1 What is the point of this document?

This is supposed to serve as a reference guide for C63. When I took the course over the fall 2022 with Nick Cheng, I found it difficult to keep track of all the concepts and how they tie together. I found C63 to require EXTREME competence at the early material to continue succeeding in the course. I.e. If you don't understand the intricacies of turing machines (configurations), you will struggle the P, NP, and PSPACE language classes. Additionally, C63 requires a **HUGE** mastery of B36 material. Many, many people (i.e. me) lost valuable points on the midterms and finals because of lacking B36 skills. I did WELL in B36 too. Additionally, the book we used (Sipser) was overly long winded and did a poor job of connecting the ideas it contains together in a coherent story. The point of this document is to prove most of the claims in the course and to document its proof techniques, that way it can serve as a reference for C63 topics. Sipser also leaves rigorous definitions as exercises to the reader, I tried not to. Feel free to message me over discord for corrections or for comments. I'd also be happy to help people out with the course.

1.1 Credits:

All the ideas in this document other than my advice come from Michael Sisper and Nick Cheng, most of the proofs are copied from Sisper's book.

1.2 What B36 skills should you review?

- Languages in b36 were defined using D/NFSA's, and PDA's, in this course, languages are defined similarly on turing machines, i.e. $L = \mathcal{L}(M) = \{\langle w \rangle \mid \text{M is a Turing Machine, } w \in \Sigma^\star, \text{ M accepts w}\}$. In plain english, $\mathcal{L}(M)$ is all the strings M accepts.
- When you are learning about co-NP, make **ABSOLUTELY SURE** you are rock solid on your ability to take the complement of a language, including languages with implications
- Make sure you understand how to take the negation of a qualified boolean formula like $\forall x_1 \exists x_2 [(x_1 \vee x_2) \wedge (x_1 \rightarrow x_2)]$, also how to translate english sentences into boolean formulas. Nick Cheng has some additional notes on this topic from B36.

1.3 What is the point of this course?

CSCC63 is fundamentally about classifying languages. The primary language classes are, Decidable, Recognisable, Co-Recognisable, P, NP, co-NP, NP-Complete, PSPACE, and NSPACE.

1.4 Notation

$\langle x \rangle$ denotes "The encoding of x". Encodings translate mathematical objects like sets into strings which can be fed as inputs to turing machines, or contained in languages as follows:
$L = \{\langle M, w \rangle \mid \text{M is a Turing Machine, } w \in \Sigma^\star, \text{ After running M on w, M accepts }\}$

2 What is a turing machine?

A turing machine is a model of computation. In C63, a Turing Machine will stand in for "Any algorithm". Any algorithm can be written as a turing machine, and in similar* running time and similar* memory usage. These are the beefiest automatons possible, no other automaton is more powerful than a turing machine, and the turing machine surpasses the DFSA, NFSA, and PDA in capabilities. This is because the turing machine has access to an infinite, addressable memory space, unlike PDA's which are only capable of addressing memory sequentially through the stack.

2.1 What is it, like specifically though

A Turing Machine $M = \langle Q, \Gamma, b, \Sigma, \delta, q_0, F \rangle$, with **FINITE** states Q, A **FINITE** set of Symbols which can appear on the tape Γ, an input alphabet $\Sigma \subset \Gamma$, a blank symbol $b \in \Gamma$, an initial state $q_0 \in Q$, a set of final states $F \subseteq Q$, and a **FINITE** transition function δ, which encodes the transition labels. The diagrams which describe turing machines encode δ.

2.1.1 A very important note

Turing Machines, by design, are finite machines which encode finite algorithm descriptions. This means that the set of turing machines can be iterated through. Not only that, but turing machine encodings ($\langle Q, \Gamma, b, \Sigma, \delta, q_0, F \rangle$) can be fed to **OTHER TURING MACHINES**. This means we can write algorithms which process other algorithms. This is the fundamental proof tool for the course.

2.1.2 Another very important note

Turing machines have 2 possible final states, accepting and rejecting. Unfortunately, turing machines also are able to loop infinitely. It will be proven later in these notes that there is **NO GENERAL ALGORITHM** for determining whether or not a given turing machine will loop on a given input.

Part I
How to determine if languages are Recognisable, Decidable, or Co-recognisable.

3 The first language classifications: Recognisable and Decidable languages

3.1 The definition of recognisability using Enumerators

3.1.1 What's an enumerator?

An enumerator is a special kind of turing machine with two tapes, the work tape, and the print tape. There is also a specialized "print" state which will "print" the contents of the print tape to an attached theoretical printer. Using this definition, let E be an enumerator, $\mathcal{L}(E) = \{x|\text{x is printed by E at some point in its runtime}\}$. Note, enumerators can halt or loop forever, this does show up in exam questions, enumators may also repeat inputs, this can be useful for lazy algorithm writers.

3.1.2 The actual definition

A language L is recognisable if and only if there exists an enumerator E st $\mathcal{L}(E) = L$. In english, this is written as, "E enumerates L"

3.1.3 Steps for proving a language is recognisable using enumerators

1. Describe an enumerator E
2. Prove that every $x \in L$ is in $\mathcal{L}(E)$
3. Prove that every $x \notin L$ is NOT in $\mathcal{L}(E)$

3.2 The definition of recognisability using regular turing machines

Given a regular turing machine D and an arbitrary language L, D recognises L if and only if

1. If $w \in L$, D accepts w
2. If $w \notin L$, D does not accept w (D may reject **OR LOOP ON** w)

A recognisable language per this definition is a language where some turing machine recognises it.

3.3 The definition of co-recognisability

Co-recognisable languages are languages who's complements are recognisable. I.e. L is co-recognisable if and only if $\overline{L}$ is recognisable

3.3.1 Proving a language L is co-recognisable

1. **CAREFULLY, MATICULOUSLY, and EXACTLY** take the language complement of L to obtain $\overline{L}$, this is MUCH HARDER than you think it is
2. Write a recogniser for $\overline{L}$ and prove it is a recogniser using any of the techniques above.

3.4 The definition of decidability

Decidable languages are langauges which are **BOTH** recognisable and co-recognisable. Proving a language L is decidable requires using both proof techniques for recognisability and co-recognisability.

4 Showing that languages are NOT recognisable or decidable using MAPPING REDUCTIONS!

The previous section detailed how to prove that a language is recognisable, the issue is that we have no techniques to show languages are NOT recognisable. The techique built in this chapter is called "The mapping reduction". This technique is extremely subtle and confusing, with AWFUL NOTATION which I have a whole rant about later in this document. Mapping reductions require us to find un-recognisable languages using other techniques first, which we must do before defining mapping reductions. To find un-recognisable languages, we must first find an un-decidable language.

4.1 The first undecidable language A_{TM}

$A_{TM} = \{\langle M, w\rangle \mid \text{M is a TM and M accepts w}\}$

4.1.1 Showing A_{TM} is recognisable

U="On input $\langle M, w\rangle$, where M is a TM and w is a string:

1. Simulate M on input w
2. If M ever enters its accept state, accepts, else if M enters the reject state, reject

For completeness, I will also prove that U is a recogniser for A_{TM}.
Given that $\langle M, w\rangle \in A_{TM}$, we know M accepts w. Therefore, U will accept $\langle M, w\rangle$
Given that $\langle M, w\rangle \notin A_{TM}$, we know M does not accept w, therefore U will not accept $\langle M, w\rangle$
Therefore, $\langle M, w\rangle \in A_{TM} \iff \langle M, w\rangle \in \mathcal{L}(U)$, showing U recognises A_{TM}

4.1.2 Showing A_{TM} is NOT decidable

Assume that A_{TM} is decidable for the purposes of contradiction.
Since A_{TM} is decidable, there must exist a turing machine H which decides A_{TM}.
Now, construct a TM D as follows:

D="On input $\langle M\rangle$, where M is a TM:

1. Run H on input $\langle M, \langle M\rangle\rangle$
2. Output the opposite of what H outputs. I.e. if H acc then rej, if H rej, then acc

D is constructed to behave as follows:
$$D(\langle M\rangle) = \begin{cases} \text{accept} & \text{If M does not accept } \langle M\rangle \\ \text{reject} & \text{If M accepts } \langle M\rangle \end{cases}$$
Now, we run D on itself, to get
$$D(\langle D\rangle) = \begin{cases} \text{accept} & \text{If D rejects } \langle D\rangle \\ \text{reject} & \text{If D accepts } \langle D\rangle \end{cases}$$
Therefore, we have derived a turing machine where running $\langle D\rangle$ on D accepts if D rejects $\langle D\rangle$, and D rejects if D accepts $\langle D\rangle$. This is contradictory behavior, therefore, no such TM D can exist, therefore, A_{TM} is NOT decidable.

4.1.3 Final note:

Since A_{TM} is recognisable, and A_{TM} is not decidable, $\overline{A_{TM}}$ must be unrecognisable

4.2 Another undecidable language: HALT

HALT=$\{\langle M, w\rangle \mid \text{M is a TM and M halts on input w}\}$

4.2.1 Showing HALT is undecidable

Assume for purposes of contradiction that TM R decides HALT, then we can construct a TM S to decide A_{TM}:

S= "On input $\langle M, w\rangle$, an encoding of a TM M and a string w:

1. Run TM R on input $\langle M, w\rangle$
2. If R rejects, reject
3. If R accepts, simulate M on w until it halts.
4. If M has accepted, accept; if M has rejected, reject."

Now, the proof that $\mathcal{L}(S) = A_{TM}$.
If $\langle M, w\rangle \in A_{TM}$, then M accepts w. Therefore, R will accept $\langle M, w\rangle$, and simulating M on w until halt will result in M accepting w, therefore, $\langle M, w\rangle \in \mathcal{L}(S)$
If $\langle M, w\rangle \notin A_{TM}$, then M rejects or loops on w
1. If M rejects w, then R will accept $\langle M, w\rangle$, and simulating M on w until halt will result in a rejection, so S rejects, implying $\langle M, w\rangle \notin S$
2. If M loops on w, R will reject $\langle M, w\rangle$, resulting in S rejecting $\langle M, w\rangle$, implying $\langle M, w\rangle \notin S$
Therefore, $\mathcal{L}(S) = A_{TM}$, implying S is a decider for A_{TM}. This implies that A_{TM} is decidable. Therefore, by contraction, HALT is undecidable

4.2.2 Showing HALT is recognisable

Here is a recogniser for HALT:

D= "On input $\langle M, w\rangle$:

1. Run M on w
2. accept

Proof that D is a recogniser for HALT.
If $\langle M, w\rangle \in$ HALT, then M halts on w, which implies that step 2 of D is reached, and D accepts, therefore, $\langle M, w\rangle \in \mathcal{L}(D)$
If $\langle M, w\rangle \notin$ HALT, then M loops on w, which implies that step 2 is never reached, so D does not accept, therefore, $\langle M, w\rangle \notin \mathcal{L}(D)$
Therefore, $\mathcal{L}(D) =$HALT

4.2.3 Conclusion

HALT is recognisable and co-unrecognisable.

4.3 The definition of a mapping reduction,

Language A is mapping reducable to language B if there is some turing machine F st $w \in A \iff F(w) \in B$. This is written as $A \leq_m B$. Be very careful with this notation, it is extremely confusing.
Note: This definition relies on the concept of using turing machines as functions. $F(w)$ is defined as "Whatever is left on the tape after F halts on w"
Note 2: F must not loop on any inputs

4.4 How can mapping reductions be used to prove that languages are not decidable or recognisable?

4.4.1 Proof that "If $A \leq_m B$ and B is turing-recongisable, then A is turing-recognisable"

Let M recognise B, here is a recogniser for A:

N= "On input w:

1. Compute $w' = F(w)$
2. Run M on input w'
3. If M accepts, accept, else reject

Here is a proof that N recognises A:
If $w \in A$, then $w' = F(w) \in B$ because of the mapping reduction. Therefore, M accepts w', and N accepts, therefore $w \in \mathcal{L}(N)$
If $w \notin A$, then $w' = F(w) \notin B$ because of the mapping reduction. Therefore, M does not accept w', and N does not accept, therefore $w \notin \mathcal{L}(N)$
Therefore, $\mathcal{L}(N) = A$, showing that A is recognisable.

4.4.2 Corollary: If $A \leq_m B$ and A is not Turing-recognisable, then B is not Turing-recognisable

We know that if $A \leq_m B$ and B is turing-recognisable, then A is turing-recognisable. Taking the contraposition, we get
"If A is not turing-recognisable, then $A \not\leq_m B$ or B is not turing-recognisable"
However, we know $A \leq_m B$ and that A is not turing-recognisable by assumption, therefore, we know that B must not be turing-recognisable using the contraposition'd statement.
Therefore, If $A \leq_m B$ and A is not turing-recognisable, then B is not turing-recognisable

4.4.3 How to use mapping reductions to prove that languages are not decidable or recognisable:

Nick Cheng additional notes

Part II
Classifying DECIDABLE langauges as P or NP

The previous part focused on which lanagues (i.e. algorithms) are can actually be run on computers. This part focuses on what time those algorithms will take to compute. Note: henceforth, we will **ONLY DEAL WITH DECIDABLE LANGUAGES. UNDECIDABLE LANGUAGES HAVE NO TIME COMPLEXITY AS P AND NP ARE SUBSETS OF THE DECIDABLE LANGUAGES.**

5 What is P?

P is the class of languages which have polynomial time solvers

5.1 What's a polytime solver?

A polytime solver for language L is a decider S, which runs in $\mathcal{O}\left(n^k\right)$ time for some constant k, where $n = |w|$, forall $w \in L$.

5.2 In english pls?

If L is in P, then there is some decider for L which runs in polynomial time.

6 What is NP?

NP is a class of languages that have polynomial time verifiers

6.1 Verifiers

A verifier for a language A is an algorithm V where
$A = \{w|$ V accepts $\langle w, c\rangle$for some string c $\}$
The time it takes to run a verifier is measured in terms of the length of w. In this system, c is the certificate

6.2 The Big Cheng 5: Proving a problem in NP the right way!

1. Show that Q is a decision problem. This is usually quite obvious from the way Q's question is asked, so merely stating that Q is a decision problem will often be sufficient.
2. Describe what a certificate would be for a Yes instance of Q. Most of the time this is also quite obvious from the way Q's question is asked. If the question asks, "is there a blah?", (i.e., it essentially asks for the existence of something), then usually the certificate is simply the blah (the thing about whose existence is asked by the question).
3. Explain why the certificate is polysize with respect to the input size. Frequently the certificate is no larger than the input. In such cases the certificate size is at most linear with respect to the input size.
4. Describe what a verification algorithm would do. The certificate is commonly something that satisfies some set of criteria. So the algorithm needs to check (or verify) that these criteria are satisfied.
5. Explain why the verification algorithm runs in polytime. Typically we would only need to give running times for the checks that the algorithm performs

This is the official, immutable way to show that a problem is in NP, this is the only way to do it, and there are no other ways. We will henceforth follow them religiously.

7 What is NP closed under?

7.1 Prove that NP is closed under union, intersection, concaterntaion, and Kleene star

7.1.1 Let $A, B \in$ NP, show $A \cup B \in$ NP

If A,B are in NP, then A and B have polynomial time verifiers verifyA and verifyB. Let L=$A \cup B$. Now, we wish to argue that $L \in$ NP

1. Since A and B are decision questions, asking if an input is in L must also be a decision question
2. A certificate would be $\langle c_1, c_2 \rangle$ where c_1 is a certificate for verifyA, and c_2 is a certificate for verifyB
3. The certificate would be polysize with respect to input size of L since c_1 and c_2 are polysize with respect to the inputs to A and B.
4. A verification algorithm would pass the input to verifyA and verifyB with the appropriate certificates. If either verifyA or verifyB returns true, then return true, else false
5. Our algorithm is polytime since it only runs two polytime algorithms, verifyA and verifyB

7.1.2 WTS: Let $A, B \in$ NP, show $A \cap B \in$ NP

If A,B are in NP, then A and B have polynomial time verifiers verifyA and verifyB. Let L=$A \cap B$. Now, we wish to argue that $L \in$ NP

1. Since A and B are decision questions, asking if an input is in L must also be a decision question
2. A certificate would be $\langle c_1, c_2 \rangle$ where c_1 is a certificate for verifyA, and c_2 is a certificate for verifyB
3. The certificate would be polysize with respect to input size of L since c_1 and c_2 are polysize with respect to the inputs to A and B.
4. A verification algorithm would pass the input to verifyA and verifyB with the appropriate certificates. If verifyA and verifyB returns true, then return true, else false
5. Our algorithm is polytime since it only runs two polytime algorithms, verifyA and verifyB

7.1.3 WTS: Let $A, B \in$ NP, show $AB \in$ NP

If A,B are in NP, then A and B have polynomial time verifiers verifyA and verifyB. Let L=AB. Now, we wish to argue that $L \in$ NP

1. Since A and B are decision questions, asking if an input is in L must also be a decision question
2. A certificate would be $\langle c_1, c_2 \rangle$ where c_1 is a certificate for verifyA, and c_2 is a certificate for verifyB
3. The certificate would be polysize with respect to input size of L since c_1 and c_2 are polysize with respect to the inputs to A and B.
4. VerifyL
5. If n=$|w|$, then, worst case, verifyA will be run n times, and verifyB will be run n times. This is n times some polynomial, which itself, is a polynomial. Therefore, this algorithm runs in polytime.

   VerifyL$\langle w, \langle c_1, c_2 \rangle \rangle$:

   for $i = 0 \rightarrow |w|$:

   if verifyA$(w[0..i], c_1)$ and verifyB$(w[i..], c_2)$:

   return true;

   return false;

7.1.4 WTS: Let $A \in$ NP, show $A^\star \in$ NP

If A is in NP, then A has a polynomial time verifier verifyA. Let L=$A^\star$. Now, we wish to argue that $L \in$ NP

1. This is a decision problem
2. A certificate $\langle c \rangle$ would be the same as a certificate to A
3. The certificate is polysize with respect to input size since its polysize with respect to inputs to A, which are less than or equal to inputs to L
4. A verifier would iterate through the input, running verifyA on $\langle w\,[0..i]\,, c \rangle$. If verifyA returns true, return true, else false
5. This is polytime with respect to $|w| = n$, since the input is of size n, and verifyA, which is polynomial, will get run n times, which is polytime

8 NP-COMPLETENESS

NP-Complete problems are problems that are both in NP and NP-hard. An NP-Hard problem is a problem that is at least as hard as all other NP-hard problems. Putting these toegether, if one NP-complete problem has a polytime solver, they all do. But, if one NP-complete problem can be shown to have no polytime solver, none of them do

8.1 The grandfather of NP-Complete problems: SAT

SAT = $\{\langle\phi\rangle \,|\phi$ is a satisfiable boolean formula$\}$
A satisfiable boolean formula is a boolean formula such that some assignment of variables makes the formula output true. Truth assignments are often labeled τ, and assigning the variables in ϕ with the assignments in τ is $\tau(\phi)$
This proof is WAY to big to put here

9 Polytime reductions

Language A is polytime reducible to language B, $A \leq_p B$ if a TOTAL, POLYTIME, and COMPUTABLE function $f : \Sigma^\star \to \Sigma^\star$ exists with the property that $w \in A \iff f(w) \in B$.
So to reduce A to B, create a function which maps yes instances of A to yes instances of B, and no instances of A to no instances of B, this is the contrapositive.

9.1 How do polytime reductions relate to time complexity?

If $A \leq_p B$, or A polytime reduces to B, and B is in P, then, here's a polytime solver for A

solveA$\langle w\rangle$:

$w_b = f(w)$
return solveB(w_b)

Since f is a polytime computable total function that maps yes instances of A to yes instances of B, and solveB is polytime, if solveB accepts, then w must be in A, and if solveB rejects, w must NOT be in A. Additionally, since B and f are polytime computable, solveA must be polytime computable. This is the core of polytime computability

9.2 Polytime reductions and it's incomprehensible notation

$\leq_p$ is one of the most confusing and poorly implemented operators I have ever seen. Typically, symbols notate a relation between two objects. The "justification" for $\leq_p$ is that if $A \leq_p B$, A is "Less than or equal to B" in terms of hardness. This is complete gibberish. This is an "operator" which communicates the thing one can prove using the existence of a relation between two objects. Imagine if we wrote all notation this way! There would be COMPLETE ANARCHY!!!! Not only that, but this convention encourages un-mathematical, un-creative, instrumentalist applications of mathematical reasoning. This is wholly unfit for a theoretical course.

9.2.1 How to remember this difficult notation

Rule 1: read right to left: if $A \leq_p B$, then a polytime reduction FROM A, TO B exists, I.e. inputs in A, go TO, B. The english phraseology gets this exactly correct. A polytime reduction goes from A to B, and the inputs of A, go TO B.

9.3 Polytime reduction definition of NP-complete

Using our new approach, we can define NP-completeness to be
"A language B is NP-complete if it satisfies two conditions:
1. B is in NP
2. if A$\in$NP, then $A \leq_p B$ for any A in NP
This works because if B has a polytime solver, then A must have a polytime solver. This means that B's solvability in polynomial time makes a claim about every single NP-complete problem's solvability in polytime, which is the point of the NP-complete class

10 The first reduction: SAT$\leq_p$3SAT

PROVE 3SAT is in NP!
3SAT = $\{\phi|\phi$ is a 3-cnf formula which is satisfiable$\}$
A 3-cnf formula, or "conjunctive normal form formula with 3 variables in each clause" is composed of clauses as follows: $(x \vee y \vee \neg x) \wedge (z \vee p \vee \neg x) \wedge \cdots$. Notably, these formulas have no constants. An easy way to remember is that cnf formulas are "sad" ($^\odot\vee^\odot$)
WTS: SAT$\leq_p$3SAT. Showing SAT Polytime reduces to 3SAT will show that if 3SAT is in P, then SAT is in P. This proves that 3SAT is NP-complete
The details of this reduction are not super important, since finding an algorithm to take any formula composed of literals, ands, or's and nots and transforming it into a 3-cnf formula is very boring and not required knowledge. But an appropriate algorithm would have the following passes:
1. Transform literals into tautologies, $\neg x \vee x = 1$, $\neg x \wedge x = 0$
2. Use de morgan's law to make all nots go directly in front of variables
3. probably many others

Intuitively, such an algorithm makes sense since we are used to transforming logical statements into other equivalent statements, since this was tested in a67 and many other courses.
By the principle of arm waiving, we now have a function $f(\phi) \rightarrow \phi'$st if ϕ is a boolean formula, ϕ' is a 3-cnf formula. and ϕ is equivalent to ϕ'.
Now, for the proof:
If $\phi \in$ SAT, then ϕ' is satisfiable, which implies since it is 3-cnf, it is in 3SAT
(using the contraposition)
If $\phi \notin$ SAT, then ϕ' is not satisfiable, which implies it is not in 3SAT
QED

11 The first REAL reduction: 3SAT$\leq_p$CLIQUE, Showing CLIQUE is NP-complete

CLIQUE = $\{\langle G, k\rangle \mid$ G is a graph which contains a clique of size k$\}$

11.1 Proof that CLIQUE$\in$NP

Here's a polytime verifier for CLIQUE:
Let the certificate C be a set of vertices that form a clique, and let G be a graph

VerifyCLIQUE$\langle\langle G, k\rangle, C\rangle$:

if $|C| < k$:

return false

for each vertex in C:

check that it connects to every other vertex in c
if it doesn't return false

return true

Worst case, G=C, so at worst, this algorithm runs in n^2 time. This is sufficient justification to show that CLIQUE$\in$NP

11.2 Proof that CLIQUE is NP-complete

A clique is a set of vertices where each vertex in the clique is connected to every other vertex in the clique with an edge.
The way we will go about reducing these is to come up with some equivalent structure to clauses, and conjunctions in graphs, often called widgets or gadgets. Since some have argued that graphs are a "universal" data structure capable of storing all other data structures, this should be easy!
The steps to make this construction go as follows, Given 3-cnf formula ϕ, output G and k as follows:
1. For each variable in ϕ, add a corresponding vertex in Gnamed after the variable
2. Connect every vertex to every other vertex in G with an edge
3. Remove all edges between "contradictory" vertexes like $\neg x$ and x
4. Remove all edges between vertexes that appear in the same clause (so if $(c_1 \vee c_2 \vee c_3)$ appears in ϕ, no edge should exist from $c_1$to c_2 to c_3)
5. set k to be the number of clauses in ϕ
Now, to argue that $\langle\phi\rangle \in$ 3SAT $\iff \langle G, k\rangle \in$ CLIQUE
If $\langle\phi\rangle \in$ 3SAT, then a satisfiable truth assignment τ exists such that every clause in ϕ returns true. This means at least one variable in each triplet evaluates to true. If one puts these three variables (henceforth vertices) into a set, these vertices will form a k-clique.
Now, for the reverse direction, suppose that G contains a k-clique:
We know the elements of this clique must all appear in distinct clauses since there are no edges to vertices of the same clause. Similarly, we know none of these vertices have edges to contradictory labels by the given construction above. Now, create a truth assignment τ with the labels from the vertices inside the k-clique. Therefore, this truth assignment must satisfy ϕ. Therefore, if $\langle G, k\rangle \in$ CLIQUE, then $\langle\phi\rangle \in$ 3SAT

11.3 What is this?

What are the gadgets?
The core idea is that a clique must contain k vertices who's variables can ALL BE TRUE at the same time, AND are of different clauses. This is a corollary of satisfiability. This is why the two restrictions on edges exist. In short, the reasoning is that "If k variables can be true at the same time (SAT), then an edge exists between all of them", and "if k vertices form a clique (CLIQUE), then they can all be true at the same time, and are of different clauses

11.4 Time Complexity Analysis

I argue that our mapping reduction is polytime because

1. It iterates over ϕ once to create G's adjacency list: $\mathcal{O}(n)$
2. It iterates over G's adjacency list once to connect all edges: $\mathcal{O}(n^2)$
3. It iterates twice more to ensure a proper construction: $\mathcal{O}(2n^2)$

Therefore, this is a polytime construction

11.5 The big result?

CLIQUE is NP-complete

12 Vertex Cover is NP Complete

VERTEX-COVER=$\{\langle G, k\rangle$ G is an undirected graph which has a k-node vertex cover$\}$
A vertex cover VC is a set of vertices such that all edges either start or end at a node in VC

12.1 Proof that VERTEX-COVER$\in$NP

Here's a polynomial time verifier for VERTEX-COVER, where C is a set of nodes which are in the vertex cover

VerifyVERTEX-COVER$\langle\langle G, k\rangle, C\rangle$:

if $|C| < k$:
return false
For each vertex in C:
mark each of its adjacent edges as covered
for each edge in G:
if the edge is not marked as covered:
return false
return true

This verifier clearly runs in polynomial time since it goes through every edge of G once, which serves a maximum complexity of $\mathcal{O}(n^2)$, where n is the number of vertices, and n^2 is the number of edges in a fully connected graph.

12.2 Proof that VERTEX-COVER is NP-Complete

While it might seem reasonable to reduce from CLIQUE $\leq_p$ VERTEX-COVER, we are instead going to choose 3SAT
WTS: 3SAT $\leq_p$ VERTEX-COVER
So, our reduction takes inputs in 3SAT, and transforms them into inputs into VERTEX-COVER, namely a graph G, and a size k. Our input is a 3-cnf formula ϕ
To make this reduction, we will
1. Take every variable in ϕ, and add a vertex in G.
2. If x_i and $\bar{x}_i$appear in G, add an edge between x_i and $\bar{x}_i$.
3. If 3 variables occur in a clause of ϕ, add a new copy of each variable into G, and add an edge between each new vertice to every other vertice (GRAPH NEEDED)
4. Connect all x_i's to other x_i's and all $\bar{x}_i$to every other $\bar{x}_i$
5. set k to be the number of variables in ϕ + 2 times the number of clauses in ϕ
Now, for the justification:
If $\langle\phi\rangle \in$ 3SAT, then, select a satisfiable assignment τ, and put all the variable widgets that are assigned true in τ into our vertex cover. Now, take 1 variable that was assigned true from each clause, and put the other two variables in its clause into the vertex cover. Since our vertex cover has 1 variable per variable widget in the VC, and 2 clause widget variables per clause in the VC, it must have k vertices in the vertex-cover. Therefore, a vertex cover of size k exists.
If $\langle G, k\rangle \in$ VERTEX-COVER, then our Vertex Cover VC, must select 1 and only 1 variable per variable widget, since if it selected 2, it would have to skimp out on including clause variable widgets, which wouldn't form a full vertex cover. In addition, the vertex cover must select 2 nodes from each clause, where the unselected node is set to true, since if it didn't it would miss the edge from a node which was set to false to the variable widget. This forms a VC of size k, and the selected variables in the vertex clause must form satisfiable truth assignment since one unselected-but-true edge must exist in each clause.

13 HAMPATH is NP-Complete

A hamiltonian path is a path that visits each vertex of a graph exactly once. A hamiltonian path from s to t consists of a hamiltonian path with its head at s and tail at t. (heads and tails since these are directed graphs)
Formally: HAMPATH = $\{\langle G, s, t\rangle \mid$ G contains a hamiltonian path from s to t$\}$

13.1 HAMPATH$\in$NP

Here is a polytime verifier for HAMPATH. Let C, the certificate, be a hamiltonian path (a set of vertexes and edges) from s to t.

VerifyHAMPATH$\langle\langle G, s, t\rangle, C\rangle$:

Trace along the path C: and mark edges G which were visited. If an edge that was previously visited is visited again, reject

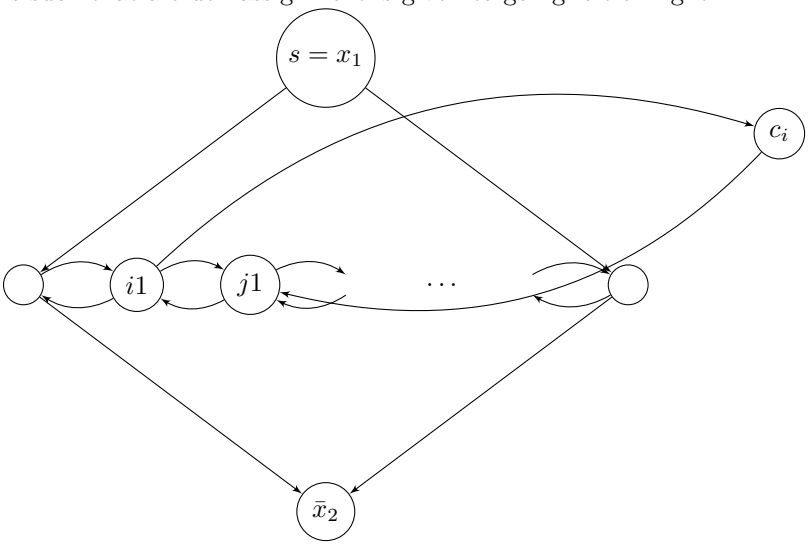
check if every vertex in G, if all were visited, accept

else reject (or loop?)

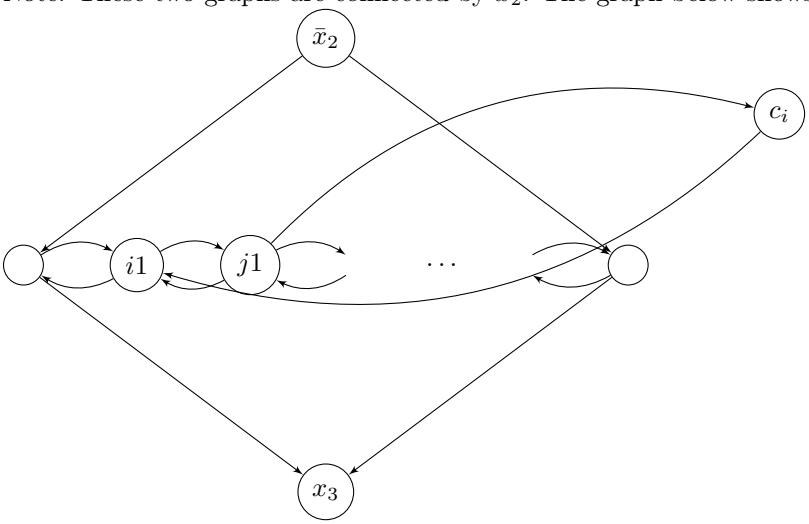
This verifier goes through the path C, which is $\mathcal{O}(n)$. Then, checking every vertex is visited takes $\mathcal{O}(n)$. Therefore, this verifier is polynomial time and takes $\mathcal{O}(n)$.

13.2 HAMPATH is NP-Complete

WTS: 3SAT $\leq_p$ HAMPATH
To do this, we need to take 3-cnf formulas ϕ and turn them into paths which contain a hamiltonian path if and only if ϕ is satisfiable. The way we do this is with two primary structures: The variable widget and the clause widget. The structure of the variable widget is such that a truth assignment is given to going left or right.



Note: These two graphs are connected by $\bar{x}_2$. The graph below shows what happens when a negated variable occurs



The number of these "vertices" in the middle is equal to 3 times the number of clauses. 3 times allows for 1 buffer node between pairs If a variable x_i is in clause c_j, then add an edge from the jth pair in the ith diamond to the jth clause node. But, if a variable $\bar{x}_i$ appears in clause c_j, then add two edges going in the reverse direction.

Finally, let G = the graph we have just constructed, and let s be the first variable widget, and t be the last variable widget. This is how we construct G,s, and t.
How this works: One can only assign 1 value (left/right or true/false) to each variable widget, and each clause widget must be gone through at least once by a single variable. This simulates a truth assignment, and that that truth assignment assigns true to each clause iff there is a satisfiable truth assignment for ϕ.

13.2.1 If $\langle\phi\rangle \in$ 3SAT

Then ϕ has a satisfiable truth assignment τ. Following the variable widget convention, we will go left/right according to what the truth assignment tells us to do. When we encounter the option to visit a clause, we will take it unless the clause has already been satisfied (iėvisited). Since this formula is satisfiable, all clause nodes will be hit, as well as the top and bottom of each variable widget, and all nodes inbetween. This implies that $\langle G, s, t\rangle \in$ HAMPATH.

13.2.2 If $\langle G, s, t\rangle \in$ HAMPATH

Then G has a hamiltonian path from s to t, where s is the first variable, and t is the last variable in a truth assignment τ. This hamilton path must cross through each clause widget once, and must cross through each variable widget once. This implies that there must be a truth assignment such that every clause is satisfied, which implies that τ is a satisfiable truth assignment for ϕ. Therefore, $\langle\phi\rangle \in$ 3SAT

14 UHAMPATH is NP-Complete

A verifier for UHAMPATH is the same as a verifier for HAMPATH

14.1 UHAMPATH is NP-Hard

UHAMPATH=$\{\langle G, s, t\rangle \mid$ G is an undirected graph which contains a hamiltonian path from s to t$\}$
We are going to reduce from HAMPATH$\leq_p$UHAMPATH
The construction goes as follows: Given a $\langle G, s, t\rangle$, an input to HAMPATH, we must construct a $\langle G', s', t'\rangle$ which is an input to UHAMPATH.
1. For each vertex (except s and t) in G, add v^i, v^m, v^o to G' where i stands for in, m stands for mid, and o stands for out.
2. set $s'$to s^o, and t' to t^i.
3. Connect v^m with v^i and v^o
4. If $(u, v) \in E(G)$, then connect u^o to v^i.

14.2 The forward direction

Assume that G contains a hamiltonian path from s to t. By the given construction above, each edge $(u, v) \in G$ has a corresponding path $u^o \to v^i \to v^m \to v^o \in G'$. This implies a UHAMPATH exists from s to t in G'.

14.3 The reverse direction

IDEK.

15 SUBSET-SUM is NP-Complete

15.1 Proof that SUBSET-SUM$\in$ NP

15.2 Proof that SUBSET-SUM is NP-Hard

WTS: 3SAT$\leq_p$SUBSET-SUM

15.2.1 The construction

Given a 3-cnf ϕ, an input to 3SAT, with l variables, and i clauses. Make a table as follows:

	x_1	x_2	$\cdots$	x_l	c_1	c_2	$\cdots$	c_i
t_1	1	0	0	0				
f_1	1	0	0	0				
t_2	0	1	0	0				
f_2	0	1	0	0				
$\vdots$	0	0	$\vdots$	0				
$\vdots$	0	0	$\vdots$	0				
t_l				1				
f_l				1				
t	1	1	$\cdots$	1				

We are going to make the rows our subset, and the row t will be our sum. If row t_1 is selected then make variable x_1 true, and if row f_1 is selected, make variable x_1 false. Since t contains a 1 in each column with a variable, this means that every variable can only be assigned true or false.
Next, modify the table. If t_j is selected, mark each clause in the row t_j 1 if x_j appears in the clause like so:

	x_1	x_2	$\cdots$	x_l	c_1	c_2	$\cdots$	c_i
t_1	1	0	0	0	1	0	$\cdots$	0
f_1	1	0	0	0	0	1	$\cdots$	1
t_2	0	1	0	0		$\vdots$		
f_2	0	1	0	0				
$\vdots$	0	0	$\vdots$	0				
$\vdots$	0	0	$\vdots$	0				
t_l				1				
f_l				1				
t	1	1	$\cdots$	1				

Now, we have to ensure that at least 1 variable in each clause is marked as true. The problem is that one clause, two clauses, and three clauses could be true. The way we control for this is adding two extra rows per clause, which both contain a 1, like so:

	x_1	x_2	$\cdots$	x_l	c_1	c_2	$\cdots$	c_i
t_1	1	0	0	0	1	0	$\cdots$	0
f_1	1	0	0	0	0	1	$\cdots$	1
t_2	0	1	0	0		$\vdots$		
f_2	0	1	0	0				
$\vdots$	0	0	$\vdots$	0				
$\vdots$	0	0	$\vdots$	0				
t_l				1				
f_l				1	1	0	$\cdots$	1
h_1	0	0	0	0	1			
g_1	0	0	0	0	1			
h_2	0	0	$\vdots$	0		1		
g_2	0	0	$\cdots$	0		1		
h_3	0	0	0	0			$\cdots$	
g_3	0	0	0	0			$\cdots$	
h_4	0	0	0	0				1
g_4	0	0	0	0				1
t	1	1	$\cdots$	1	3	3	3	3

Now, let t be the t row, and the subset S be all the other rows of the table.

15.2.2 Given that $\phi \in$ 3SAT, show $\langle S, t \rangle \in$ SUBSET-SUM

If ϕ is in 3SAT, that means there is a truth assignment τ which satisfies ϕ. This ϕ must also satisfy at least one variable in each clause. By the above construction, that implies that there must be some subset which sums to t.

15.2.3 Given that $\langle S, t\rangle \in$ SUBSET-SUM, show $\langle \phi \rangle \in$ 3SAT

We know S has some subset which sums to t. Using the variable construction, we know this subset must select either t_i or f_i for $1 \leq i \leq l$, where l is the number of rows (variables) in this section of the construction. Let our truth assignment τ assign true and false accordingly (true if t_i is selected, false if f_i is selected). In the second quadrent of the table, the clause section, if there is a subset sum which sums to t, then that subset must contain at least a 1 for each clause. By the above construction, this implies τ sets each clause to true. This implies that ϕ is satisfiable, which implies that $\langle \phi \rangle \in$ 3SAT

15.2.4 Justification that this reduction is polytime.

To constructing the structure (dimensions and layout) of the table, which is assumed to be constant time, filling out the variables section takes $\mathcal{O}(l)$ time, and filling out the clause section takes $3 \cdot \mathcal{O}(l)$ time, where l is the number of variables. If n is the number of variables in ϕ, this reduction takes $\mathcal{O}(l)$ time, which is polytime.

16 3-COL is NP-Complete

3-col:
Input: A graph G
Question: Is there a coloring of G with 3 colors such that no two adjacent vertices share the same color.

16.1 Proving that 3-COL is in NP

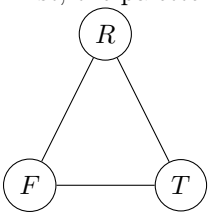
1. 3-COL asks us if an object exists or not. This means its a decision problem
2. A certificate would be a function which maps vertices to colorings.
3. For 3-COL, the input size is measured as the number of vertices in G. This is linear to a map which takes the vertices of G as input, for obvious reasons
4. The verification algorithm would iterate over each vertex, and check that none of its neighbors are colored with the same color
5. For a worst-case fully connected graph, this verifier would have to check each node and its n neighbors. This means it runs in n^2 time

16.2 Showing that 3-COL is NP-Hard

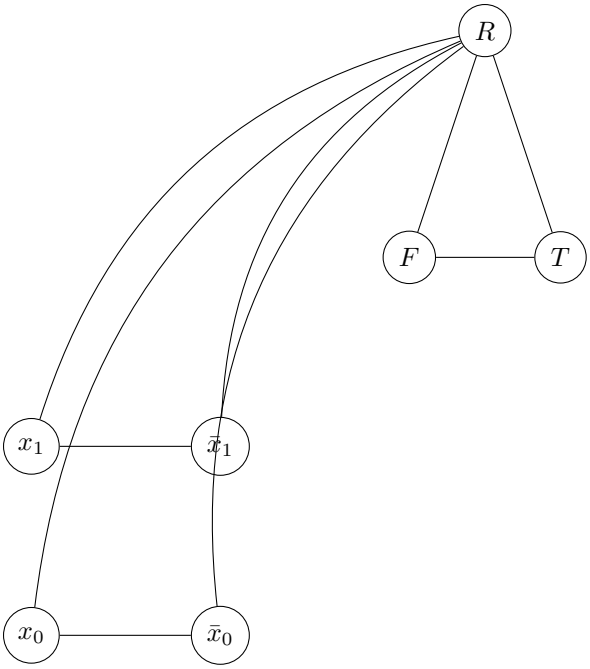
WTS: 3SAT$\leq_p$3COL

16.2.1 The construction

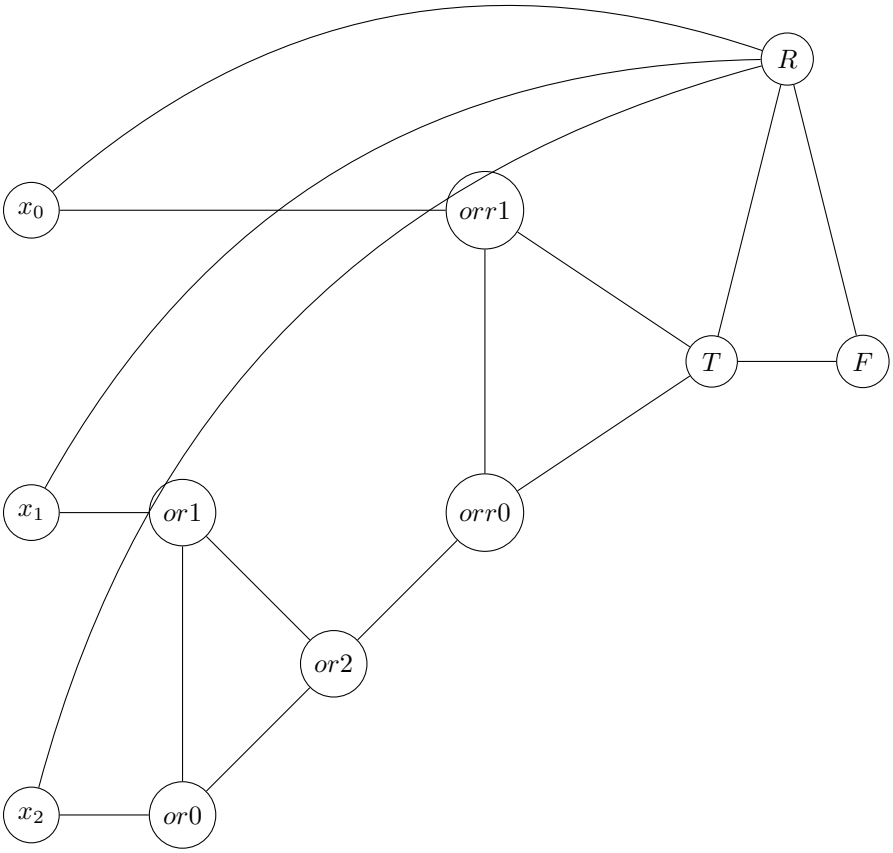
We need three core technologies to complete this reduction: The Variable Widget, the Clause Widget, and the Palette, as well as our three colors: True, False, and Red.
First, the palette looks like:



In order for this graph to be 3-colorable, this palette widget must have 3 distinct colors on each vertex. With out loss of generality, let these vertexes be named True, False, and Red, with the assumption that a 3-coloring of this graph will color these vertices appropriety.
Next is the variable widget:



This widget assures that variables can only be assigned the true color or the false color (which will correspond with their truth assignments). In addition, these widgets assure that a variable and its complement cannot be assigned to the same color.
Finally, and most complicated: The clause widget:



16.2.2 Assume $\langle\phi\rangle \in$ 3SAT, show $\langle G\rangle \in$3COL

16.2.3 Assume $\langle G\rangle \in$3COL, show $\langle\phi\rangle \in$3SAT

16.2.4 The reduction is polytime

17 COL is NP-complete

COL=$\{\langle G, k\rangle \mid$ the nodes in G can be colored with k colors st no two nodes joined by an edge have the same color$\}$

17.1 Proof that COL$\in$ NP

1. Col is a decision problem since its a language
2. A certificate to a verifier for COL would be a k-coloring of each node in G
3. The certificate would be polysize with respect to $\langle G, k\rangle$ since it would have one entry per vertex in G, which implies linearity
4. A verifier would check that the coloring was valid by iterating through every vertex and checking that its neighbors are not the same color
5. The verifier would be polytime because, worst case, for a fully connected graph, the verifier would have to check n^2 edges total. This is polytime

17.2 Proof that COL is NP-Complete

WTS: 3COL$\leq_P$ COL

17.2.1 The construction

First, the trivial cases need to be put out of their misery: The case when k=1 is trivial: Unless the graph is empty or 1-vertex, it is false. And cases where k<1 are not worth considering. If k=2, this is the same thing as saying that every node has only two or less vertices leading out of it. This is polytime and not worth further consideration.
Now, building on the ideas in a previous piazza post, we are going to do our construction by first picking a $k > 2$ out of a magic hat. Now, modulo k by the number of vertices. Then, let $j = k - 3$. Now, add j new vertices, and connect the new vertices to every node in the graph. This will make it so that j of our k colors are taken, and the potentially-3-colorable graph in the middle must be 3-colorable for our input to be in COL

17.2.2 The forward direction

Assuming that $G \in$ 3COL, let $\langle G', k\rangle = f(G)$. Since G is a subgraph of G' and G is 3-colorable, we know that G' is k-colorable since we can assign the other j colors to j added nodes.

17.2.3 The reverse direction

Assuming that $G \notin$ 3COL, let $\langle G', k\rangle = f(G)$. We know G is not 3-colorable so we must use extra colors to color in G. This implies that one of our j nodes must have the same color as a vertex in G. This implies that G' is not k-colorable

17.2.4 Justification for polytimeness

This construction is polytime because pulling a magic number out of a hat is polytime if you hook up a turing machine to some uranium-235 as a hardware RNG source. Then, adding $\mathcal{O}(k)$ edges is $n \cdot k$ time, which is n^2 time since k is bound by n. Therefore, our construction is polytime

Part III
Space complexity

I didn't end up writing this