CSC 363 Tutorial 1 : Write-Once Turing Machines

Tutor: Noah Fleming
Scribes: Noah Fleming and Robert Robere

January 11, 2016

In this tutorial we reviewed what it means for a Turing Machine to "compute", and then we examined *write-once Turing Machines*, which are a unusual modification of Turing Machines which are only allowed to alter each tape square at most once, yet (surprisingly!) are able to simulate regular Turing Machines. The definition of a write-once Turing Machine is identical to that of a regular Turing Machine; the only difference is in *how* they compute languages.

Definition 1. A Turing Machine $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$ is said to be *write-once* if on any input $x \in \Sigma^*$ each tape square (including the input squares) are altered by the tape head at most once.

Let us give a careful definition of a computation of a write-once Turing Machine M. First, recall the definition of a *configuration* and the *yields* relation. Suppose that $y, z \in \Gamma^*$, $\alpha, \beta, \gamma \in \Gamma$, and $q, q' \in Q$. Let C be the configuration $y\alpha q\beta z$ (note that $y\alpha\beta z$ is the string encoded on the tape at this time). Let $\delta(q, \beta) = (q', \gamma, D)$. If $D = L$, then C *yields* the configuration $yq'\alpha\gamma z$. Intuitively, the machine has replaced the symbol β under the read-write head with γ, changed its internal state to q', and moved to the left a single step. If $D = R$, then C *yields* the configuration $y\alpha\gamma q'z$. Now, the machine has replaced the symbol and changed its state like before, but it has moved the read-write head to the right.

If the read-write head is at one of the ends of the configuration we have some special cases. Suppose the configuration is of the form $q\beta z$. The read-write head is at the left end of the tape, and so if $D = L$ then $q\beta z$ yields the configuration $q'\gamma z$, since it cannot move off the left side of the tape. If $D = R$ then $q\beta z$ yields $\gamma q'z$.

Finally, suppose that the configuration is of the form $y\alpha q$. This configuration is equivalent to $y\alpha q\sqcup$ since the read-write head is pointing at the blank symbol at the right end of the string on the tape. If $D = L$, then $y\alpha q = y\alpha q\sqcup$ yields $yq'\alpha\gamma$; if $D = R$ then $y\alpha q$ yields $y\alpha\gamma q$.

A sequence of configurations $C_0, C_1, \ldots, C_m$ is a *halting computation* of M on input x if the

1. C_0 is the initial configuration on x,
2. C_i yields C_{i+1} for all $i = 0, 1, \ldots, m-1$,
3. C_m contains the accept state or the reject state,

4. For each pair of adjacent configurations C_i, C_{i+1} in the sequence where γ is the symbol under the read-write head in C_i and γ' is the symbol on the tape in the same square as γ in C_{i+1}, if $\gamma \neq \gamma'$ then C_i is the only configuration in the sequence where this tape square was altered.

The machine M *accepts* the input x if C_m is in an accept state, and otherwise it *rejects* x.

Exercise: How does this definition compare to the definition of a computation of a regular Turing Machine? How would you define a *write-twice* Turing Machine?

Now we outline a proof that a write-once Turing Machine can simulate a regular Turing Machine. To make things easier, first we show that a *write-twice* Turing Machine can simulate a regular Turing Machine. We start with a Turing Machine M and, using M, construct a write-twice Turing Machine M' such that for every input $x \in \Sigma^*$, $x \in \mathcal{L}(M)$ if and only if $x \in \mathcal{L}(M')$. In this case it is quite easy to prove that the two models are equivalent: M' will simulate a single step of M by copying the entire working memory of the machine to a new section of the tape, altering the new copy of the working memory in the same way that M would have altered the memory. To implement this formally, we define the tape alphabet

$$\Gamma = \Sigma \cup \{\sqcup, \#\} \cup \{\sigma' \mid \sigma \in \Sigma\} \cup \{\dot{\sigma} \mid \sigma \in \Sigma\},$$

where $\#$ will be used to mark the beginning and the end of the "current working memory" of the machine and $\sigma', \dot{\sigma}$ for each $\sigma \in \Sigma$ are "marked" versions of the characters in Σ used when we are copying the working memory.

We proceed to define the write-twice machine M' more formally. On input x, the machine M' begins by moving to the end of the input x and appending a $\#$ symbol, and then moving back to the beginning of the tape. To simulate a single step of M the machine M' first replaces the character γ under the read-write head with $\dot{\gamma}$. It then moves to the previous $\#$ symbol (or, if this is the first step of M, the beginning of the tape) and begins to copy all characters occurring between this $\#$ and the next $\#$ symbol. To do this, it reads a character α, replaces it with α' to denote that it has been copied, and then checks both of the neighbouring tape squares to see if either of them contain $\dot{\gamma}$. If not, then the machine simply scans the head to the right until it reaches the blank symbol at the end of the tape, where it writes α. If so, and if the machine M would move to the tape square containing α after modifying $\dot{\gamma}$, then instead when it moves to the end of the tape it writes $\dot{\alpha}$. This is to denote where the tape head should stay after the copying has finished

If the currently copied character α is $\dot{\gamma}$ (that is, the character under the read-write head before we began copying), then it replaces $\dot{\gamma}$ with γ', proceeds to the first blank space, and instead of writing γ it instead writes β, where β is the new character that the simulated machine M would have replaced γ with in its normal operation.

Finally, during the copying procedure, if the machine M' reaches a $\#$ symbol, it proceeds to the blank symbol at the end of the tape, writes a $\#$, and then returns to the symbol $\dot{beta}$ and moves to the left or right depending on whether the original machine M moves left or right. If the machine M is in an accepting or rejecting state after the execution of this step, then M' similarly halts and accepts or rejects, respectively.

It should be clear that each tape square is altered at most twice in the operation of M': each input symbol is changed at most twice (once during the copy, and the symbol under the read-write head before the copy is changed twice), and each blank symbol is changed at most twice (once with the initial write, and potentially once during the copy if the blank symbol was not overwritten by a $\#$). It should also be clear that $x \in \mathcal{L}(M')$ if and only if $x \in \mathcal{L}(M)$. This simulation proves that any Turing Machine M can be simulated by a write-twice Turing Machine M'. We state this as a theorem.

Theorem 2. *For any Turing Machine M there is a Turing Machine M' such that M' alters the value of each tape square at most twice, and on every input $x \in \Sigma^*$ M' halts whenever M halts and $x \in \mathcal{L}(M')$ if and only if $x \in \mathcal{L}(M)$.*

How do we improve this simulation to get a write-once Turing Machine? We give a rough sketch how to do it. First, notice that the only real "trick" that we used to transform the regular TM M into a write-twice TM M' was "marking" tape symbols. If we could somehow simulate this "marking" of tape symbols in a way that only altered each tape square once, then we would be done by essentially the same construction that was presented above.

To simulate the marking operation, we imagine replacing each tape square of the original Turing Machine with two tape squares: the first tape square represents the symbol, $\gamma \in \Sigma \cup \{\#, \sqcup\}$, and the second tape square contains a symbol in $\{', \dot{}, \sqcup\}$, which represents whether or not the symbol in the first tape square has been marked. Thus, each time we write an unmarked symbol γ to a blank square in the write-twice Turing Machine M', in the write-once machine we will move one square further to the right and write γ to leave a space for the mark of the previous symbol. Similarly, if we mark a symbol γ in the write-twice Turing Machine, say, by replacing it with γ', in the write-once machine we will move to the tape square to the right of γ and write the symbol $'$ to denote that the γ left has been marked. We leave a detailed implementation of this operation to the reader.

CSC 363 Tutorial 2
Non-deterministic Turing Machines

Tutor: Assimakis Kattis
Scribe: Robert Robere

January 18, 2016

The next important Turing Machine variant that we consider are *nondeterministic Turing Machines* (NTMs), which are analogous to the non-deterministic finite automata that you may be familiar with from earlier courses. Informally, these are Turing Machines that come equipped with a special power of "guessing" — at some points in the computation, the machine may choose any one out of a number of possible transitions. Why should we consider such a special power? Clearly such a model leaves the realm of what we would consider to be a "reasonable" computational model: the computers that sit on our desks can certainly not magically guess a correct sequence of state transitions to make to the correct answer. It turns out that while non-determinism is not realistic, it is still *equivalent* in power to a regular Turing Machine, and it turns out to be hugely important in the study of complexity theory.

This power of guessing does make defining what it means for an NTM to "accept" its input somewhat problematic: we say that an NTM M *accepts* its input if *there exists* a sequence of guesses which cause M to end in an accept state.

Definition 1. A *non-deterministic Turing Machine* is a 7-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$ where:

1. Q is a finite set of *states*.
2. Σ is a finite set of symbols called the *input alphabet*.
3. $\Gamma \supseteq \Sigma$ is a finite set of symbols called the *tape alphabet*, which contains a special "blank" symbol $\sqcup \notin \Sigma$.
4. $\delta \subseteq Q \times \Gamma \times Q \times \Gamma \times \{L, R\}$ is a *transition relation*, which relates state-symbol pairs (q, γ) with triples (q', γ', D) where $q' \in Q, \gamma' \in \Gamma$, and $D \in \{L, R\}$.
5. $q_0 \in Q$ is the *start state*.
6. $q_{acc} \in Q$ is the *accept state*.
7. $q_{rej} \in Q$, distinct from q_{acc}, is the *reject state*.

Formally, the definition of a non-deterministic Turing Machine varies from a regular Turing Machine in that its transition function is replaced with a *transition relation*, and if at some time step the non-determinstc TM is reading a symbol γ and in a state q, it may choose to transition to *any* of the possible triples (γ', q', D) such that $((\gamma, q), (\gamma', q', D)) \in \delta$. The definition of an accepting computation is similar to that of a deterministic Turing Machine except that we say a NTM M accepts an input x if there is any sequence of legal transtions using the transition relation δ which leads to the acceptance of x.

Despite the power of guessing (and accepting if any sequence of possible guesses is correct), it turns out that non-deterministic TMs are equivalent in power to deterministic TMs. They also give clean alternative definitions of the classes SD and D.

Theorem 2. *For any language L, $L \in$ SD if and only if there is an NTM that computes L.*

Proof. If L is in SD, then there is a deterministic Turing Machine M such that $\mathcal{L}(M) = L$, and since any deterministic Turing Machine is a non-deterministic Turing Machine (by interpreting the transition function as a relation) we get that M is an NTM that computes L. So, the hard direction of the proof is in converting an NTM M computing L to a deterministic TM M' computing L.

As illustrated in tutorial, we can think of the computation of a non-deterministic Turing Machine as a "computation tree", representing at each configuration C reachable by the NTM the possible configurations C' reachable from C in one step via the transition relation δ. At a high level, our proof proceeds by constructing a deterministic Turing Machine M' which explores this computation tree via a breadth-first search that halts and accepts if and only if there is an accepting configuration C in the computation tree of the NTM M. Note that the computation tree may be infinite (if a sequence of non-deterministic guesses leads to a non-halting computation), which is why we explore the tree in a *breadth* first manner: a depth-first search could accidentally search down one of these infinite branches forever. By the correctness of the breadth-first search algorithm, for every node n in the computation tree of M the algorithm will eventually explore n; thus, if there is an accepting sequence of transitions in the NTM on input x the breadth-first algorithm will find it and accept. □

We say that a non-deterministic Turing Machine is a *decider* if every branch in the computation tree is finite — that is, every valid sequence of transitions in the NTM M eventually halts. By modifying the above proof, it is not hard to convince yourself of the following fact:

Theorem 3. *For any langauge L, $L \in$ D if and only if there is a non-deterministic decider M which computes L.*

Proof. The proof is analogous to the previous proof: now, we are guaranteed that the breadth-first search must eventually halt. If, upon halting, all leaves of the computation tree are labelled with reject, then the breadth-first search algorithm rejects; otherwise, it accepts. □

CSC363 Tutorial 3
Decidability

Tutor: Robert Robere
Scribe: Robert Robere

January 20, 2016

In this tutorial we gave a detailed proof that the following language is both semi-decidable and decidable, like A_{TM}:

$$L = \{(\langle \mathcal{M} \rangle, x) \mid \exists n \in \mathbb{N} : \mathcal{M} = M_1, M_2, \ldots, M_n \text{ is sequence of TMs and } x \in \mathcal{L}(M_i) \text{ for some } i\}.$$

For completeness, here is the definition of A_{TM} again

$$A_{\text{TM}} = \{(\langle M \rangle, x) \mid M \text{ is a TM and } x \in \mathcal{L}(M)\}.$$

It is not hard to see that L is like a "harder" or "more general" version of A_{TM}, in the sense that L is like A_{TM} except we have a sequence of Turing Machines instead of a single TM. We will show that $L \notin \mathsf{D}$ — which is not surprising, considering that it is very closely related to A_{TM} — and that $L \in \mathsf{SD}$, which *should* be surprising since the sequence of Turing Machines can be arbitrarily long. First we show that $L \notin \mathsf{D}$.

Theorem 1. *L is not decidable.*

Proof. We follow the proof that A_{TM} is not decidable, using A_{TM} in place of DIAG. Assume that L is decidable and let M_0 be the Turing Machine which decides L. Using M_0, we give a Turing Machine that decides A_{TM}, contradicting the fact that A_{TM} is not decidable.

We use the Church-Turing thesis. The algorithm for A_{TM} is quite simple: On input $(\langle M \rangle, x)$, reject if $\langle M \rangle$ is not a good encoding of a Turing Machine. If it is, then define the sequence of encodings of Turing Machines $\mathcal{M} = (\langle M \rangle)$ (note that the sequence only has one member!). Simulate M_0 on the input $(\langle \mathcal{M} \rangle, x)$, and accept (or reject) if it accepts (rejects, respectively).

For any input $(\langle M \rangle, x)$, if the algorithm for A_{TM} accepts the input, then M_0 accepts the input $(\langle \mathcal{M} \rangle, x)$. Since M_0 decides L, it follows by the definition of L that at least one Turing Machine in the sequence $\mathcal{M} = (\langle M \rangle)$ accepts x. Thus M must accept x, and so $(\langle M \rangle, x) \in A_{\text{TM}}$.

On the other hand, if the algorithm for A_{TM} rejects the input, then M_0 must have rejected the input $(\langle \mathcal{M} \rangle, x)$, and so similarly we conclude that M does not accept x. It follows that $(\langle M \rangle, x) \notin A_{\text{TM}}$. It is easy to see that the algorithm always halts, and so this algorithm decides A_{TM}. Contradiction! □

Notice that we used A_{TM} in the proof instead of DIAG: now that we know that A_{TM} is not decidable, we may use it in further "simulation" (or *reduction*) arguments to prove other languages are undecidable.

We now prove that A_{TM} is semi-decidable.

Theorem 2. *L is semi-decidable.*

Proof. To show L is semi-decidable we just need to give a Turing Machine M such that $\mathcal{L}(M) = L$. Algorithm 1, which uses dovetailing, accepts L:

Algorithm 1: Dovetailing

Input: A sequence of encodings of Turing Machines $\mathcal{M} = M_1, M_2, \ldots, M_n$, and a string $x \in \Sigma^*$

for $i = 1, 2, 3, \ldots$ **do**

  For each $j = 1, 2, \ldots, n$, simulate the machine M_j encoded by $\langle M_j \rangle$ on x for i steps;

  Accept x if any of the above simulations accept;

  Reject x if all of the above simulations reject;

end

The input $(\mathcal{M}, x)$ is accepted by the above algorithm if and only if there are $i, j \in \mathbb{N}$ such that the machine M_j accepted x after i computation steps, and so the algorithm accepts all inputs in L. If an input $(\mathcal{M}, x)$ is not in L, then either all machines encoded in the sequence $\mathcal{M}$ reject x (in which case the above algorithm rejects), or none of the machines halt on x (in which case the algorithm does not halt). It follows that the language accepted by the above algorithm is L, and so L is semi-decidable. □

CSC363 Tutorial 5
Reductions

Tutor: Assimakis Kattis
Scribe: Robert Robere

February 4, 2016

In this tutorial we classified the following two languages with respect to the computability classes $\mathsf{D}, \mathsf{SD}, \mathsf{coSD}$:

$$\text{EMPTY} = \{\langle M\rangle \mid M \text{ is a TM that does not accept any input}\}$$
$$\text{INF} = \{\langle M\rangle \mid M \text{ is a TM that accepts infinitely many strings}\}.$$

The classifications of both languages are recorded in the following two propositions.

Proposition 1. *The language* EMPTY *is not in* D *or* SD*, but it is in* coSD*. Equivalently*

$$\text{EMPTY} \in \mathsf{coSD} \setminus \mathsf{D}.$$

Proof. Showing that $\text{EMPTY} \in \mathsf{coSD}$ is easy, so we do that first. The complement of EMPTY is the language

$$\overline{\text{EMPTY}} = \{\langle M\rangle \mid \langle M\rangle \text{ does not encode a TM or } \exists y : y \in \mathcal{L}(M)\}.$$

Here is an algorithm that recognizes EMPTY.

Algorithm for $\overline{\text{EMPTY}}$

1. On input $\langle M\rangle$.
2. If $\langle M\rangle$ does not encode a Turing Machine, reject.
3. Dovetail the simulation of M over all $y \in \Sigma^*$. If any simulation accepts, then accept.

Clearly if M accepts any $y \in \Sigma^*$ then the above algorithm accepts $\langle M\rangle$, and otherwise it rejects or does not halt. Thus $\overline{\text{EMPTY}} \in \mathsf{SD}$, or equivalently $\text{EMPTY} \in \mathsf{coSD}$.

Now we prove that EMPTY is not in D by a reduction from

$$\text{DIAG} = \{\langle M\rangle \mid M \text{ is a TM and } \langle M\rangle \notin \mathcal{L}(M)\}.$$

For any Turing Machine M, consider the TM M^*_M defined by the following algorithm.

Algorithm for M^*_M

1. On input y.

2. Clear the tape, write $\langle M \rangle$.

3. Simulate M on $\langle M \rangle$, and accept if and only if the simulation accepts.

For any TM M, our reduction from DIAG to EMPTY maps $\langle M \rangle \mapsto \langle M^*_M \rangle$. It is clearly computable since given the encoding of the machine $\langle M \rangle$ we can easily compute the encoding of M^*_M. To finish showing the reduction is well-defined, we must show that

$$\langle M \rangle \in \text{DIAG} \Leftrightarrow \langle M^*_M \rangle \in \text{EMPTY}.$$

First, suppose that $\langle M \rangle \in \text{DIAG}$. Then $\langle M \rangle \notin \mathcal{L}(M)$, and so the algorithm M^*_M rejects all input strings $y \in \Sigma^*$. Thus $\langle M^*_M \rangle \in \text{EMPTY}$. On the other hand, if $\langle M^*_M \rangle \in \text{EMPTY}$, then by the definition of M^*_M we must have that $\langle M \rangle \notin M$. Thus $\langle M \rangle \in \text{DIAG}$, and the proof is complete. Since $\text{DIAG} \notin \mathsf{D}$, it follows that $\text{EMPTY} \notin \mathsf{D}$; furthermore since $\text{EMPTY} \in \mathsf{coSD}$ and $\mathsf{D} = \mathsf{SD} \cap \mathsf{coSD}$ we have that $\text{EMPTY} \notin \mathsf{SD}$. □

Proposition 2. *The language* INF *is not in* SD *or* coSD.

Proof 1: Reduction from A_{TM}. We show that INF is not in coSD by a reduction from

$$A_{\text{TM}} = \{(\langle M \rangle, x) \mid M \text{ is a TM and } x \in \mathcal{L}(M)\}.$$

For any pair (M, x), consider the TM $\hat{M}_{(M,x)}$ defined by the following algorithm.

Algorithm for $\hat{M}_{(M,x)}$

1. On input y.

2. Simulate M on x, and accept if and only if the simulation accepts.

The reduction maps $(\langle M \rangle, x) \mapsto \langle \hat{M}_{(M,x)} \rangle$, and once again it is clearly computable. We now prove that

$$(\langle M \rangle, x) \in A_{\text{TM}} \Leftrightarrow \langle \hat{M}_{(M,x)} \rangle \in \text{INF}.$$

If $(\langle M \rangle, x) \in A_{\text{TM}}$ then, by definition, M must accept x. It follows that $\hat{M}_{(M,x)}$ must accept all inputs, and so $\mathcal{L}(\hat{M}_{(M,x)}) = \Sigma^*$. In particular, it is infinite, and so $\langle \hat{M}_{(M,x)} \rangle \in \text{INF}$.

On the other hand, if $\langle \hat{M}_{(M,x)} \rangle \in \text{INF}$, then by the definition of the algorithm we must have that M accepts x. It follows that $(\langle M \rangle, x) \in A_{\text{TM}}$. The reduction is well-defined, and so since $\mathsf{A}_{\text{TM}} \notin \mathsf{coSD}$ we have that $\text{INF} \notin \mathsf{coSD}$.

Now we show that INF is not in SD by a reduction from

$$A^*_{\text{TM}} = \{(\langle M \rangle, x) \mid M \text{ is a TM and } x \notin \mathcal{L}(M)\}.$$

Note that A^*_{TM} is "almost" the complement of A_{TM}, which is actually

$$\overline{A}_{\text{TM}} = \{(\langle M \rangle, x) \mid \langle M \rangle \text{ does not encode a TM or } x \notin \mathcal{L}(M)\}.$$

For the sake of variety, we first show that A^*_{TM} is not in SD (and along the way we implicitly show how to get rid of those pesky "or" conditions in the definition of the complement.)

We know that $\overline{A}_{\text{TM}} \notin \mathsf{SD}$ since $A_{\text{TM}} \notin \mathsf{coSD}$, and observe that we can write

$$\overline{A}_{\text{TM}} = A^*_{\text{TM}} \cup \{\langle M \rangle \mid \langle M \rangle \text{ does not encode a TM}\}.$$

The second set in the above partition is clearly semi-decidable, so if A^*_{TM} is SD then $\overline{A}_{\text{TM}}$ is SD, which would be a contradiction. Thus A^*_{TM} is not in SD.

Now, for the reduction. Let $(\langle M \rangle, x)$ be any TM/input pair, and consider the TM $M'_{(M,x)}$ defined by the following algorithm.

Algorithm for $M'_{(M,x)}$

1. On input y.
2. Simulate M on x for $|y|$ steps. If M has accepted x when the simulation halts, reject. Otherwise, accept.

As usual, our reduction maps $(\langle M \rangle, x) \mapsto \langle M'_{(M,x)} \rangle$. It is easily seen to be computable, so we focus on the other condition.

Let $(\langle M \rangle, x) \in A^*_{\text{TM}}$. Then M does not accept x, and so the machine $M'_{(M,x)}$ never rejects any input y. Thus $\mathcal{L}(M'_{(M,x)})$ is infinite and so $\langle M'_{(M,x)} \rangle \in \text{INF}$. On the other hand, suppose that $\langle M'_{(M,x)} \rangle \in \text{INF}$. If the machine M accepts the input x, then it must have done so after a finite number of computation steps. In such a case, the machine $M'_{(M,x)}$ must accept only finitely many inputs, and so we would conclude that $\langle M_{(M,x)} \rangle \notin \text{INF}$, a contradiction. Thus M must not accept x, and so $(\langle M \rangle, x) \in A^*_{\text{TM}}$. Since $A^*_{\text{TM}} \notin \mathsf{SD}$, it follows that $\text{INF} \notin \mathsf{SD}$. $\square$

Proof 2: Reduction from EMPTY. Now we show that INF is not in SD or coSD by reducing from EMPTY. First let us reduce EMPTY to INF, and then we reduce EMPTY to $\overline{\text{INF}}$.

Let $\langle M \rangle$ be any TM, and consider the TM $\hat{M}_M$ defined by the following algorithm.

Algorithm for $\hat{M}_M$

1. On input $y \in \Sigma^*$.
2. Dovetail M over all $z \in \Sigma^*$ for $|y|$ steps. If any simulation halts and accepts, reject. Otherwise accept.

Our reduction maps $\langle M \rangle \mapsto \langle \hat{M}_M \rangle$, and now we prove that $\langle M \rangle \in \text{EMPTY}$ if and only if $\langle \hat{M}_M \rangle \in \text{INF}$. First, suppose that $\langle M \rangle \in \text{EMPTY}$. Then for any $y \in \Sigma^*$ the machine $\hat{M}_M$ will accept y since M does not accept any inputs z. It follows that $\mathcal{L}(\hat{M}_M) = \Sigma^*$ and so $\langle \hat{M}_M \rangle \in \text{INF}$.

On the other hand, if $\langle \hat{M}_M \rangle \in \text{INF}$, then by the definition of $\hat{M}_M$ we must have that $\mathcal{L}(M) = \emptyset$. Why? By contradiction, if $\mathcal{L}(M) \neq \emptyset$ then for some $i \in \mathbb{N}$ we will have that $\hat{M}_M$ rejects all $y \in \Sigma^*$ with $|y| \geq i$ (this will happen as soon as the strings y are "long enough" to allow the simulation of M to accept some $z \in \Sigma^*$). Thus $\mathcal{L}(\hat{M}_M)$ is finite, and so we have a

contradiction. Thus $\text{EMPTY} \leq_m \text{INF}$ and since $\text{EMPTY} \notin \mathsf{SD}$ we have that INF is not in SD.

Now let us reduce EMPTY to

$$\overline{\text{INF}} = \{\langle M \rangle \mid \langle M \rangle \text{ does not encode a TM or } \mathcal{L}(M) \text{ is finite}\}.$$

Once again, let M be any TM and consider the TM M'_M defined by the following algorithm.

Algorithm for M'_M

1. On input $y \in \Sigma^*$.
2. Dovetail M over all $z \in \Sigma^*$ for $|y|$ steps. If any simulation halts and accept, accept. Otherwise reject.

The proof of this reduction is very similar to the previous case. Suppose that $\langle M \rangle \in \text{EMPTY}$ and we show that $\langle M'_M \rangle \in \overline{\text{INF}}$. Since $\langle M \rangle \in \text{EMPTY}$ none of the simulations of M in the algorithm $\langle M'_M \rangle$ will accept, and so it follows that M'_M rejects all $y \in \Sigma^*$. Thus $\mathcal{L}(M'_M) = \emptyset$ — in particular it is finite, and so $\langle M \rangle \in \overline{\text{INF}}$.

On the other hand, suppose that $\langle M'_M \rangle \in \overline{\text{INF}}$. It is a well-defined encoding of a TM, and so we must have $\mathcal{L}(M'_M)$ is finite. Suppose by way of contradiction that $\langle M \rangle \notin \text{EMPTY}$, and so $\mathcal{L}(M) \neq \emptyset$. Let $z^* \in \mathcal{L}(M)$, and suppose that M accepts z^* after i computation steps. Then for all $y \in \Sigma^*$ with $|y| \geq i$ we have that M'_M accepts y, and so $\mathcal{L}(M'_M)$ is infinite (a contradiction). Thus $\langle M \rangle \in \text{EMPTY}$.

Since $\text{EMPTY} \notin \mathsf{SD}$, it follows from the above reduction that $\overline{\text{INF}} \notin \mathsf{SD}$, and so $\text{INF} \notin \mathsf{coSD}$. □

Tutorial 6: Dominating Set and Subgraph Isomorphism

Tutor: Noah Fleming
Scribe: Robert Robere

March 8, 2016

In this tutorial we introduced two new problems in NP. The first is of a different flavor than the other NP problems we have seen so far. If G and H are graphs, we say that H is a *subgraph* of G if H can be obtained from G by deleting vertices and edges.

Problem 1. Subgraph Isomorphism
Input: Two graphs G and H.
Problem: Decide if H is a subgraph of G.

The second problem is similar to Vertex Cover. Recall that a *vertex cover* of a graph $G = (V, E)$ is a subset of vertices $V' \subseteq V$ such that each edge $uv \in E$ has either $u \in V'$ or $v \in V'$. A *dominating set* is similar: a subset of vertices $V' \subseteq V$ is a *dominating set* of G if every vertex $v \in V$ is either in V' or has a neighbour in V'.

Problem 2. Dominating Set
Input: A graph G and an integer k.
Problem: Decide if G has a dominating set with at most k vertices.

First we show that both problems are actually in NP.

Subgraph Isomorphism in NP. To show Subgraph Isomorphism is in NP we must give a polynomial-time verifier for it. This is fairly easy: the corresponding certificate y is just an encoding of a list of vertices in G.

Verifier for Subgraph Isomorphism

1. On input G, H and certificate y.
2. Check that y encodes a list H' of $|H|$ vertices in G. If not, reject.
3. For each pair of vertices $u, v \in H'$, check that the edge $uv \in G$ if and only if $uv \in H$. If this is true for all pairs, accept. Otherwise, reject.

Clearly if H is a subgraph of G, then there is a certificate y encoding a subgraph H' of G isomorphic to H (just choose the subset of vertices corresponding to H in G). On the other

hand, if the algorithm accepts, then G clearly contains H as a subgraph. Finally, each step of the algorithm executes in time at most polynomial in $|G|, |H|$, so the above algorithm is a verifier for Subgraph Isomorphism.

Dominating Set in NP. This is very similar to the verifier for vertex cover.

Verifier for Dominating Set

1. On input G, k, and certificate y.
2. Check that y encodes a subset V' of vertices of G. If not, reject.
3. For each vertex $u \in G$ check if u is in V' or if there is a neighbour of u in V'. If not, reject.
4. If all of the previous checks accept, then accept.

Now we need to show that for all G, k, G contains a dominating set of size at least k if and only if there is a certificate y that makes the above verifier accept, and that the above verifier runs in polynomial time. Well, if G contains a dominating set V' with at most k vertices, choose y to encode the list of vertices in V', and it is clear that the above verifier will accept G, k, y. On the other hand, if the above verifier accepts G, k and y, then by the definition of the verifier it must be that y encodes a dominating set for G. Finally, observe that each step of the above algorithm runs in time polynomial in $|G|$ and k. Thus the above algorithm is a verifier computes Dominating set.

Now we give two reductions: we show that solving Subgraph Isomorphism allows us to solve the Clique problem, and we also show that solving the Dominating Set problem allows us to solve Vertex Cover.

Clique to Subgraph Isomorphism. Let G be a graph and let k be a positive integer with $k \leq |G|$, and we construct a pair of graphs G', H' such that G has a clique of size at least k if and only if G' has H' as a subgraph.

The reduction is quite easy: let $G' = G$, and let $H' = K_k$, where K_k is the complete graph on k vertices. If G contains a clique C with at least k vertices, then in particular C appears as a subgraph of G. But K_k is a subgraph of C, and so $G' = G$ contains K_k as a subgraph. Conversely, if G' contains K_k as a subgraph then $G = G'$ contains a k-clique by definition.

Vertex Cover to Dominating Set. Now we do a reduction that is a little more involved. Let G be a graph, and let k be a positive integer. We construct a graph G' and an integer k' such that G has a vertex cover with at most k vertices if and only if G' has a dominating set with at most k' vertices.

First we make a useful observation — a vertex cover is (essentially) a dominating set. Let $G = (V, E)$ be a graph and suppose $V_0 \subseteq V$ is a vertex cover. Then every edge $e \in E$ has one of its endpoints in V_0. It follows that every vertex $v \in V$ that has at least one edge connected to it is covered by V_0. Thus, to create a dominating set from V' we just need to add every isolated vertex to V_0.

This suggests an obvious reduction: just set $G' = G$ and $k' = k$, and one half of what we need to prove is done (that is, if G has a vertex cover of size at most k then G also has a dominating set of size at most k). What's the problem? Well, the converse is *not* true! Consider the path graph P_4 with four vertices. The two endpoints of the path are a dominating set, but the edge in the center is not covered (see Figure 1).

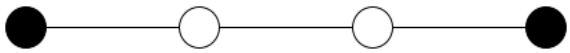


Figure 1: The black vertices form a dominating set in P_4 but are not a vertex cover.

So, we need to somehow modify the graph G' so that we can convert dominating sets of G' back into vertex covers of G. We do this by introducing a new "special vertex" u_e for each edge e in G which is connected to both endpoints of e by two new edges. Figure 2 depicts the reduction when applied to the graph P_4.

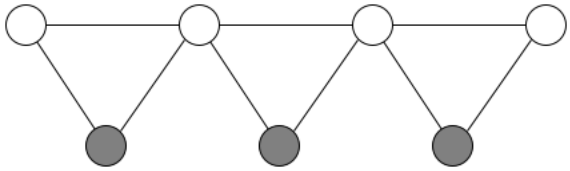


Figure 2: We introduce one new vertex (coloured grey) for each edge.

Formally, let $G' = (V', E')$ be defined as follows. The vertices V' are obtained by adding all non-isolated vertices of G, plus a new vertex u'_e for each edge $e \in E$:

$$V' = V \cup \{v_e \mid e \in E\}.$$

The edges E' contain all edges in G, plus two new edges (u'_{uv}, u) and (u'_{uv}, v) for each new vertex u'_{uv} connecting it to the endpoints of the edge uv:

$$E' = E \cup \{(u'_{uv}, u), (u'_{uv}, v) \mid uv \in E\}.$$

Finally, let $k' = k$.

Now we argue that G has a vertex cover of size at most k if and only if G' has a vertex cover of size at size at most $k' = k$. Let $V_0 \subseteq V$ be a vertex cover of size at most k in G. We claim that V_0 is a dominating set of G'. To see this, let $u \in V'$ be a vertex, and we need to show that either $u \in V_0$ or every neighbour of u is in V_0.

First observe that u is not isolated, since we deleted all isolated vertices of G from G'. Assume that $u \notin V_0$, as otherwise it is trivial, and let v be a neighbour of u. If v is a vertex in the original graph, then the edge uv is covered by V_0 and so v must be in V_0 since u is not. On the other hand, if v is a new vertex, then there must be a third vertex w connected to u such that v is the new vertex corresponding to the edge uw (i.e. $v = u'_{uw}$). Since V_0 is a vertex cover and $u \notin V_0$ we must have that $w \in V_0$, as otherwise the edge uw would not be covered by V_0. But w is a neighbour of v, and so we are done. Thus V_0 is a dominating set in G'.

Now, suppose that $V_1' \subseteq V$ is a dominating set in G' with at most k vertices, and we construct a vertex cover $V_1 \subseteq V$ in G. The cover is constructed as follows: if $V_1' \subseteq V$ (i.e. it contains only old vertices) then set $V_1 = V_1'$. Otherwise, for each new vertex $u'_{uv} \in V_1'$, replace u'_{uv} with one of u or v arbitrarily, and let V_1 be the resulting set of vertices. Clearly $|V_1| = |V_1'| \leq k$.

Finally, we argue that V_1 is a vertex cover. Let $uv \in E$ be any edge in G, and consider the corresponding new vertex u'_{uv} in G'. In order to cover the new vertex u'_{uv} in G', we must include one of u'_{uv}, u, or v inside the dominating set V_1'. If either u or v is included in V_1', then uv is covered by V_1. Otherwise, if $u'_{uv} \in V_1'$, then by replacing it with u or v we cover the edge uv in G, and the proof is complete. In either case the edge uv is covered, and the proof is complete.

Tutorial 7: SAT and k-SAT

Tutor: Assimakis Kattis
Scribe: Robert Robere

March 9, 2016

In this tutorial we introduce an NP-Complete problem which is very different from the graph-type problems that we have seen so far.

This problem is based in logic, and in the rest of the notes we identify 1 with True and 0 with False. Let x, y be boolean variables, and recall the semantics of the basic operations $\wedge$ (AND), $\vee$ (OR), and $\bar{\cdot}$ (NOT):

- $x \wedge y = 1$ if and only if $x = 1$ and $y = 1$,
- $x \vee y = 1$ if and only if $x = 1$ or $y = 1$,
- $\overline{x} = 1$ if and only if $x = 0$.

A *boolean formula* ϕ on n variables is a sentence composed of the basic boolean operations and n propositional variables. For example,

$$(x \wedge y) \vee (\overline{x} \wedge \overline{y})$$

is a boolean formula which is true (outputs 1) if and only if $x = y$.

A *literal* is a variable x or a negation of variable $\overline{x}$. The formula ϕ is called a *clause* if it is an OR of a set of literals. So, the formula

$$x_1 \vee x_2 \vee \overline{x}_3$$

is a clause while

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

is not.

We say that ϕ is in *conjunctive normal form* (or CNF) if it consists of an "AND of ORs" of variables (equivalently, a conjunction of clauses), where the only negations in the formula appear on the variables. For example, the formula

$$\phi(x_1, x_2, x_3) = (x_1 \vee x_2 \vee \overline{x}_3) \wedge (\overline{x}_2 \vee x_3) \wedge (\overline{x}_1 \vee x_3)$$

is in conjunctive normal form, while neither of the formulas

$$\phi_0(x_1, x_2, x_3) = x_1 \vee (x_2 \wedge x_3),$$
$$\phi_1(x_1, x_2, x_3) = \overline{(x_1 \vee x_2)} \wedge (\overline{x}_2 \vee x_3)$$

are in CNF: the first is not a conjunction of clauses, and in the second there is a negation which is not applied to a variable.

A boolean formula ϕ is *satisfiable* if there is an assignment z to the variables of ϕ such that $\phi(z) = 1$.

Problem 1. CNF-SAT
Input: A boolean formula ϕ in CNF.
Problem: Decide if ϕ is satisfiable.

Remarkably, this problem is NP-Complete (historically, it is the *first* NP-Complete problem). The fact that CNF-SAT is NP-Complete is known as Cook's theorem, named after Stephen Cook.

Theorem 1 (Cook's Theorem)**.** *CNF-SAT is* NP*-Complete.*

Reviewing the summary given at the end of Lecture 9, to show that CNF-SAT is NP-Complete we must, by definition, show that

1. CNF-SAT is in NP.
2. CNF-SAT is NP-Hard, i.e., for every language L in NP, $L \leq_p$ CNF-SAT.

It is not hard to show that CNF-SAT is in NP — we leave this as an exercise (what is the obvious certificate?). The hardness result is much more involved, and if we have time we will prove this in class. For now, we will use Cook's Theorem to show that the following restricted version of CNF-SAT (where we bound the number of variables in each clause) is also NP-Complete. In the next definition let k be a positive integer.

Problem 2. k-SAT
Input: A boolean formula ϕ in CNF where every clause has at most k variables.
Problem: Decide if ϕ is satisfiable.

It turns out that k-SAT is NP-Complete whenever $k \geq 3$ (the case $k = 4$ will be useful for Question 4 on Assignment 3!). The case when $k = 2$ is polynomial-time solvable (we leave this as an exercise).

Theorem 2. *For every $k \geq 3$, k-SAT is* NP*-Complete.*

Proof. We must show that k-SAT is in NP and it is NP-Hard. Well, showing that it is in NP is quite easy: k-SAT is just a subproblem of SAT where the input is restricted to be of a certain form, and SAT is in NP by Cook's Theorem. We focus on proving NP-Hardness, and we

do so by a reduction from CNF-SAT. To be even more concrete we will show that 3-SAT is NP-Hard: this immediately implies that k-SAT is NP-Hard for all $k \geq 3$ by similar reasoning as before.

First we describe the reduction from CNF-SAT to 3-SAT: i.e. a function mapping an instance ϕ of CNF-SAT to an instance ϕ' of 3-SAT. Then we argue that the reduction can be efficiently computed, and finally we argue correctness. Let $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_m$ be a boolean formula in CNF, composed of clauses $C_1, C_2, \ldots, C_m$ defined on n variables $x_1, x_2, \ldots, x_n$. For each $i = 1, 2, \ldots, m$ write

$$C_i = y_{i,1} \vee y_{i,2} \vee \cdots \vee y_{i,\ell_i}$$

where $y_{i,j}$ is either a variable x or a negation $\overline{x}$. If $\ell_i \leq 3$, then we do not need to do anything to C_i and just add it to the new formula ϕ'. So, suppose that $\ell_i > 3$. In this case, we convert C_i into a collection of clauses $C_{i,1}, C_{i,2}, \ldots, C_{i,w}$ such that

1. Every satisfying assignment to C_i can be converted into a satisfying assignment for *all* of the clauses $C_{i,1}$ through $C_{i,w}$ and vice-versa.

2. Each clause $C_{i,j}$ has at most 3 variables.

This conversion proceeds by "breaking C_i up" by adding in some extra variables. To see this, let's first consider the simplest case when $\ell_i = 4$. Then

$$C_i = y_{i,1} \vee y_{i,2} \vee y_{i,3} \vee y_{i,4}$$

where each $y_{i,j}$ is either a copy of a variable or a negation of a variable. Let z be a new variable, distinct from the input variables $x_1, x_2, \ldots, x_n$, and consider the new clauses

$$C_{i,1} = y_{i,1} \vee y_{i,2} \vee z$$
$$C_{i,2} = \overline{z} \vee y_{i,3} \vee y_{i,4}.$$

Observe that if $z = 1$, then the first clause is immediately satisfied and the second clause simplifies to $y_{i,3} \vee y_{i,4}$, while if $z = 0$ then the second clause is satisfied and the first clause simplifies to $y_{i,1} \vee y_{i,2}$. Thus, in order to satisfy both clauses $C_{i,1}$ and $C_{i,2}$ simultaneously, we must set $z = 1$ and satisfy $y_{i,3} \vee y_{i,4}$, or set $z = 0$ and satisfy $y_{i,1} \vee y_{i,2}$. In other words, we can think of the variable z as a "switch": if $z = 1$ then the second half of the variables in C_i must be satisfied, and if $z = 0$ then the first half of the variables in C_i must be satisfied.

We now describe the general transformation. Let

$$C_i = y_{i,1} \vee y_{i,2} \vee \cdots \vee y_{i,\ell_i}$$

and we create $\ell_i - 3$ extra variables $z_{i,1}, z_{i,2}, \ldots, z_{i,\ell_i-2}$. Define the $\ell_i - 2$ clauses $C_{i,1}, C_{i,2}, \ldots, C_{i,\ell_i-2}$

as follows:

$$\begin{aligned}
C_{i,1} &= y_{i,1} \vee y_{i,2} \vee z_{i,1} \\
C_{i,2} &= \overline{z}_{i,1} \vee y_{i,3} \vee z_{i,2} \\
C_{i,3} &= \overline{z}_{i,2} \vee y_{i,4} \vee z_{i,3} \\
&\vdots \\
C_{i,j} &= \overline{z}_{i,j-1} \vee y_{i,j+1} \vee z_{i,j} \\
&\vdots \\
C_{i,\ell_i-2} &= \overline{z}_{i,\ell_i-3} \vee y_{i,\ell_i-1} \vee y_{i,\ell_i}.
\end{aligned}$$

Let ϕ' be the boolean formula in CNF obtained by applying the transformation to every clause in ϕ and taking the $\wedge$ of all of the clauses that result. Clearly every clause in ϕ' has at most three variables, and it should also be obvious that the formula ϕ' can be constructed from the formula ϕ in polynomial time. This is because we can process the clauses C_i in ϕ one at a time, producing the collection $C_1, C_2, \ldots, C_{i,\ell_i-2}$ by scanning across the variables in C_i, "breaking" the clause up and adding in new variables $z_{i,j}$ as we go.

We finally prove correctness. To prove this, we must show that ϕ is satisfiable if and only if ϕ' is satisfiable. First, let $x^* = x_1^*, x_2^*, \ldots, x_n^* \in \{0,1\}^n$ be a satisfying assignment to all of the variables of ϕ, and we construct a satisfying assignment to the variables of ϕ'. This is easy to do, but it is tedious. The variables of ϕ' are split into two sets: we have the old variables $x_1, x_2, \ldots, x_n$, and also the new variables $z_{i,j}$. The old variables will be given the same assignment x^*, and we construct the assignment to the $z_{i,j}$ variables one at a time.

The key observation is this: for each clause C_i in ϕ, since the assignment x^* satisfies C_i, there must be a literal $y_{i,j}$ in C_i such that $y_{i,j} = 1$ under the assignment x^*. There is a clause $C_{i,j'}$ in the collection of clauses corresponding to C_i which will be satisfied thanks to $y_{i,j}$; so, we can satisfy all of the other clauses by choosing appropriate assignments to the switch variables $z_{i,j}$. There are three cases, depending on where $y_{i,j}$ occurs in C_i.

Case 1. $y_{i,j} = 1$ where $j = 1$ or $j = 2$. Set $z_{i,j'} = 0$ for all $j' = 1, 2, \ldots, \ell_i - 3$. The clause $C_{i,1}$ is satisfied since it contains $y_{i,1}$ and $y_{i,2}$, and all other clauses $C_{i,2}, C_{i,3}, \ldots, C_{i,\ell_i-2}$ contain a $\overline{z}_{i,j}$ for some j, and so they are all satisfied.

Case 2. $y_{i,j} = 1$ where $j = \ell_i$ or $j = \ell_i - 1$. Set $z_{i,j'} = 1$ for all $j' = 1, 2, \ldots, \ell_i - 3$. Now the clause C_{i,ℓ_i-2} is satisfied by $y_{i,j}$, and all earlier clauses are satisfied by the assignment to $z_{i,j}$.

Case 3. $y_{i,j} = 1$ where $2 < j < \ell_i - 1$. Set $z_{i,j'} = 1$ for all $j' \leq j - 2$ and $z_{i,j'} = 0$ for all $j' \geq j - 1$. The clause containing $y_{i,j}$ (specifically, clause $C_{i,j-1}$) will be satisfied by the assignment to $y_{i,j}$. All clauses before $C_{i,j-1}$ are satisfied by the assignment $z_{i,j'} = 1$, and all clauses after are satisfied by the assignment $z_{i,j'} = 0$.

In each case we can satisfy all of the clauses $C_{i,j}$ in ϕ' corresponding to the clause C_i in the formula ϕ. By applying this procedure to every clause C_i in the formula ϕ we can iteratively construct a satisfying assignment to the formula ϕ'. Thus, if ϕ is satisfiable then ϕ' is satisfiable.

Conversely, suppose that ϕ' is satisfiable and we show that ϕ is satisfiable. Let x^* be an assignment to the variables of x from the formula ϕ', and let z^* be the assignment to the variables $z_{i,j}$. The assignment to the formula ϕ is easy to construct: we just assign the variables of ϕ exactly according to x^*. To show that this works, observe that for each clause C_i in the formula ϕ and every assignment to the variables $z_{i,j}$ for $j = 1, 2, \ldots, \ell_i - 1$, there is a clause $C_{i,j}$ such that both z literals occurring in $C_{i,j}$ are set to 0. Since ϕ' is satisfied, the corresponding $y_{i,j+1}$ literal must be set to 1, and so the clause C_i is satisfied. $\square$

Tutorial 7: Search-to-decision for Hamilton Path and Max Clique

Tutor: Noah Fleming
Scribe: Robert Robere

March 20, 2016

In this tutorial we give search-to-decision reductions for search versions of some other NP-Complete problems we have seen. If $G = (V, E)$ is a graph recall that a *Hamiltonian path* in G is a path which touches each vertex exactly once. A natural search version of the Hamilton Path problem is the following.

Problem 1. Hamiltonian Path Search
Input: A graph $G = (V, E)$.
Problem: Output a Hamiltonian path in G, or reject if no such path exists.

For completeness, we include a definition of the regular Hamilton path problem.

Problem 2. Hamiltonian Path
Input: A graph $G = (V, E)$.
Problem: Decide if G contains a Hamiltonian path.

We give a search-to-decision reduction from Hamilton Path Search to Hamilton Path.

Proposition 1. *There is a search-to-decision reduction from Hamilton Path Search to Hamilton Path.*

Proof. Following the summary at the end of Lecture 10 in the lecture notes, we need to find a polynomial-time algorithm M computing Hamiltonian Path Search with an oracle for the Hamiltonian Path problem. The intuition for this problem is quite similar to the intuition for the CNF-SAT-Search algorithm given in the same lecture notes: we delete edges one at a time from G, using the oracle to test whether or not the graph that remains has a Hamiltonian path. If so, then we continue, and if not we put the edge back and delete another edge. Eventually every edge in the graph will be deleted except for those edges forming a Hamiltonian path in G.

Algorithm for Hamiltonian Path Search

1. On input $G = (V, E)$.
2. Query oracle to see if G contains a Hamiltonian path. If not, reject.

3. Let $P = G$.

4. For each edge $e \in E$.

   (a) Let P' be the graph obtained by deleting e from P.

   (b) Query the oracle to see if P' contains a Hamiltonian path. If so, set $P = P'$. Otherwise continue.

5. Output P.

The algorithm clearly runs in polynomial time, since it consists of a single loop through all of the edges of the graph of a sequence of polynomial time computations. We focus on proving correctness. First, if G does not contain a Hamiltonian path then the algorithm rejects immediately, so assume that G contains a Hamiltonian path. We claim that the graph P output by the algorithm is a Hamiltonian path in G. Clearly P is a subgraph of G, and by definition the algorithm deletes exactly those edges which would not remove all Hamiltonian paths from G. It follows that P is a Hamiltonian path of G. □

Now we give a search-to-decision reduction for the optimization problem related to Clique.

Problem 3. Max Clique
Input: A graph $G = (V, E)$.
Problem: Output any of the largest cliques in G.

Problem 4. Clique
Input: A graph $G = (V, E)$, a positive integer k.
Problem: Decide if G contains a clique with at least k vertices.

This requires a little bit more work. Given a graph G, we first use the oracle for Clique to determine what the size of the largest clique in G. Then we follow a similar iterative approach to construct the largest clique as we have taken with Hamiltonian Path and CNF-SAT.

Proposition 2. *There is a search-to-decision reduction from Max Clique to Clique.*

Proof. Here is the algorithm, following the sketch before the proposition.

Algorithm for Max Clique

1. On input $G = (V, E)$.

2. Let $k' = 1$.

3. For each $k = 2, \ldots, |V|$.

   (a) Query Clique oracle to see if the graph G has a clique of size at least k. If so, set $k' = k$. If not, break out of the loop.

4. For each vertex $v \in V$.

(a) Let G' be the graph obtained by deleting v and its incident edges from G.

(b) Query the oracle to see if G' contains a k' clique. If so, set $G' = G$. Otherwise continue.

5. Output the vertices of G'.

Again, it is easy to see that this algorithm runs in polynomial time, so we focus on correctness. Let $G = (V, E)$ be any graph, and consider the execution of the algorithm on G. First, observe that after the first loop finishes the value k' is size of the largest clique in G. We argue that G' is a clique with k' vertices in G. This is easy to see: by definition of the algorithm, if we delete any vertex from G' then the resulting graph will not contain a k'-clique. This is only possible if G' *is* a k'-clique, and since G' is a subgraph of G it follows that G contains a k'-clique. □

Tutorial 9: Subset Sum

Tutor: Assimakis Kattis
Scribe: Robert Robere

March 30, 2016

In this tutorial we study an example of a slightly more difficult reduction. The Vertex Cover to Clique reduction involved very simple manipulations on graphs: if you take a connected graph and take the complement, then any clique will turn into an independent set and what remains will be a vertex cover. The 3-SAT to Clique reduction involved an interesting encoding of the 3-SAT clauses into the graph, but it is quite natural once you consider that the "edges" represent "boolean assignment consistency".

In this tutorial we introduce the Subset-Sum problem:

Problem 1. Subset Sum
Input: A list of integers $S = \{a_1, a_2, \ldots, a_n\}$ and an integer k.
Problem: Decide if there is a subset of integers $T \subseteq S$ such that

$$\sum_{a \in T} a = k.$$

As you can probably see the Subset Sum problem is quite simple: for example, if we were given the list of integers $S = \{1, 5, 7, -3, -2\}$ and the integer $k = 4$, then we can take the set $T = \{1, 5, -2\}$ since $1 + 5 - 2 = 4$. However, if instead $k = 15$ then we would be unable to find any set which satisfied it. Despite the apparent simplicity of this problem, we will show it is NP-Complete (try and find an efficient algorithm for it? How would it work? How long would a brute-force algorithm take?). The reduction that we exhibit will be from the Vertex Cover problem: so, somehow, we will have to encode a graph as a list of integers such that the list of integers has a certain sum if and only if the original graph has a vertex cover.

Problem 2. Vertex Cover
Input: A graph $G = (V, E)$ and a positive integer c.
Problem: Decide there is a set of c vertices $S \subseteq V$ such that each edge $(u, v) \in E$ has either $u \in S$ or $v \in S$.

Suppose we have a graph $G = (V, E)$ and a positive integer c, and we want to determine whether or not the graph G has a vertex cover with c vertices. Here is the idea of the reduction.

We define the integer k so that

$$k = c \cdot 2^{2|E|} + \sum_{i=0}^{|E|-1} 2^{2i+1}.$$

Why would we define k like that? It helps to look at the binary expansion of k:

$$k_2 = \underbrace{11\cdots1}_{\log c} \; \underbrace{101010\cdots10}_{\text{10 repeated } |E| \text{ times.}}$$

The first $\log c$ bits of k are used to count the number of vertices in the cover. Then, every pair of bits afterwards corresponds to an edge in E. We add integers to the set S corresponding to the vertices and edges of the graph, such that the edge e_i can be covered if and only if both of its endpoints or one of the endpoints and the edge is included in the set T.

Theorem 1. *Subset Sum is* NP*-Complete.*

Proof. First we give a verifier V for the Subset Sum problem. Intuitively, V will take the list of integers S and the integer k, and it interprets the certificate y as encoding a subset T of the integers S. It sums up the integers appearing in T and checks if the sum is k: if it is, the verifier accepts. If the sum is not k, or if the certificate does not actually encode a subset T of S, then the verifier rejects. This verifier clearly runs in polynomial time: it takes time at most linear in $|S|$ to sum up each of the numbers. We leave arguing the correctness of this verifier to the reader (check back to the other NP-Completeness proofs and the definition of a verifier. What do you need to do to argue correctness?).

Now we prove that Subset Sum is NP-Hard by a reduction from the Vertex Cover problem. Let $G = (V, E)$ be a graph and let c be a positive integer (representing the size of the desired vertex cover). We let $V = \{1, 2, \ldots, n\}$ and $E = \{e_1, e_2, \ldots, e_m\}$. We define k (the target value in the Subset Sum problem) as above, and we instead focus on defining the list of integers S.

The set S will consist of $n+m$ integers: for each vertex $i \in V$ we introduce a *vertex integer* a_i, and for each edge $e = (i, j) \in E$ we introduce an *edge integer* $b_{(i,j)}$. Like k, we define each of the integers by their binary expansion: they will each have $2|E| + 1$ bits. The leading bit will be a 1 if the integer is a vertex integer and 0 if it is an edge integer. There are $2|E|$ bits remaining. For each $\ell = 1, 2, \ldots, |E|$ we associate the two bits in the expansion indexed by $(\ell + 1, \ell + 2)$ with the edge e_ℓ. If b_ℓ is an edge integer, then the two bits at $(\ell + 1, \ell + 2)$ will be set to 01, and all other bits in the expansion are set to 0. If a_i is a vertex integer and i is in the edge e_ℓ, then the two bits at $(\ell + 1, \ell + 2)$ in a_i will be set to 1 as well. In this way, the integers a_i encode all of the edges that the vertex i is contained in, since each edge e containing i will have its corresponding bits in a_is expansion set to 01). The edge integers b_ℓ only have a single 1 bit in their expansion, in the two-bit field corresponding to the edge e_ℓ.

For a moment, we drop the general argument and focus on an example. Suppose that the input graph $G = (V, E)$ was a triangle and $c = 2$. So, $V = \{1, 2, 3\}$ and $E = \{(1, 2), (2, 3), (1, 3)\}$. We introduce three integers a_1, a_2, a_3 corresponding to each of the vertices in the graph, and three integers $b_{(1,2)}, b_{(2,3)}, b_{(1,3)}$ corresponding to each of the edges in

the graph. We define the integers by their binary expansions in the following table (the header row of the table explains which bit fields correspond to which edges).

Integer	Vertex?	$(1,2)$	$(2,3)$	$(1,3)$
a_1	1	01	00	01
a_2	1	01	01	00
a_3	1	00	01	01
$b_{(1,2)}$	0	01	00	00
$b_{(2,3)}$	0	00	01	00
$b_{(1,3)}$	0	00	00	01

The integer k would then be defined as

$$k = 10101010.$$

One possible vertex cover with 2 vertices for this graph is $\{1, 2\}$. If we sum up the two vertex integers a_1, a_2 (corresponding to the vertices in the cover) we get

$$1010001 + 1010100 = 10100101$$

where the addition was perform in base 2. Compare this integer with the integer k:

$$\begin{aligned} k &= 10101010 \\ a_1 + a_2 &= 10100101. \end{aligned}$$

These two integers are identical except for the last four bits. If we add in the integers $b_{(2,3)}$ and $b_{(1,3)}$ to a_1 and a_2 we will get

$$a_1 + a_2 + b_{(2,3)} + b_{(1,3)} = 10101010.$$

What do these edge integers have in common? They each intersect only one of the vertices a_1, a_2! So, if we take all of the vertex integers in the cover and all of the edge integers which contain only one vertex in the cover then the resulting sum will be the integer k: the first $\log c$ bits count the number of vertices appearing the cover. The remaining "bit pairs" will be 10 if the corresponding edge is covered by both of the vertex integers or one of the vertex integers and one of the edge integers.

Now let us return to the general argument. We define the n vertex integers $a_1, a_2, \ldots, a_n$ and the m edge integers $b_1, b_2, \ldots, b_m$ as above. They can clearly be defined in polynomial time from the vertices and the edges of the graph G. We prove the correctness of the reduction.

Suppose that $C \subseteq V$ is a vertex cover of the graph G with $|C| = c$, and we construct a subset T of integers which sum to k. (So, for every edge $(u, v) \in E$ either $u \in C$ or $v \in C$.) Write $C = \{i_1, i_2, \ldots, i_c\}$. We choose the vertex integers $a_{i_1}, a_{i_2}, \ldots, a_{i_c}$ and all of the edge integers $b_{(i,j)}$ where (i, j) contains exactly one of the vertices in the cover C. Since we have c vertices in the cover, when we sum up the integers $a_{i_1}, a_{i_2}, \ldots, a_{i_c}$ we will add $c \cdot 2^{2|E|+1}$ to the sum, which equals the leading bits of k. For each edge $e = (i, j)$ we consider the two bits in k corresponding to e: either i and j are both contained in C (in which case $a_i + a_j$ will

have these two bit fields sum to 10), or one of i and j is in C, in which case the vertex integer chosen and the edge integer $b_{(i,j)}$ will have these two bits fields sum to 10. Thus, the sum will be equal to k.

Now suppose that we have a set of integers $T \subseteq S$ chosen such that summing all of the integers in T will be equal to k, and we construct a vertex cover of size c for the graph G. Exactly c of the integers in T must be vertex integers, as otherwise the leading $\log c$ bits of k would not be equal to the sum of the integers in T. We claim that the c vertices corresponding to these vertex integers form a vertex cover for the graph G. Why? The last $2|E|$ bits of the integer k correspond to the edges of the graph G. Choose any edge $e = (i, j) \in E$ and look at the corresponding pair of bits in e. We know that once we sum the integers in T these two bits will be set to 10. There are only three integers in the set S which can affect the values of these two bits: the integers a_i, a_j, and $b_{(i,j)}$. The set T must contain at least one of the integers a_i, a_j, which means that the edge (i, j) is covered. Since this is true for any edge we must have covered them all, and so this set of vertices makes a cover. □

Tutorial 10: Three Colouring

Tutor: Noah Fleming
Scribe: Lalla Mouatadid (edited by Robert Robere)

March 31, 2016

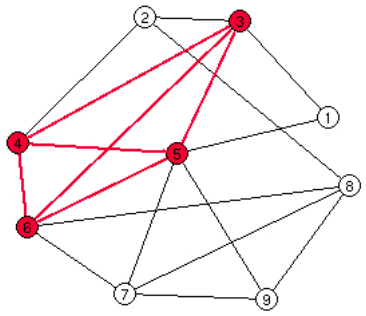
(**Note:** The majority of these notes were written by Lalla Mouatadid for her CSC 373 class at UT-St. George. They have been lightly edited – Thanks Lalla!)

We next show that 3-colouring is NP-complete. What's the **colouring problem** on graphs?

Given a graph $G(V, E)$, the colouring problem asks for an assignment of k colours to the vertices $c : V \to \{1, 2, ..., k\}$. We say that a colouring is **proper** if adjacent vertices receives different colours: $\forall (u, v) \in E : c(u) \neq c(v)$.

The **minimum colouring problem** asks for the smallest k to properly colour G. The k**-colouring problem** asks whether G can be properly coloured using at most k colours.

In this lecture we will show that the colouring problem on arbitrary graphs becomes NP-complete even for $k = 3$! This should be quite surprising — an obvious way to check that a graph *cannot* be 3-coloured is to check if G has a clique of size 4, like in the example below[1]:



Of course, the Clique problem is hard for large k, but for $k = 4$ this is quite easy (just check all subsets of size 4). Thus the fact that even checking three colourability of graphs is NP-Complete shows that 4-cliques are not the *only* obstruction to three colouring — in fact, the obstructions to 3-colouring can be "global" (i.e. using vertices that are separated widely inside the graph), instead of "local", like a 4-clique.

Theorem 1. *3-COLOURING is NP-complete.*

[1] All the pictures are stolen from Google Images and UIUC's algo course.

Where:

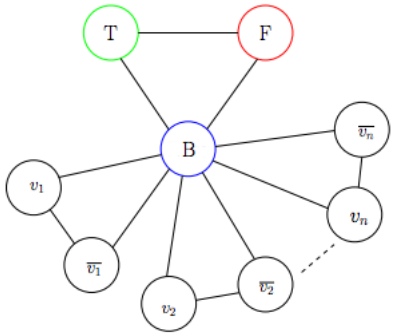
3-COLOURING: Given a graph $G(V, E)$, return decide if there is a proper colouring of G using at most 3 colours.

Proof. To show the problem is in NP, our verifier takes a graph $G(V, E)$ and a colouring c, and checks in $\mathcal{O}(n^2)$ time whether c is a proper coloring by checking if the end points of every edge $e \in E$ have different colours.

To show that 3-COLOURING is NP-hard, we give a polytime reduction from 3-SAT to 3-COLOURING. That is, given an instance ϕ of 3-SAT, we will construct an instance of 3-COLOURING (i.e. a graph $G(V, E)$) where G is 3-colourable iff ϕ is satisfiable.

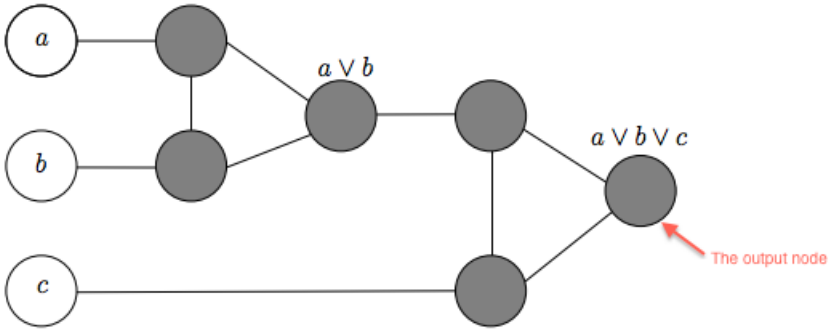
Let ϕ be a 3-SAT instance and $C_1, C_2, ..., C_m$ be the clauses of ϕ defined over the variables $\{x_1, x_2, ..., x_n\}$. The graph $G(V, E)$ that we will construct needs to capture two things: 1. somehow establish the truth assignment for $x_1, x_2, ..., x_n$ via the colors of the vertices of G; and 2. somehow capture the satisfiability of every clause C_i in ϕ.

To achieve these two goals, we will first create a triangle in G with three vertices $\{T, F, B\}$ where T stands for True, F for False and B for Base. Think of $\{T, F, B\}$ as the set of colours we will use to colour (label) the vertices of G. Since this triangle is part of G, we already need 3 colours to colour G. We next add two vertices $v_i, \overline{v}_i$ for every literal x_i and create a triangle $B, v_i, \overline{v}_i$ for every $(v_i, \overline{v}_i)$ pair, as shown below:



Notice that so far, this construction captures the truth assignment of the literals. Since if G is 3-colourable, then either v_i or $\overline{v}_i$ gets the colour T, and we just interpret this as the truth assignment to v_i. Now we just need to add constraints (edges? extra vertices?) to G to capture the satisfiability of the clauses of ϕ. To do so, we introduce the **Clause Satisfiability Gadget**, a.k.a the OR-gadget.

For a clause $C_i = (a \vee b \vee c)$, we need to express the OR of its literals using our colours $\{T, F, B\}$. We achieve this by creating a small *gadget* graph that we connect to the literals of the clause. The OR-gadget is constructed as follows:



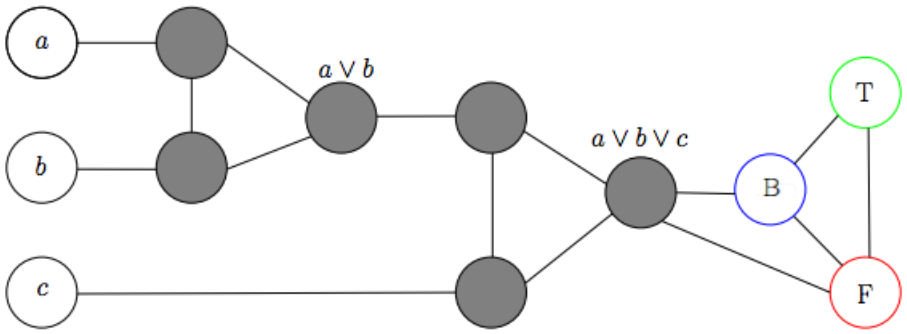
But what is it actually doing? You can think of this gadget graph as a circuit whose output is the node labeled $a \vee b \vee c$. We basically want this node to be coloured T if C_i is satisfied and F otherwise. Notice that this is a two step construction: The node labelled $a \vee b$ captures the output of $(a \vee b)$ and we repeat the same operation for $((a \vee b) \vee c)$.

If you play around with some assignments to a, b, c, you will notice that the gadget satisfies the following properties:

1. If a, b, c are all coloured F in a 3-colouring, then the output node of the OR-gadget **has** to be coloured F. Thus capturing the unsatisfiability of the clause $C_i = (a \vee b \vee c)$.

2. If one of a, b, c is coloured T, then **there exists a** valid 3-colouring of the OR-gadget where the output node is coloured T. Thus again capturing the satisfiability of the clause.

We're almost done our construction. Once we add the OR-gadget of every C_i in ϕ, we connect the output node of every gadget to the Base vertex **and** to the False vertex of the initial triangle, as follows:



Done :) Now we prove that our initial 3-SAT instance ϕ is satisfiable if and only the graph G as constructed above is 3-colourable.

Suppose ϕ is satisfiable and let $(x_1^*, x_2^*, ..., x_n^*)$ be the satisfying assignment. If x_i^* is assigned True, we colour v_i with T and $\overline{v}_i$ with F (recall they're connected to the Base vertex, coloured B, so this is a valid colouring). Since ϕ is satisfiable, every clause $C_i = (a \vee b \vee c)$

must be satisfiable, i.e. at least of a, b, c is set to True. By the second property of the OR-gadget, we know that the gadget corresponding to C_i can be 3-coloured so that the output node is coloured T. And because the output node is adjacent to the False and Base vertices of the initial triangle only, this is a proper 3-colouring.

Conversely, suppose G is 3-colourable. We construct an assignment of the literals of ϕ by setting x_i to True if v_i is coloured T and vice versa. Now suppose this assignment is not a satisfying assignment to ϕ, then this means there exists at least one clause $C_i = (a \vee b \vee c)$ that was not satisfiable. That is, all of a, b, c were set to False. But if this is the case, then the output node of corresponding OR-gadget of C_i must be coloured F (by property (1)). But this output node is adjacent to the False vertex coloured F; thus contradicting the 3-colourability of G!

To conclude, weve shown that 3-COLOURING is in NP and that it is NP-hard by giving a reduction from 3-SAT. Therefore 3-COLOURING is NP-complete. □