzles and simulations can often be helpful to encapsulate real life problems especially in fields like bioinformatics.

Thus, it would indeed be surprising if understanding of games and augmented reality would provide us with no further **help in the understanding of the nature** as well as in **trying to answer some of the major philosophical questions** encompassing life and reality.

Game Theory

Game Theory serves as a good formalism to study a certain category of games (mostly non-cooperative multi-player games). It is very well studied field and we shall start of with it and then move forward to other formalisms and study more categories of games like simulations, puzzles, their video game counterparts as well as multi-player video games.

Introduction

> An equilibrium is not always an optimum; it might not even be good.

One very important common motivation of both Game Theory and Computer Science is to be able to **model rationality** and solve cognitive tasks such as **negotiation**.

In this module, we shall introduce Game Theory with a Complexity Theory minded approach. As it turns out, Game Theory has given rise to several interesting challenges with respect to computational . complexity especially in the realm of the complexity classes PPAD and PLS.

Prisoner's Dilemma and Nash Equilibrium

In the movie 'A Beautiful Mind', there is a memorable scene where we find Russell Crowe playing Dr. John Nash say that Adam Smith's statement - "In competition, individual ambition serves the common good" - was incomplete. Instead, Nash postulates that the best result is for everyone in the group doing what’s best for himself and the group. This conversation serves as a very

informal introduction to the idea behind Nash Equilibrium. Here, we shall now formalise it starting with an example.

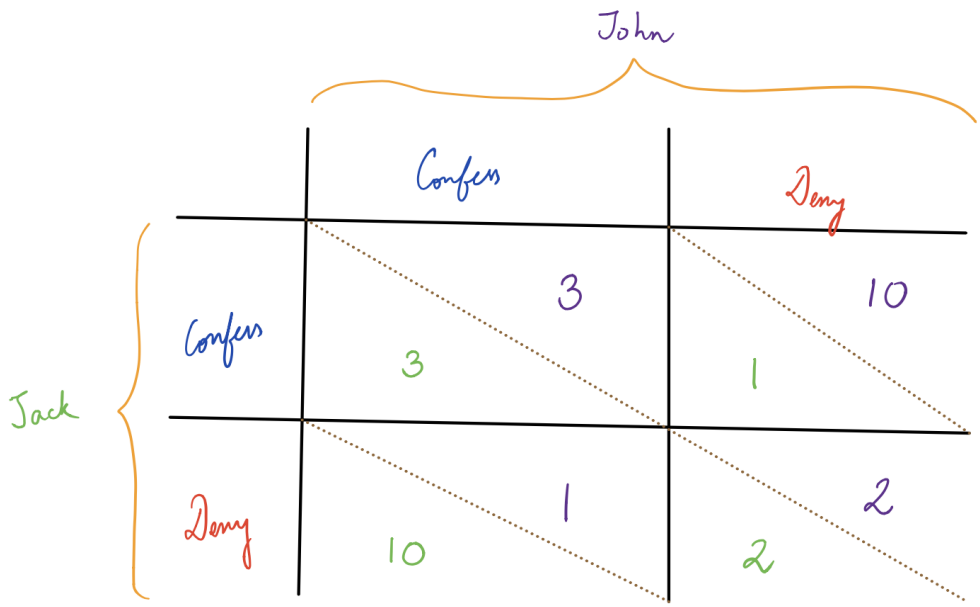
The Prisoner's Dilemma

Suppose that Jack and John are two two individuals charged and convicted with a minor case of money laundering. They both have to serve 2 years of prison each. However, the judge has a suspicion that both of them (they are acquaintances) are involved in some armed burglary (serious felony).

Now, the judge has no evidence present with him at hand. So, he proposes the following to both of them:

- if both of you deny involvement in the burglary, then both of you receive only 2 years of prison each
- if one confesses while the other denies, then the one who confessed gets 1 year while the other gets 10 years of prison
- if both of you confess, then both of you receive 3 years of prison each

Assuming that the Jack and John have no loyalty amongst themselves, we should observe that they will pick a non-optimal scenario.



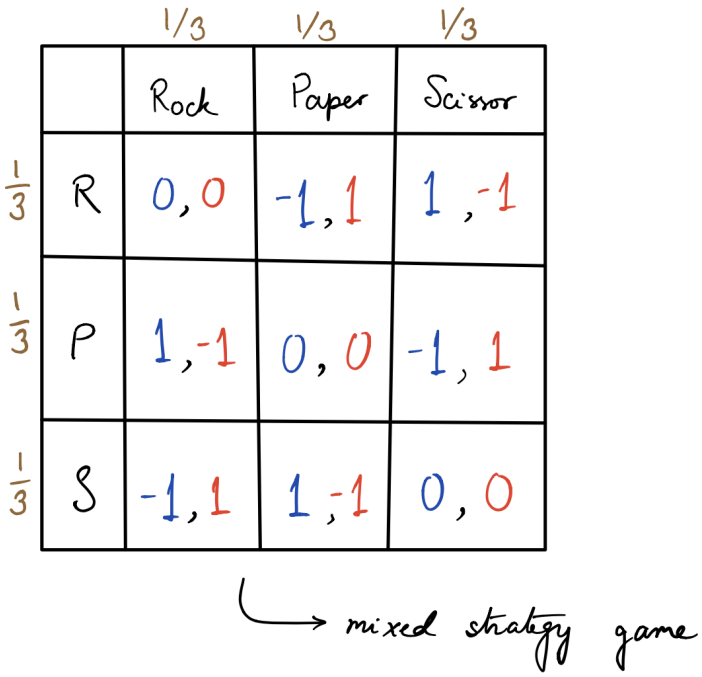
Here, there exists a global optimum in the case where both the prisoners lie. However, given our predisposed knowledge of the situation it might not be the best rational choice for the prisoners.

The best rational choice for the prisoners would be a sub-optimal choice presented in the case where both of them confess. The case serves as a **sub-optimal stable equilibrium** and is referred to as the Nash Equilibrium.

Rock-Paper-Scissors

Let us try the above payoff matrix method shown above for the popular game "Rock-Paper-Scissors".

If you have played the game for a lot of times, you may already have a good intuition as to which scenario will give rise to a Nash Equilibrium.



Given the payoff matrix, it is evident that only when both players randomize with equal probability, we obtain a Nash Equilibrium.

Interestingly, this game also serves as a proof by contradiction for the statement that not all games have a Pure Nash Equilibrium. So what's a Pure Nash Equilibrium? Let's find out.

Games: Definitions and Notations

Games can be defined with a set of **players**, where each player has his own set of allowed **actions** (known as **pure strategies**). Any combination of actions will result in a numerical **payoff** (specified by the game) for each player.

Let the number of players be $1, 2, 3, ..., k$ and S_i denote player i's set of actions. n denotes the size of the largest S_i. If k is a constant we look for algorithms that are $O(n^\kappa)$ where $\kappa \in \mathbb{N}$.

Let us look at the above statements in the light of the game of Rock-Paper-Scissors which was discussed above.

- players, $p = \{1, 2\}$
- $\forall i \in p,\ S_i = \{\text{rock}, \text{paper}, \text{scissor}\}$
- Here, of course $k = 2$, which actually is one of the most studied special case in game theory.
- $n = 3$

Let $S = S_1 \times S_2 \times S_3 \times ... \times S_k$. Then, S is the set of **pure strategy profiles**. Then, each $s \in S$ gives rise to payoff to each player and u_s^i denotes the payoff to player i when all players choose s.

- The list of all possible u_s^i's yields a **normal-form game**, which for a k-player game should be a list of kn^k numbers.
- We have have other models too. For example, a **Bayesian game** is one where u_s^i can be a probability distribution over i's payoff.

Nash Equilibrium

A Nash Equilibrium is a set of strategies - one strategy per player - such that no player has an incentive to unilaterally change his strategy (stable sub-optima). In case of two player zero-sum games, Nash Equilibrium is also optimal.

- **Pure** NE: where each player chooses a pure strategy
- **Mixed** NE: where for each player a probability distribution over his pure strategies is applied (in case of MNE we have expected payoff's associated with each player)

Formal Definition of Nash Equilibrium

From the above, $u_s^i = u_i(s)$ where $s \in S$.

An **individual mixed strategy** is a probability distribution on the set of available strategies. Such as in the last example, selecting one of "rock", "paper", or "scissors" uniformly at random is an example of a mixed strategy.

A **pure strategy** is simply a **special case** of a mixed strategy, in which one strategy is chosen 100% of the time.

- A **Nash equilibrium** is a strategy profile $s = (s_1, s_2, \ldots, s_n)$ with the property that,

$$u_i(s) \geq u_i((s_1, s_2, \ldots, s_i', \ldots, s_n))$$

  $\forall i$, where $s_i' \in S_i$ denotes a strategy other than s_i available to player i.

In the event that this inequality is strict, $u_i(s) > u_i((s_1, s_2, \ldots, s_i', \ldots, s_n))\ \forall i$, the profile s is called a **Strong Nash Equilibrium.** Otherwise, s is called a **Weak Nash equilibrium**.

Existence Theorem

Any game with a finite set of players and finite set of strategies has a Nash Equilibrium of Mixed Strategies or MNE.

- Nash's existence theorem guarantees that as long as S_i is finite $\forall i$ and there are a finite number of players (p), at least one Mixed Strategy Nash equilibrium exists.

Some Computational Problems

First Decision Problem

Input: A game in normal form.

Question: Is there a Pure Nash Equilibrium?

Special Case Solutions:

- Determining whether a strategic game has a pure Nash equilibrium is NP-complete given that the game is presented in GNF having a bounded neighborhood or in the case of acyclic-graph

or acyclic-hypergraph games.

- Pure Nash Equilibrium existence and computations problems are feasible in logarithmic space for games in standard normal form.

A Second Better Problem

Input: A game in normal form.

Output: Is there a Mixed Nash Equilibrium?

Following the steps of quite a many papers and books written on this topic, we shall call this problem NASH.

Speculation

By Nash's Existence Theorem, this is intrinsically a search problem and not a decision problem.

This might tempt one to look for efficient algorithms for the above problem. There were many attempts in that direction.

> John von Neumann in the 1920s that MNE can
> be formulated in terms of linear programming for zero-sum games.

But what about games that are not zero-sum?

Several algorithms were proposed over the next half century, but all of them are either of unknown complexity, or known to require, in the worst case, exponential time.

Well then, one might ask, is NASH NP-complete?

Complexity of NASH

Equal-Subsets

Input: $A = \{a_1, \ldots, a_n\}$ where $\Sigma_{\forall i} a_i < 2^n - 1$ and $a_i \in \mathbb{Z}, \forall i$

Output: Two distinct sets $A_1,\ A_2$ such that $\Sigma_{\forall j \in A_1} j = \Sigma_{\forall k \in A_2} k$.

Before we further discuss NASH let's talk about the problem of EQUAL-SUBSETS.

It is evident that **EQUAL-SUBSETS** $\in$ **NP**, just like NASH. But is it NP-hard too or in other words - **is it reducible from SAT**? Is it NP-complete?

Total Search Problems

EQUAL-SUBSETS and NASH form a very specific group of problems which appear to be different from typical NP-complete problems like SAT.

Why? **Because unlike SAT,** these problems have a **guaranteed solution.** These problems (EQUAL-SUBSETS, NASH etc.) thus are Total Search Problems - problems where every instance has a solution. But is this difference enough to conclude that EQUAL-SUBSETS and NASH are not NP-complete?

Proof: **Total Search Problems in NP are not NP-complete**

Suppose Ψ is a total search problem in NP which is also NP-Complete. Then, SAT $\preceq_p \Psi$ and $\exists$ an algorithm A for SAT that runs in polynomial time, provided that it has access to poly-time

algorithm A' for Ψ.

Now, suppose A has a non-satisfiable formula ϕ which calls A' some number of times. the algorithm eventually returns the answer NO implying that ϕ is not satisfiable.

Now, further suppose that we replace A' with a similar non-deterministic algorithm which in case of EQUAL-SUBSETS would be guess-and-test. This gives us a non-deterministic poly-time algorithm for SAT. The entire algorithm can recognize this non-satisfiable formula ϕ, as before and we have a **NP algorithm that recognizes unsatisfiable formulae.**

This implies $\mathbf{NP} = \mathbf{co\text{-}NP}$.

And, this is specifically why NASH too is very unlikely to be NP-complete. Below, we have another excellent argument by Megiddo.

> Suppose we have a reduction from SAT to NASH, that is, an efficient algorithm that takes as input an instance of SAT and outputs an instance of NASH, so that any solution to the instance of NASH tells us whether or not the SAT instance has a solution. Then we could turn this into a nondeterministic algorithm for verifying that an instance of SAT has *no* solution: Just guess a solution of the NASH instance, and check that it indeed implies that the SAT instance has no solution.
> The existence of such a non-deterministic algorithm for SAT (one that can verify that an unsatisfiable formula is indeed unsatisfiable) is an eventuality that is considered by complexity theorists almost as unlikely as $\mathbf{P} = \mathbf{NP}$. We conclude that NASH is very unlikely to be **NP**-complete.

This implies that NASH and similar problems are highly likely to be easier than NP-complete problems. What what is the complexity class we can associate them to? A question still remains as to is it easy or is it hard?

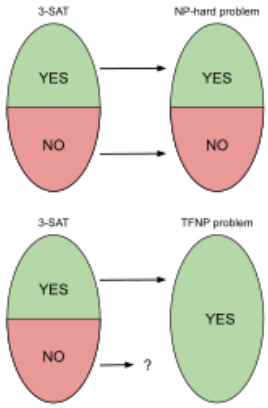
TFNP: The Complexity Class

TFNP or Total Function Non-deterministic Polynomial problems are total function problems which can be solved in non-deterministic polynomial time. It is a subset of FNP - a function problem extension of the decision class NP.

This is the complexity class where we shall prove that NASH exists. It also contains the problems FACTORIZATION and EQUAL-SUBSETS.

> TFNP is widely conjectured to contain problems that are computationally intractable.

But, there hasn't been any results yet which show that TFNP problems are NP-hard. Moreover, TFNP-complete problems are not believed to exist.



> The top image shows the typical form of a reduction that shows a problem is NP-hard. Yes instances map to yes instances and no instances map to no instances. The bottom image depicts the intuition for why it is difficult to show TFNP problems are NP-hard. TFNP problems always have a solution and so there is no simple place to map no instances of the original problem. This is the intuition behind the lack of NP-hardness results for TFNP problems.

Proving TFNP Membership

Proving that various search problems are total often use a "non-constructive step" which is hard to compute (unless the problem itself can be efficiently solved). However, it turns out that for most known total search problems these non-constructive steps are one of very few arguments.

These arguments define sub-classes within TFNP since they have mathematical theorems associated that prove their guarantee of existence of solution. TFNP, thus, is often studies through the lenses of these sub-classes and interestingly, these sub-classes have complete problems by virtue of the argument associated evn though TFNP itself does not have known complete problems.

Arguments and Sub-classes

- “If a function maps n elements to n − 1 elements, then there is a collision.” This is the *pigeonhole principle*, and the corresponding class is **PPP**.
- “If a directed graph has an unbalanced node (a vertex with different in-degree and out-degree), then it must have another.” This is the *parity argument* for directed graphs, giving rise to the class **PPAD** considered in this article.
- “If a graph has a node of odd degree, then it must have another.” This is the parity argument, giving rise to the class **PPA**.
- “Every directed acyclic graph must have a sink.” The corresponding class is called **PLS** for *polynomial local search*.

To understand how to define membership to these classes let's go back to our old friend the class NP.

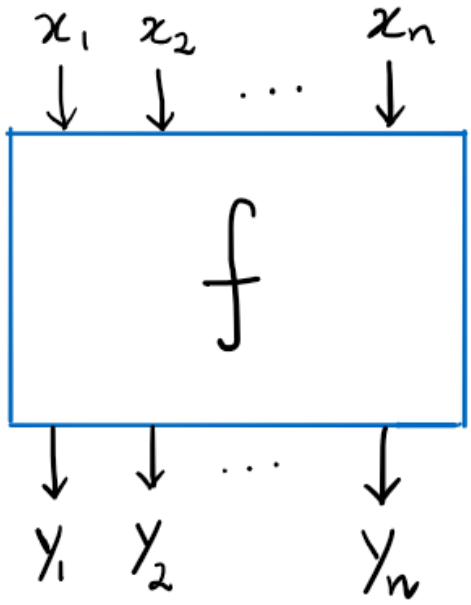
One of the many ways to define NP is *class of all problems that can be reduced into instances of SAT*. Now, we shall define the above subclasses in similar fashion - **define classes** by their

complete problems.

PPP and EQUAL-SUBSETS

We used the similarity between EQUAL-SUBSETS and NASH to introduce this new class of problems, namely TFNP. Now, it would be quite a shame if we don't actually prove the membership of EQUAL-SUBSETS in TFNP. Moreover, since it is a member of PPP, it will be quite interesting to see how a very simple yet powerful theorem (the pigeonhole principle) that we all have been introduced quite early in high school gives rise to a complexity class of it's own.

PPP stands for *polynomial pigenhole principle*. For the purpose of reductions we construct a gadget in the form of a PIGEONHOLE-CIRCUIT which in itself by definition is a PPP-complete problem.



Input: A boolean circuit C that takes n inputs and n outputs.
Output: A binary vector $\vec{x}$ such that $C(\vec{x}) = 0^n$ or alternatively, 2 vectors $\vec{x}$ and $\vec{x'}$ with $f(\vec{x}) = f(\vec{x'})$.

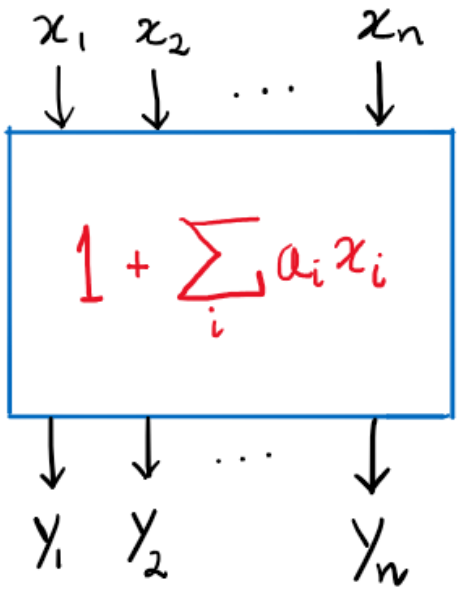
With regard to questions of polynomial time computation, the following are equivalent:

- n inputs/outputs $\rightarrow C$ of size n^2
- n inputs/outputs $\rightarrow C$ of size $O(n^k), k \in \mathbb{N}$
- $n =$ number of gates in C, number of inputs = number of outputs

Definition: A problem Ψ belongs to PPP if Ψ reduces to PIGEONHOLE CIRCUIT in polynomial time. Furthermore, if we can reduce Ψ from PIGEONHOLE CIRCUIT then Ψ is PPP-complete.

EQUAL-SUBSETS belongs to PPP

Taking inspiration from the book Proof without Words, we have:



Example:

Consider, $A = \{2, 3, 4, 5\}$ and $C = 1 + \Sigma_i(a_i x_i) = 1 + \begin{bmatrix} 2 & 3 & 4 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$

Then?

We have two vectors 2 vectors $\vec{x} = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix}$ and $\vec{x'} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$ such that, $C(\vec{x}) = C(\vec{x'})$.

Why?

Since, $1 + \begin{bmatrix} 2 & 3 & 4 & 5 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} = 1 + \begin{bmatrix} 2 & 3 & 4 & 5 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = 5 = (0101)_b$.

Now, before we delve into the next class and classifying NASH let's refine the problem statement bore by NASH.

Refining NASH

Let's state the original problem that we posed.

Input: A game in normal form.

Output: Is there a Mixed Nash Equilibrium?

Now, in this problem for two-player games *the probabilities used by the players are rational numbers* (given that their utilities are also rational). So, it is clear how to write down the solution of a 2-player game.

However, as pointed out in Nash's original paper, when there are more than two players, there may be **only irrational solutions**.

As a result, the problem of computing a Nash equilibrium has to deal with issues concerning numerical accuracy. Therefore, we introduce the concept of Approximate Nash Equilibrium.

Approximate Nash Equilibrium

Input: A game in normal form where, $u_s^p \in [0, 1]$.

Output: Is there a Mixed ϵ-Nash Equilibrium?

$$\epsilon\textbf{-Nash Eqm: } E[\text{payoff}] + \epsilon \geq E[\text{payoff}] \text{ of best possible response}$$

This should be at least as tractable as finding an exact equilibrium, hence any hardness result for approximate equilibria carries over to exact equilibria.

NASH: "Re"-Definition

> Bounded rationality fixes irrationality.

Given the *description of a game* (by explicitly giving the utility of each player corresponding to each strategy profile), and a *rational number* $\epsilon > 0$, compute a *(mixed) Approximate Nash Equilibrium*.

END-OF-A-LINE Problem

Input: A directed graph G and a specified unbalanced vertex of G.

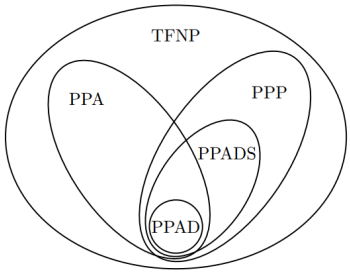
Output: Some other unbalanced vertex.

Now, by parity argument to know that a solution always exists.

> Suppose G has 2^n vertices, one for every bit string of length n (the parameter denoting the size of the problem). For simplicity, we will suppose that every vertex has at most one incoming edge and at most one outgoing edge. The edges of G will be represented by two boolean circuits, of size polynomial in n, each with n input bits and n output bits.
> Denote the circuits P and S (for predecessor and successor). Our convention is that there is a directed edge from vertex v to vertex v', if given input v, P outputs v' and, vice-versa, given input v', P outputs v.
> Suppose now that some specific, identified vertex (say, the string $\{0\}^*$) has an outgoing edge but no incoming edge, and is thus unbalanced. With the restriction of at most one incoming and one outgoing edge, the directed graph must be a set of paths and cycles; hence, following the path that starts at the all-zeroes node would eventually lead us to a solution. The catch is, of course, that this may take exponential time. Is there an efficient algorithm for finding another unbalanced node without actually following the path?

PPAD: The Class

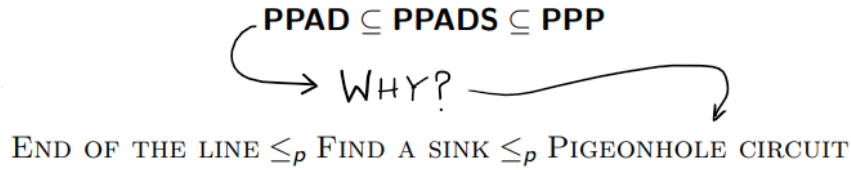
> PPA... what?
>
> **- Papadimitriou**



PPAD (which stands for *Polynomial Parity Arguments on Directed graphs*) is the class of all problems *reducible to END-OF-A-LINE*.

As expected, PPAD problems are believed to be hard, but obtaining PPAD-completeness is a weaker evidence of intractability than that of obtaining NP-completeness.

Arguably, the most celebrated application of the complexity of total search problems is the characterization of the computational complexity of finding a Nash Equilibrium in terms of PPAD-completeness. This problem lies in the heart of game theory and economics. The proof that Nash Equilibrium is PPAD-complete initiated a long line of research in the intersection of computer science and game theory and revealed connections between the two scientific communities that were unknown before.



Does PPAD contain hard problems?

In the absence of a proof P $\neq$ NP, we cannot confirm whether PPAD has hard problems. However, END-OF-A-LINE does appear to be a hard problem. Why?

- END-OF-A-LINE is a hard problem in the query complexity model, i.e., when S, P are given as black-box functions. We are interested in the computational variant where we have access to the circuits. But, except for very few exceptions, we don't know any way of looking inside a circuit and extracting useful information from it.
- Moreover under certain cryptographic assumptions (e.g., indistinguishablility obfuscation) the computational variant of END-OF-LINE is hard.

NASH is PPAD-complete

Here, we shall provide an outline to give an intuition behind why NASH is PPAD-complete.

Two parts of the proof are:

- Nash is in PPAD i.e., $\text{NASH} \preceq_p \text{END-OF-A-LINE}$
- $\text{END-OF-A-LINE} \preceq_p \text{NASH}$ or, NASH is PPAD-hard

Part 1:

To prove that NASH is in PPAD, we can use a lot from Nash's original proof of existence, which utilizes Brouwer's fixed point theorem - it is essentially a reduction from NASH to BROUWER (problem of finding an approximate Brouwer fixed point of a continuous function).

Now, BROUWER can be reduced, under certain continuity conditions, to END-OF-A-LINE, and is therefore in PPAD.

$$\text{NASH} \preceq_p \text{BROUWER} \preceq_p \text{END-OF-A-LINE}$$

Let G be a normal form game with k players, $p = \{1, 2, \ldots, k\}$, and strategy sets $S_i = [n],\ \forall i \in [k]$ and let $\{u_s^i : i \in [k], s \in S\}$ be the utilities of the players.

Also let $\epsilon < 1$. In time polynomial in $|G| + \log(1/\epsilon)$, we will specify two circuits S and P each with $N = \text{poly}(|G|, \log(1/\epsilon))$ input and output bits and $P(0^N) = 0^N \neq S(0^N)$, so that, given any solution to END-OF-A-LINE on input S, P, one can construct in poly-time an ϵ-approximate Nash Equilibrium of G. This is enough for reducing NASH to END-OF-A-LINE with some further observations.

The details of construction of S and P as well as the complete proof has properly been presented in the original 2008 Paper by Daskalakis et. al.

Part 2:

For this part we shall use the result that BROUWER is PPAD-complete and simulate the BROUWER with a game, which effectively reduces BROUWER to NASH.

The PPAD-complete class of Brouwer functions that appear above have the property that their function F can be efficiently computed using arithmetic circuits that are built up from standard operators such as addition, multiplication and comparison. The circuits can be written down as a *data flow graph*, with one of these operators at every node. The key is to simulate each standard operator with a game, and then compose these games according to the *data flow graph* so that the aggregate game simulates the Brouwer function, and an ϵ-Nash equilibrium of the game corresponds to an approximate fixed point of the Brouwer function.

We shall not be adding an elaborate reduction here, which can be found in the original paper as well.

2-NASH is PPAD-complete

Previously, we had discussed why we study Approximate Nash Equilibrium. We had mentioned how for a two-player game, approximation is not required necessarily. Here, we shall concern ourselves with that apparent special case.

Input: A 2-player game in normal form.
Output: Is there a Mixed Nash Equilibrium?

Original version of NASH with 2 players.

Let, this problem be called 2-NASH.

Remember that unlike our redefined NASH, 2-NASH is not mandated to be approximate.

Proof

- 2-NASH to BROUWER: As in before, this is guaranteed by John Nash's original work on the proof of existence of MNE.
- END-OF-A-LINE to BROUWER: This too is evident from the previous discussions.

- BROUWER to IMITATION-GAME: We shall try to reduce IMITATION-GAME from BROUWER. Here, consider the following utility functions.

$$u_1(x,y) = -||x-y||_2^2, \text{ and } u_2(x,y) = -||f(x)-y||_x^2$$

Only when, $y = x = f(y)$ this game has a Nash Equilibrium. However, now both players have infinite actions associated. So, we make the space discrete and both players have exponentially many actions. To fix these, we first reduce to a similar game with $2n$ players with the following utility functions.

$$u_{2i-1}(x_{2i-1}, y_{2i-1}) = -||x_{2i-1} - y_{2i-1}||^2, \text{ and } u_{2i}(x, y_{2i}) = -||f_{2i}(x) - y_{2i}||_x^2$$

- BROUWER to $2n$-NASH: We have $2n$ utility functions as described above and effectively reduce the actions per player to a constant number of actions.
- $2n$-NASH to 2-NASH using LAWYER'S-GAME: Let, Alice and Bob be the lawyers for the players $2i-1$ and $2i$ respectively. Since, lawyer's choose actions of players we have the following.

$$\text{Alice's actions: } S_1 \times S_3 \times \ldots \times S_{2n-1}$$
$$\text{Bob's actions: } S_2 \times S_4 \times \ldots \times S_{2n}$$

We can furthermore define the utilities of lawyers and make some necessary concessions or arrangements such as forcing the lawyer's to use all of their players. We use HIDE-AND-SEEK to achieve this. Effectively, Alice incentives to choose the same index as Bob whereas Bob tries to hide.

$$(u_A)_{MP}(i,j) = \begin{cases} M & \text{if } i = j \\ 0 & \text{otherwise} \end{cases} = -(u_B)_{MP}(i,j)$$

Here, M is a sufficiently large number. Condition of MNE: when Alice and Bob play uniformly at random.

Conclusions

Game Theory encompasses a **large genre** of games and auctions. It serves as a formalism for these too. And, we have shown how Nash Equilibrium a fundamental concept in Game Theory is likely to be hard as well as related the field to the study of complexity classes and effectively to computation itself. However, it is **not possible to model** all forms of games especially **simulations and puzzles** (which include 2D platform video games) into Game Theory.

> How about games such as chess?

We can capture this and other similar games (go, checkers) in the present framework by considering two players, Black and White, each with a huge action set containing all possible maps from positions to moves - but of course such formalism (Game Theory) is **not very helpful for analyzing complexity of chess and similar games**.

Formalism: A Short Introduction

> Its all a game.
>
> **- Erik Demaine**

In Part I, as we prove that Celeste, our original game of choice is NP-complete and in a different case PSPACE-complete by reducing it from **Circuit Formalism** of SAT and TQBF respectively. This is the more or less standard procedure in classic game complexity literature.

The Satisfiability problem (SAT) is by nature *a puzzle*: the player must existentially make a series of variable selections, so that the formula is true. And thus, the corresponding model of computation obtained is non-determinism, and the natural complexity class is **NP**.

When we add existential and universal quantifiers, we create the True Quantified Boolean Formula (TQBF) which has a natural interpretation as a 2-player game. The corresponding model of computation is alternation and the associated class is **PSPACE**.

Furthermore, allowing the players to continue to switch the variable states indefinitely creates a formula game of unbounded length, raising the complexity to **EXPTIME**.

Typically, most hardness results we find, including ours, are direct reductions (many seem more natural than several theoretical problems) from such formula games or their variants.

However, even though these Circuit Formalisms are exremely versatile and reducible into classic puzzles and games, they do suffer a few limitations in certain genres.

Limitations of Circuit Formalism

- Geometric constraints typically found in board games do not naturally correspond to any properties of formula games.
- Extension to higher classes maybe cumbersome.
- Too many gadgets? It can be a problem in certain cases.

Constraint Logic

This new formalism which is originally formulated by Erik Demaine and Robert Hearn (2009). The main purpose for this formalism is to address the limitations of Circuit Formalism and provide an unifying framework which can be used to prove the hardness of several classes and genres of games - ranging from simulations to puzzles and 2-player games (Board games, Platform video-games, etc.).

Constraint Logic can be considered an intersection of Graph Theory and Circuit Formalism. We have what Demaine refers to as a *constraint graph* and *rules* that define legal moves on such graphs. The games serves as computation when it is played on a constraint graph and simultaneously "serves as a useful problem to reduce to other games to show their hardness".

Advantages

- Planar graphs provide natural correspondence with typical board game topology.

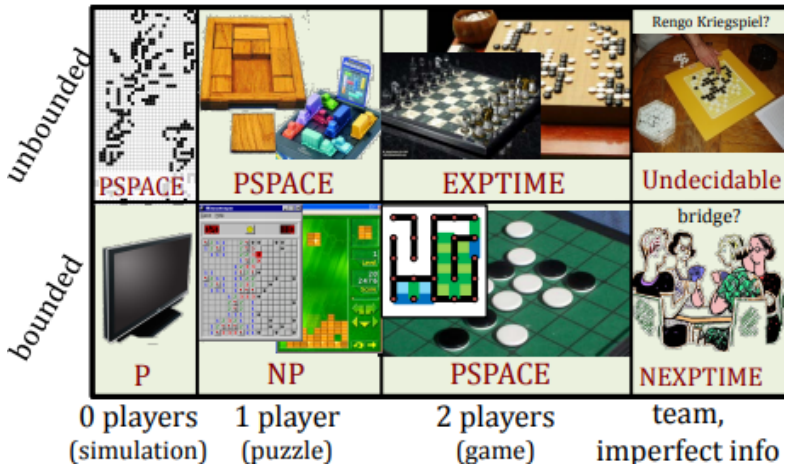
- No variables, formulas or large number of gadgets - only a *constraint graph*. This results in simpler reductions (often).
- Generalizability - more versatile and flexible

> This is awesome.
>
> **- Me**

Now, before we delve into understanding *constraint graphs* and therefore C*onstraint Logic*. Let us look at an amazing generalisation that this theory manages to produce with relative ease (unlike others).

	Zero player (simulation)	One player (puzzle)	Two player	Team, imperfect information
Unbounded	PSPACE	PSPACE	EXPTIME	Undecidable
Bounded	P	NP	PSPACE	NEXPTIME

Okay, now you might be asking yourself questions like what the hell is unboundedness (even though I gave away a slight explanation before). Well, we shall define these terms properly later but to give a taste here goes a better classification. Why better? **Because, it has examples.**



In the next section, we shall introduce the important concepts associated in understanding Constraint Logic formalism. The main target would be to be able to provide a bulk understanding of the topic, in general.

Constraint Logic: Theory

A constraint graph serves as a model of computation, whose orientation describes the machine's state. What does that mean? Well, a machine in this computational model is an undirected graph with red and blue edges. A configuration of the machine gives directions to the edges, thus is a directed graph. Depending on the nature of the game (further constraints), the computation performed can be deterministic, non-deterministic, alternating etc. The constraint graph then accepts the computation just when the game can be won.

Definition: A *constraint graph* G is a directed graph, with edge weights $\in \{1, 2\}$. An edge is called *red* or *blue* respectively. The *inflow* at each vertex is the sum of the weights on inward-directed edges, and each vertex has a non-negative *minimum inflow* $(= 2)$.

Legal Configuration: $\text{inflow}_i \geq \text{min-inflow}(= 2),\ \forall i \in V(G)$ which serves as the constraint. We can further assume, $\deg_i \leq 3,\ \forall i \in V(G)$ or in other words, G is 3-regular.

Legal Move: A legal move on a constraint graph is the reversal of a single edge, resulting in a legal configuration.

The Game: Reverse a given edge, by executing a sequence of moves. In multiplayer games, each edge is controlled by an individual player and each player has his own goal edge.

Deterministic Game: An unique sequence of reversals is forced.

Bounded Game: Each edge may only reverse once.

Major Theorems

1. **Existence**: The existence of a configuration for a given machine is NP-complete. A modified version would be to prove that Bounded Non-deterministic Constraint Logic is NP-complete.
2. **Reversibility Problem**: Given a constraint graph, can we reverse a specified edge by a sequence of valid moves, is PSPACE-complete.
3. **Reachability Problem**: The reachability problem, that is, given two configurations C_1 and C_2, can we reach C_2 from C_1 by a sequence of valid moves, is PSPACE-complete.

Note: The above two theorems hold for *planar graphs* in which each vertex touches either 3-*blue edges* or 2-*red*-1-*blue edges*.

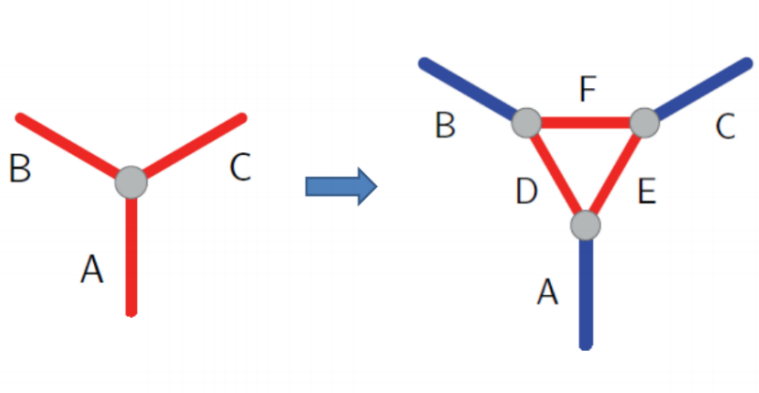
The Gadgets

AND vertex: C may be directed outward iff A and B are both directed inward.

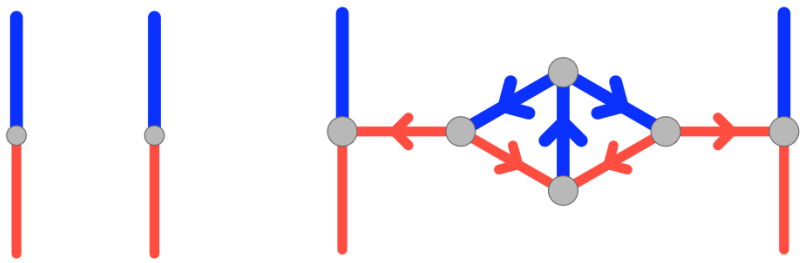
OR vertex: C may be directed outward iff either A or B are directed inward.

SPLIT vertex gives an alternative view of the AND vertex. It takes 2 red edges as outputs and 1 blue edge as input. The outputs can be active only if the input is active.

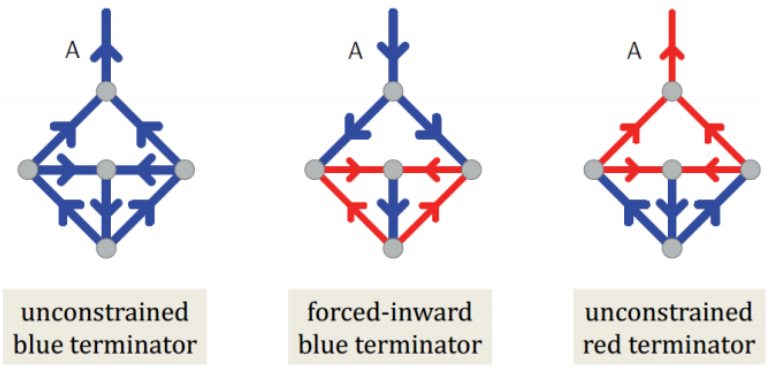
CHOICE vertex: Represented as a node with one red input edge and two red output edges. The idea is that if the input is active, only one of the two outputs can be active, hence the choice. An equivalent construction would use one SPLIT as input and two ANDs as outputs.



RED-BLUE CONVERSION gadget: is used to turn the output of an AND or an OR vertex into the input for an AND or CHOICE vertex. This changes the color of the edge from blue to red.



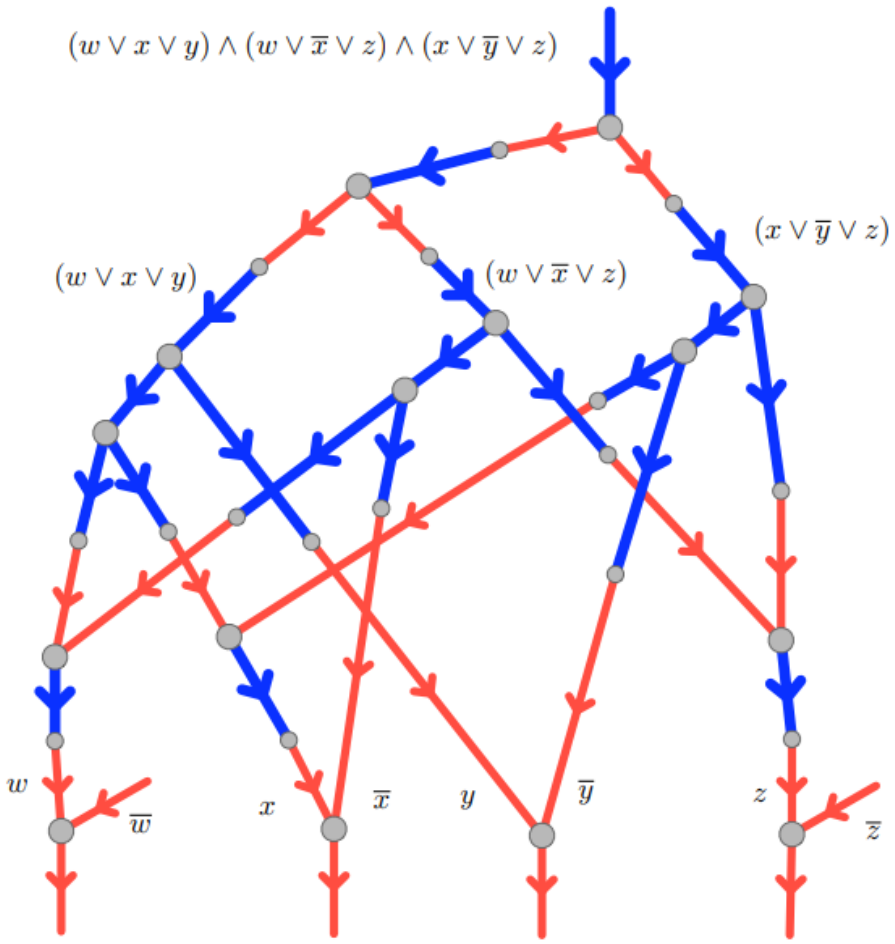
WIRE TERMINATOR gadget: Takes a 1 degree vertex and turns it into a 3 degree vertex. Special cases arise depending on the color and desired orientation of the wire.



Non-Deterministic Constraint Logic

Non-deterministic Constraint Logic serves as the one-player version of Constraint Logic. Since, most of our work on this project deals with Single-player games - especially 2-D Platform video games we shall discuss this first and then move on to simulations and multiplayer games.

The following two proofs serve as the backbone for proving the hardness of puzzles and single player platform video games, which essentially are digital puzzles.



Proof: Bounded NCL is NP-complete

Given an instance of 3-SAT, we construct graph G' as described above. Let F be the corresponding boolean formula. Then, clearly F is satisfiable, the CHOICE vertex edges may be reversed in correspondence with a satisfying assignment, such that the output edge may eventually be reversed. Similarly, if the output edge may be reversed, then a satisfying assignment may be read directly off the CHOICE vertex outputs. Now, $\exists$ an easy poly-time reduction from G' to G where we replace the vertices with 3-*blue-edge* vertex or 2-*red*-1-*blue-edges* vertex. Thus, Bounded NCL is NP-Hard.

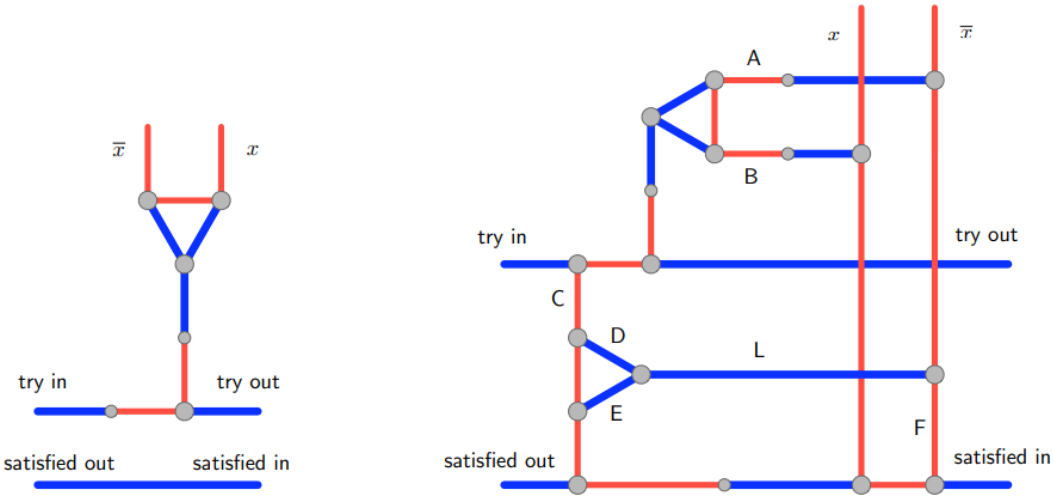
Thus, a proper interpretation of the above proof should be that **Bounded NCL is NP-complete.** Moreover Bounded NCL is NP-complete for planar graphs as well.

This proof serves as the backbone for proving that Bounded Puzzles are NP-Complete (for example, the first version of our CELESTE proof).

Proof: NCL is PSPACE-complete

NCL is in PSPACE because the state of the constraint graph can be described in a linear number of bits, specifying the direction of each edge, and the list of possible moves from any state can be computed in polynomial time.
Thus, we can have an NPSPACE algorithm where we non-deterministically traverse the state space while maintaining only the current state. Now, by Savitch's Theorem ($\mathrm{PSPACE} = \mathrm{NPSPACE}$) for this NPSPCE algorithm there exists a PSPACE algorithm into which the previous can be converted.

Moreover, NCL is also PSPACE-hard because deciding whether an edge may reverse also decides the TQBF problem. For the complete proof we require gadgets for the quantifiers.



The above are the required EXISTENTIAL and UNIVERSAL quantifier gadgets respectively. Since, we have already had an elaborate reduction from TQBF worked out for CELESTE we leave the rest of the proof to the reader.

Games which are not Puzzles

Here, we shall discuss simulations and 2-player games and then quickly move on to looking at the whole picture (genralization) before concluding.

Simulations

One of the most famous simulations is Conway's Game of Life. This game in our formalism is a Unbounded DCL or Deterministic Constraint Graph. Just as Non-deterministic Constraint Logic serves as the one-player version of Constraint Logic, the zero-player version is DCL (as expected).

Let's talk about the results for simulation:

1. *Bounded DCL is P-complete*: This result is quite boring for the simple fact that there are no interesting games that tend to lie in this genre.
2. *DCL is PSPACE-complete*: This result can be used to model Game of Life and provide a simpler proof as to why it is PSPACE-complete.

2-Player Games

> I have come to the personal conclusion that while all artists are not chess players, all chess players are artists.
>
> **– Marcel Duchamp**

On that note, let's talk about chess. Suppose, that our realm in only 8×8 chess with all its classic rules, expect one - the 50 move rule, which we shall ignore here.

So, which class does 8×8 classic chess falls in? Well, it does fall in PSPACE as it takes constant space. But, that is not very interesting is it? So, what about $n \times n$ chess or generalised chess?

Bounded and Unbounded 2-Player Games

We have seen how moving from deterministic to non-deterministic computation creates a new and useful model of computation but what if we add an even larger degree of non-determinism (moving from 1-player to 2-player)? Well, we achieve *alternation*.

This results in the complexity of bounded games being raised to PSPACE-complete from NP-completeness and that of unbounded ones from PSPACE-completeness to EXPTIME-complete.

The two-player version of Constraint Logic is called Two-Player Constraint Logic or 2CL (very innovative). To create different moves for the two players, Black and White, we label each constraint graph edge as either Black or White. *This is independent of the red/blue coloration, which is simply a shorthand for edge weight*. Black (White) is allowed to reverse only. Black (White) edges on his
move. Each player has a target edge he is trying to reverse.

Theorems

1. *Bounded 2CL is PSPACE-complete*.
2. *2CL is EXPTIME-complete*.

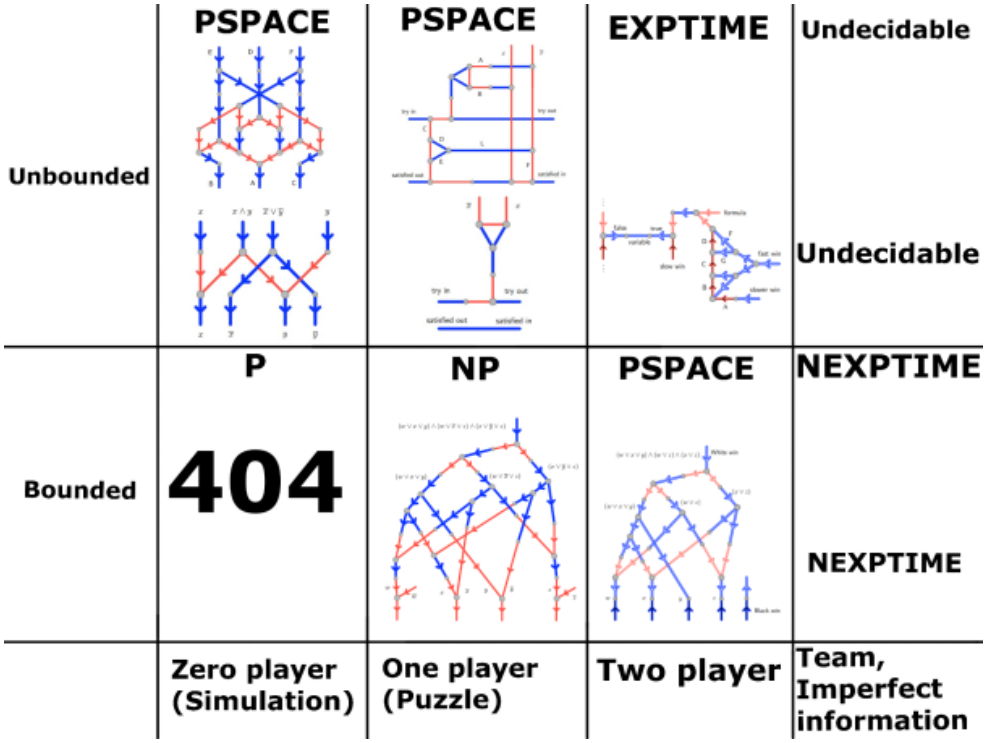
Moreover, just like all previous similar theorems, these hold for Planar Graphs as well.

Interesting Results?

1. Well, it means that generalized versions of chess, go and checkers are provably EXPTIME-complete. A proper reduction from suitable constraint logic graph would do it.
2. For the first time, we have arrived at a class where, $\mathbf{P} \neq \mathbf{EXPTIME}$. Thus, generalised chess is provably intractable, unlike all other games that we have encountered in this project before.

Generalization and Conclusion

Finally, we have got a taste of how beautiful and powerful this new formalism is. Also now, we can further appreciate the several decades of work in Games and Complexity that have resulted in the generalization as follows.



An Analysis of Game Design

Discussion on the Design Aspects of Platformers

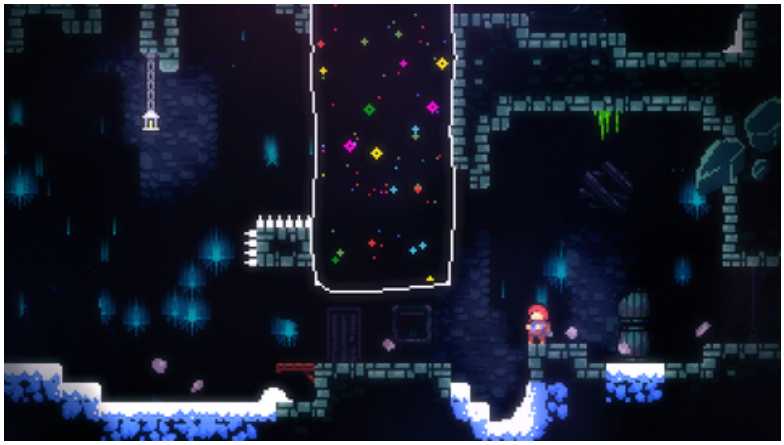
Before talking about how to **generalize hardness results** concerning 2D Platform Games, let's talk about **how games are designed**.

Introduction

In October 1958, Physicist William Higinbotham created `Tennis for two' which is thought to be the first ever video game. The idea was very simple: Two players on either side of the net adjust the angle of their shots and try to hit the ball over the net. Since then, more and more games were designed, some with very unique aspects to them, leading to the current day, where we have a vast ocean of games with tonnes of unique designs.

In this article, we'll be discussing some of the key design aspects of one specific genre, i.e., platformers.

Platformers



Celeste

Platformers are 2D, side-scrolling video games where the player controls one character. The objective of platformers is to reach set checkpoints in each level, usually indicating the end of that level.

Examples

Classics like Super Mario Bros., Super Contra, Sonic the Hedgehog and Celeste illustrate the idea of platformers very clearly. We will use them as `role-models' throughout our discussion.

Structural Analysis

Frequent Rewards

Platformers are usually designed with multiple levels (like in Super Mario Bros.). The idea behind multiple levels is to keep the player motivated to keep completing more and more levels. This reward system at small intervals is usually the safer and most commonly used method of designing platformers.

Continuity

Straying slightly from the previous structure are games like Celeste, where the game is designed in a continuous manner. The rewards for completing `levels' are placed relatively far apart. This kind of approach is usually backed with a very strong storyline to keep the player motivated instead of a frequent reward system. In Celeste, specifically, each 'scene' of the game itself is very challenging and hence sort of acts like a level in itself. Overcoming the challenge in each scene acts as an alternative to a level-by-level reward system even though no explicit reward is presented to the player. Along with this, Celeste is backed with a beautiful storyline that keeps the players more than just engaged.

Rewarding with a high score

Apart from this, there are also infinite side-scrollers where the objective is to maximize your score. The motivation for the player to keep playing is to beat his/her own score each time or to beat scores of other players.

Game Over

Most games have a very basic storyline to the least. The game usually ends when the story comes to a conclusive end. The player usually completes the bigger objective that was known from the start (in most cases). For example, in Super Mario Bros., Mario wants to rescue Princess Peach. In Celeste, Madeline wants to climb to the top of a mountain. This sense of closure makes the player feel positive about the game and is possibly the safest strategy that game designers use in hopes that the player leaves positive reviews about the game and to make an overall good impression of the game.

As opposed to this, some designers like to make their video game open-ended. This is not a safe strategy at all as it can lead to dissatisfaction among the players. Then why make it open-ended at all? An open-ended game is, in most cases, backed with an absolutely great story/great gameplay. It works only when the game delivers enough to actually make the player think about the open ending. These kind of thoughts on the game are a sign of `greatness of the game' in the video game community that all game designers aspire for. Note that the kind of open-endings we discussed are different from cliffhangers. An open-end is where the player is left uncertain as to what happened next. The player is never intended to know and the end is left open to interpretation. A cliffhanger, on the other hand is used to create a feeling of suspense, where the mystery element is known to be revealed in the future. This is usually done to create anticipation for the next video game in the series.

Analysis of the game's elements

A games element include enemies/obstacles, power-ups and other things that the player can interact with. We'll discuss some platformer-specific elements.

Ground

This may seem like an obvious aspect to most of the games, but it needs to be discussed as it is the most important element in any platformer. As the name suggests, platformers are based on the fact that the player interacts with the platform (ground). The key aspect of a platformer that separates it from other genres is the placement of the ground.
In most games, the ground is only used as a background factor. It doesn't usually play any role other than having something to stand on.
In platformers, however, the ground's placement itself poses a challenge to the player. The ground itself becomes one of the obstacles for the player. In Super Mario Bros., for example, there are gaps in the ground that Mario has to jump over in order to not die. The ground is also placed in other interesting ways to make getting things like power-ups harder. It is a designer's job to focus primarily on the ground's placement, before the other elements.

Obstacles

Enemies or other forms of obstacles are also an integral part of platformers. The difficulty of a level primarily depends on the way these obstacles are placed and also on how the obstacles interact with the player. The choice of these obstacles and their placement should be done extremely carefully as putting strong ones at the very beginning of the game could lead the player to think that the game is too hard and be demotivated. A weak set of obstacles placed a long way from the beginning could make the player feel the opposite way and would lead to dissatisfaction. The safest strategy is to place them in an increasing order of difficulty. The obstacles can be of various types, some are destroy-able like the enemies in Super Mario, some are not like the spikes in Celeste.

The Player

The player is completely controlled by the player and hence it is important for a game designer what freedoms the player needs to be given. In most platformers, the player is given the freedom of moving left-right and jumping. Here, too, the kind of jump the player makes is very important to the player as it determines the player's mastery of control throughout the entire game. Celeste, in particular, does an amazing job in the jump aspect. Different games also give other freedoms like the ability to fire a gun in Super Contra and shoot fireballs in Super Mario. Celeste gives the ability to dash and climb walls. Sonic gives the ability to roll. It is impossible to list down the possible freedoms as it is upto the designer's creativity.

Death

Deciding how the player dies is also extremely important as this can be the difference between an extremely hard game and an easy one. In games like Super Mario, a very beautiful system is implemented. You die immediately once you hit an enemy. But this can be avoided if you have the power of a mushroom. So this makes the game hard but if you play good enough to conserve a mushroom, then you're at low risk. Also in Super Mario, you have a limit to the number of times you can die, making the game very challenging. In Super Contra, a health system is used. In Celeste, you die immediately when you hit an obstacle. No exceptions. But here, you can try a level infinite number of times. All of these different systems are designed very well, giving the player just the right amount of challenge that fits well with the game.

Conclusion

We discussed some key aspects of platformers and why they are designed the way they are. There are a lot of more aspects that can be talked about but I shall end the discussion here as these are some general aspects that apply to most platformers.

Platformers: Introduction to Generalizations

The analysis of 2-D platform games are of great mathematical and computational interest in the recent times. Many of the traditional games and puzzles have undergone rigorous mathematical analysis over the last century and are proved to be conditionally hard (whether $P \neq NP$) while others are provably intractable (as discussed with respect to chess). A similar approach to video games, in particular 2-D platform games, has been fairly recent and the complexity of many video games are yet to be studied and remains largely unexplored.

In essence, 2-D Platformers are puzzles. As a result they are often complete for either NP or PSPACE. It might even be argued that this factor allows such games to be "humanly interesting" to play.

The completeness analyses that we are interested about are often found un-applicable for modern games, especially non-platform games. This is because most of modern age games use Turing-equivalent scripting languages that allow designers to have *undecidable puzzles* as a part of the game play.

NP-Completeness

> In case of NP-complete games, we have levels whose solution demands some degree of *ingenuity*, but such levels are usually solved within a polynomial number of *manipulations*, and the challenge is merely to find them.

PSPACE-Completeness

> In contrast, the additional complexity of a PSPACE-complete game seems to reside in the presence of levels whose solution requires an exponential number of manipulations, and this may be perceived as a nuisance by the player, as it makes for tediously long playing
> sessions.

On Platformers

We approach the analysis of 2-D platform games by observing that all such games have certain common features. Any 2-D platform game typically involves a sequential set of puzzles (*levels*). At each level, the player must manipulate his/her avatar to the predesignated end point, performing tasks along the way. The avatar is controlled by the player using a limited simple commands (step, jump, crouch etc.). Further, the avatar is affected by some form of gravity, as a consequence of which maximum jump height is limited and avatar could potentially fall from a height. Each level in such games are represented by a 2-D map representing a vertical slice of a virtual world which usually consists of horizontal platforms, hence the name *platform game*.

In addition, many of these games have certain common features, as enlisted below:

- long fall: The height of the longest safe fall (that does not hurt the avatar), which is taken to be larger than the maximum jump height.
- opening doors: The game world may contain a variable number off-doors and suitable mechanisms to open them.
- closing doors: The game world contains a mechanism to close doors, and a way to force the player to trigger such a mechanism.
- collecting items: The game world contains items that must be collected.
- enemies: The game world contains enemy characters that must be killed or avoided in order to solve the level.
- time limit: The given level of the game must be completed within the given time limit.

These features allow for several generalizations or *metatheorems* concerning these platformers which help us understanding and approaching the problem (*decision problem associated*) with greater abstraction.

While certain or combination of features are easy from an algorithmic point, other features, or certain combinations of thereof, are hard to manipulate. Thus having such a feature within the game essentially imply that the game itself is hard to solve, regardless of other details.

For any given game, these meta-theorems can be adjusted according to details of the particular game. In the next section we state the several metatheorems as well as some relevant

conjectures.

Meta-theorems

1. *A 2-D platform game where the levels are constant and there is **no time limit** is in **P**, even if the collecting items feature is present.*
2. *Any game exhibiting both **location traversal** (with or without a starting location or an exit location) and **single-use paths** is **NP-hard**.*
3. *Any 2-D platform game that exhibits the features l**ong fall and opening doors** is **NP-hard**.*
4. *Any 2-D platform game that exhibits the features **long fall, opening doors and closing doors** is **PSPACE-hard**.*
5. *A game is **NP-hard** if either of the following holds:*

   *(a) The game features **collectible** tokens, toll roads, and location traversal.*
   *(b) The game features **cumulative** tokens, toll roads, and location traversal.*
   *(c) The game features **collectible cumulative** tokens, toll roads, and the avatar has to reach an exit location.*
6. *A game is **NP-hard** if it contains **doors and one-way paths**, and either of the following holds:*

   *(a) The game features **collectible** keys and location traversal.*
   *(b) The game features **cumulative** keys and location traversal.*
   *(c) The game features **collectible cumulative** keys and the avatar has to reach an exit location.*
7. *If a game features **doors and pressure plates**, and the avatar has to reach an exit location in order to win, then:*

   *(a) Even if **no door** can be **closed by a pressure plate**, and if crossovers are allowed, then the game is **P-hard**.*

   *(b) Even if **no two pressure plates** control the same door, the game is **NP-hard**.*

   *(c) If **each** door may be controlled by **two pressure plates**, then the game is **PSPACE-hard**.*
8. *If a game features **doors and k-buttons**, and the avatar has to reach an exit location in order to win, then:*

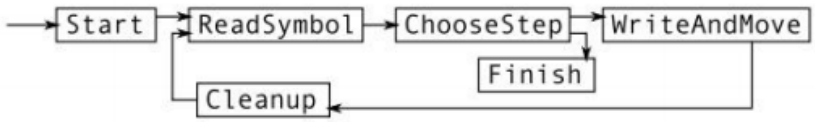
   *(a) If **k > 1,** and crossovers are allowed, then the game is **P-hard.***
   *(b) If **k > 2,** then the game is **NP-hard.***
   *(c) If **k > 3,** then the game is **PSPACE-hard.***
9. *There exists an NP-complete game featuring doors and 2-buttons in which the avatar has to reach an exit location. (One that we end up proving in CELESTE).*

Applications

These results can be use to directly analyse the complexity of several 2-D platform games. For instance, Commander Keen, Crystal Caves, Secret Agent and Bio-Menace can be shown to be **NP-hard** due to presence of switches for activation of moving platforms. Similarly, Jill of the Jungle
and Hocus Pocus are also **NP-hard** due to the presence of switches that toggle walls, so are

Crash Bandicoot and Jazz Jackrabbit 2, due to crates on breaking which sets of new floor tiles are activated. Duke Nukem with the traditionally observed 4 key-cards will be **P**, while the generalized case with n-keys will be **NP-hard**.

The game *Prince of Persia* is quite interesting. An instance of the word problem for linear bound automata (WordLBA) is reduced to a instance of the game. The general layout of an instance of the game Prince of Persia is as shown below:



The reduction allows us to conclude that the game *Prince of Persia* is **PSPACE-hard**. Further, it is known that the game is in **NPSPACE**. These results, along with Savitch's theorem which states that **NPSPACE = PSPACE**, can be used to show that the game Prince of Persia belongs to **PSPACE-complete**.

The case of the generalized version of the game *Lemmings* is also quite interesting. A proof was given by Cormode in his 2004 paper in which *Lemmings* is shown to belong to **NP**. However, the author of the original article on this topic argues against the proof by noting that the setup used Cormode's proof with an exhaustive trial and error algorithm fails if the number of theoretically possible configurations is not polynomial in the input size. Using the earlier given meta-theorems, the author goes on to show the sequence of instances $\{I_n\}_{n=1}^{\infty}$ is valid for the game *Lemmings* and a counter example to the Cormode's proof. Further, this result is shown to be true for a simple (non generalized version) of the game *Lemming* too. The author goes on to conclude that while Concorde's proof of the game being **NP-hard** is valid, we cannot comment whether the problem belongs **NP** or not.

Part - II: Conclusion

> Gaming is a hard job, but someone has to do it.
>
> **- Giovanni Vigilietta**

Thus, with the beautiful quote from the author of the original paper that motivates this section, we conclude Part II of our Project. Hopefully, through this endeavor of ours, we have managed to put out the beauty of Complexity Theory through games and their reduction proofs, as well as, to show that understanding games often result in marvelous breakthroughs in not only Computer Science but several other fields as well. Artificial Intelligence serves as a breathing example for this.