CSE 431 lectures 19 and 20

Michael Whitmeyer

February 2025

1 Lecture 19: Clique, Vertex Cover, Hamiltonian Path, and Space Complexity

1.1 Last Time

Recall that $A \in \mathsf{NP}$ iff there exists a deterministic machine V (a "verifier") that runs in time $\mathsf{poly}(|x|)$ s.t.

$$x \in A \iff \exists c \text{ s.t. } V \text{ accepts } \langle x, c\rangle.$$

Also recall that A is NP**-complete** iff

1. $A \in \mathsf{NP}$ and
2. $B \leq_m^P A$ for some NP-complete language B.[1]

We saw a proof that

$$\mathsf{INDSET} := \{\langle G, k\rangle : \text{there exists an independent set of size } k \text{ in } G\}$$

is an NP-complete problem via a reduction from $\mathsf{3SAT}$.

Today:

1. Proof that the k-clique and k-vertex cover problems are NP-complete.
2. Proof that the hamiltonian cycle problem is NP-complete.
3. Some space complexity definitions (if time).

1.2 CLIQUE is NP-complete

Let us formally define the language:

$$\mathsf{CLIQUE} := \{\langle G, k\rangle : G \text{ has a set of } k \text{ vertices that form a clique}\}.$$

Lemma 1.1. $\mathsf{CLIQUE} \in \mathsf{NP}$

Proof. The certificate is simply the set of k vertices that form a clique. Poly time to check, and there exists a certificate iff G actually has a clique of size k. □

Lemma 1.2. CLIQUE *is* NP*-hard.*

Proof. We define a reduction from INDSET. Let $\langle G, k\rangle$ be the input to our independent set problem. Construct $\overline{G}$ s.t. $(u, v) \in E(\overline{G})$ iff $(u, v) \notin E(G)$.

If G has an independent set of size k, then the same set of vertices now forms a clique of size k. Moreover, if $\overline{G}$ has a clique of size k, then it is immediate that G will have an independent set on those same vertices.

Finally, $\overline{G}$ is constructible in polynomial time. □

The fact that CLIQUE is NP-complete follows from Lemma 1.1 and Lemma 1.2.

[1] Recall that this follows from the transitivity of poly-time mapping reductions.

1.3 VERTEXCOVER is NP-complete

Let us formally define the problem:

$$\mathsf{VERTEXCOVER} := \{\langle G, k\rangle : G \text{ has a vertex cover of size } k\}.$$

Recall that a vertex cover is simply a subset $U \subseteq V$ of vertices such that every edge $e \in E$ has at least one endpoint in U.

It is easy to verify that VERTEXCOVER is in NP (the witness is the set of vertices in the cover).

We shall once again prove that VERTEXCOVER is NP-complete via a reduction from INDSET. The following lemma should be very suggestive as to how the reduction will go.

Lemma 1.3. *We have that U is an independent set in G iff $V(G) \setminus U$ is a vertex cover of G.*

Proof. Suppose U is an independent set, so there are no edges in the subgraph induced by U. Then if we take the set of vertices other than U, they must touch every edge, and hence be a vertex cover.

On the other hand, if we have a vertex cover, and we look at the vertices outside that cover, there can be no edges between them, otherwise we did not have a vertex cover. □

Using Lemma 1.3, a reduction from INDSET to VERTEXCOVER is immediate: simply map $\langle G, k\rangle$ to $\langle G, n-k\rangle$, where n is the number of vertices in G. The fact that the reduction works is guaranteed by Lemma 1.3, and it is clear that the reduction is in polynomial time.

Corollary 1.4. *We have that*

$$\mathsf{INDSET} \leq_m^P \mathsf{VERTEXCOVER},$$

and therefore that VERTEXCOVER *is* NP*-complete.*

1.4 HAMPATH is NP-complete

Let us formally define the language:

$$\mathsf{HAMPATH} = \{\langle G, s, t\rangle : G \text{ is a directed graph with a path from } s \text{ to } t \text{ that touches every vertex exactly once.}\}$$

It should be easy to recognize that $\mathsf{HAMPATH} \in \mathsf{NP}$, since a verifier, given a supposed path, can check it easily in polynomial time.

It is a bit less clear how to make a reduction from INDSET work here since now we are dealing with a directed graph. Instead, our reduction will be directly from 3SAT.

Lemma 1.5.

$$\mathsf{3SAT} \leq_m^P \mathsf{HAMPATH}.$$

Proof. Given a 3SAT instance φ, we need to construct a graph G_φ (that depends on the variables/clauses in φ) such that G_φ has a path from start node s to final node t which touches all vertices once iff φ is satisfiable.

The first idea is that we would like to force any path from s to t to "choose" a value for each variable. This can be accomplished as shown in Figure 1. For each diamond in Figure 1, there are only two ways to get through it that touch all vertices, and we have (arbitrarily) said that taking the purple path in diamond i corresponds to setting $x_i = 1$.

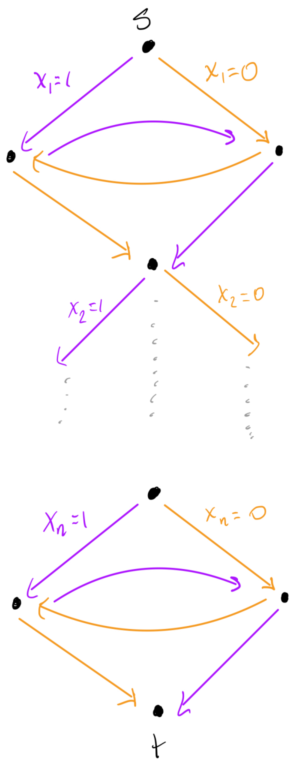

Figure 1: The initial gadget that forces choices for variable values.

So far, however, we have not used anything about the clauses of φ! To fix this, we will add a vertex to our graph for each clause C in φ. Somehow, we need to make it so that if clause C contains x_i (and it is not negated), then the vertex corresponding to C is reachable if we took the "purple path" (i.e. set $x_i = 1$) in diamond i, but is not reachable (from diamond i, at least) if we took the orange path (i.e. set $x_i = 0$). This rough idea is illustrated in Figure 2.

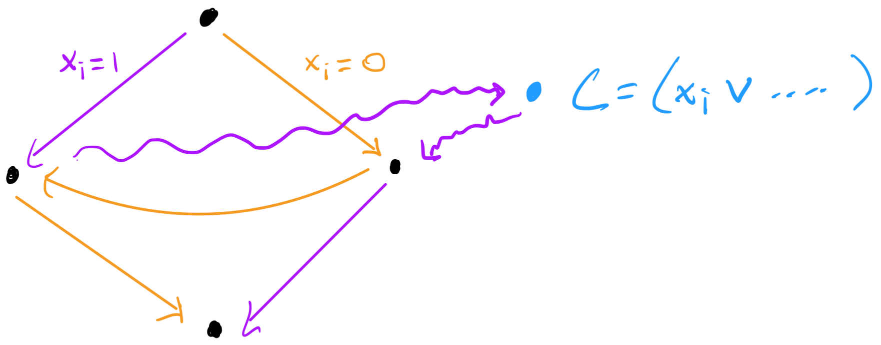

Figure 2: Roughly what we would like to happen in order to incorporate the clauses of φ.

The actual execution of this idea requires a bit of finesse, since x_i and its negation can appear in many clauses! To deal with this, we will add dummy nodes in the middle of each diamond. That way, we will be able to make it so that each clause that contains a particular variable will be reachable from that diamond iff we set that variable the correct way. This is best illustrated with a picture, see Figure 3.

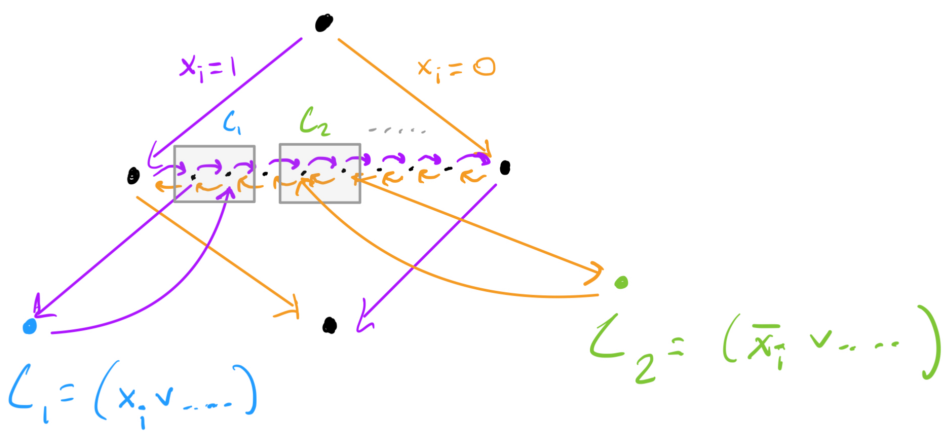
$x_i=1$
$x_i=0$
C_1
C_2
$C_1 = (x_i \vee \ldots)$
$C_2 = (\overline{x_i} \vee \ldots)$

Figure 3: Refining the idea. We add dummy nodes ($3m$ of them, if m is the number of clauses of φ) in the middle of each diamond. We then add wires that allow a detour on the purple path if x_i appears in C, and add wires that allow an orange detour if $\overline{x_i}$ appears in C, and no wires otherwise. Note that there are "separator" nodes between clause detours – this will be important later.

After doing the wiring illustrated in Figure 3, we have completed our graph G_φ. Note that the construction of G_φ will take polynomial time.

It therefore remains to argue that there is a hamiltonian path from s to t in G_φ if and only if φ is satisfiable.

1. ($\Longleftarrow$): Suppose φ is satisfiable, and let $\tilde{x}$ be the assignment. Then it is clear by construction of G_φ that there is a hamiltonian path, by following the purple path in diamond i if $\tilde{x}_i = 1$, and the orange path otherwise, and taking a detour to a clause node whenever it is possible.

2. ($\Longrightarrow$): this is the trickier direction. We need to argue that if we have a hamiltonian path from s to t, then there must a satisfying assignment to φ.

   Call a hamiltonian path in G_φ "normal" if it traverses each diamond in order. Clearly, if our hamiltonian path is normal, we can deduce a satisfying assignment, by picking $x_i = 1$ if the path goes on the purple edges in the i'th diamond, and otherwise choosing $x_i = 0$.

   It therefore suffices to argue that every hamiltonian path from s to t must be normal. In other words, the situation depicted in Figure 4 cannot happen.

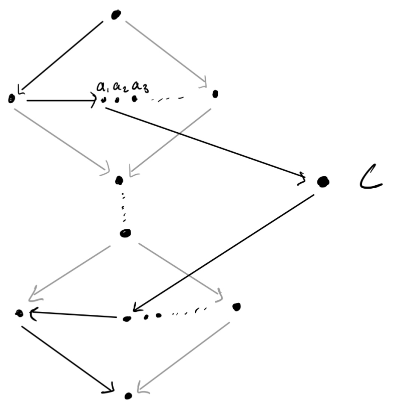

Figure 4: This cannot occur.

The reason for this is because we added the separator nodes! We claim that a_2 in Figure 4 will never be visited by the supposed hamiltonian path in the figure. If a_2 was a separator node, then it could only be visited by a path from a_1 or a_3. It is clearly not being visited from a_1, and if it is visited from a_3, then it must continue on to a_1, which is a contradiction. Otherwise, if a_3 were a separator node, then a_1 and a_2 are a clause pair, so a_2 is connected to a_1, c, and a_3. But a_1 and c both point elsewhere, and a_3 cannot point to a_2 since then there's no other free neighbors.

This completes the proof. □

From Lemma 1.5 and the fact that HAMPATH is in NP, we get that HAMPATH is an NP-complete problem.

Actually, we can say even more. We can define an undirected version of the hamiltonian path problem:

$$\mathsf{UHAMPATH} := \{\langle G, s, t\rangle : G \text{ is an undirected graph with a hamiltonian path from } s \text{ to } t.\}.$$

Lemma 1.6. UHAMPATH *is* NP-*complete.*

Proof. We exhibit a reduction HAMPATH $\leq_m^P$ UHAMPATH. We locally replace each node with three nodes, as illustrated in Figure 5.

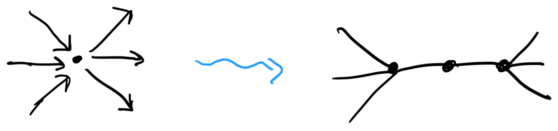

Figure 5: Making an directed graph undirected.

Observe that if the original directed graph has a hamiltonian path, then so will the new graph. On the other hand, if the new graph has a hamiltonian path then all the middle nodes must be touched, so the original directed graph will too. □

1.5 Starting Space Complexity

We now switch gears to space complexity. Just like time, we can view space as a resource (in practice, this can be a very important and limiting resource).

In order to define space complexity, we use the model described in Chapter 8.4 in Sipser.[2] We introduce a "read-only" two-tape Turing machine, where one tape contains the input and is *read-only*, while the other tape is a "work tape". There are two heads (one for the reading the input, one pointing to the work tape(s)), but we only allow modifications to the work tape.

Definition 1.7. The space used by a read-only TM M is a function $S : \mathbb{N} \to \mathbb{N}$ such that

$$S(n) := \{\text{maximum number of work tape cells used by } M \text{ on any } w \in \Sigma^n.\}$$

Similar, if we have a nondeterministic machine:

Definition 1.8. The space used by a read-only NTM M is a function $S : \mathbb{N} \to \mathbb{N}$ such that

$$S(n) := \{\text{maximum number of work tape cells used by } M \text{ on any computation path for any } w \in \Sigma^n.\}$$

With this model in hand, we can define space complexity.

[2]Notably, we are not using the model defined in Chapter 8.1 of Sipser. This follows a now-standard practice/definition for space complexity

Definition 1.9. We define

$$\mathsf{SPACE}(S(n)) := \{A : A \text{ is decided by a TM } M \text{ with read only input tape using space } O(S(n))\},$$

and similarly

$$\mathsf{NSPACE}(S(n)) := \{A : A \text{ is decided by a NTM } M \text{ with read only input tape using space } O(S(n))\}.$$

Observe that if A is a regular language, then $A \in \mathsf{SPACE}(1)$.

Theorem 1.10. $\mathsf{SAT} \in \mathsf{SPACE}(n)$.

Proof. Simply copy the inputted formula onto the work tape, and then iterate through all possible assignments, seeing if they work. Crucially, reuse space when doing this. This takes exponential time, but only linear space. □

Finally, we can define polynomial space.

Definition 1.11. We define

$$\mathsf{PSPACE} := \bigcup_k \mathsf{SPACE}(k)$$

and

$$\mathsf{NPSPACE} := \bigcup_k \mathsf{NSPACE}(k).$$

2 Lecture 20: Space Complexity, Savitch's Theorem

Agenda:

1. Relating Time and Space
2. Savitch's theorem

2.1 Relating Time and Space

Theorem 2.1. *We have that*

1. $\mathsf{TIME}(T(n)) \subseteq \mathsf{SPACE}(T(n))$ *and* $\mathsf{NTIME}(T(n)) \subseteq \mathsf{SPACE}(T(n))$*, and*
2. *For* $S(n) \geq \log n$*, we have* $\mathsf{SPACE}(S(n)) \subseteq \mathsf{TIME}(2^{O(S(n))})$*, and*
3. *for* $S(n) \geq \log n$*, we have that* $\mathsf{NSPACE}(S(n)) \subseteq \mathsf{TIME}(2^{O(S(n))})$.

Proof.

1. For the first item, even if our Turing machine runs in nondeterministic time $T(n)$, this means that every possible computation path takes at most this much time. In particular, each computation path can take at most $O(T(n))$ space. So, we can simply simulate all possible computation paths (we use a counter to keep track of which one we are on, which takes an additional $O(T(n))$ space), erasing each one after we are done, and accept if any of them accept, which is exponential time but $O(T(n))$ space, as desired.

2. If a machine only uses $O(S(n))$ space, then there are only $n \cdot 2^{O(S(n))}$ many possible configurations of the machine, since we have n possibilities for where the input tape head is pointing, $O(S(n))$ options for the work tape head, $2^{O(S(n))}$ for the work tape contents, and of course $O(1)$ options for the state.

Putting this together, along with the assumption that $S(n) \geq \log n$, we get that there are at most $2^{O(S(n))}$ possible configurations for our machine. We can then simulate the machine for this many steps[3], and if we haven't already accepted/rejected, then by the pigeonhole principle a configuration will have repeated itself, and the turing machine will be looping, so we can halt and reject.

3. This final one is a bit trickier. The most naive algorithm would be to simulate each possible nondeterministic computation path. Each possible path can only be of length $2^{O(S(n))}$ by what we have already argued. But this fails because now the "tree of possible computation paths" has depth $2^{O(S(n))}$, so would have $2^{2^{O(S(n))}}$ many leaves/paths!

   We must do something a bit more clever. The key point is that even though there are doubly exponentially many possible computation paths, there are still only $2^{O(S(n))}$ many possible configurations for our machine. This leads to the following definition.

   Definition 2.2 (Configuration Graph)**.** For a given space-bounded TM M and input x, define its *configuration graph* $G_{M,x}$ as the directed graph with vertices corresponding to configurations. There is a directed edge between configuration C and configuration D iff it is possible for the machine to go from configuration C to configuration D in one step.

   Note that in Definition 2.2, the outdegree of any node is bounded by some constant b that depends on the code (transition function) of M. If M is deterministic, then the outdegree is just 1 for all vertices. The starting configuration looks like $C_0 = (q_0x, \sqcup)$, since the work tape starts out empty. We may furthermore assume without loss of generality that there is a single unique accepting configuration $C_{acc} = (q_{acc}x, \sqcup)$, since if not, we can always reprogram M to erase its work tape and return the input head to the beginning before accepting.

   With Definition 2.2 in hand, observe that M accepts on input x iff there exists some path from C_0 to C_{acc} in $G_{M,x}$. Therefore, we can run BFS (or DFS) on this graph with $2^{O(S)}$ vertices, which takes time $(2^{O(S)})^2 = 2^{O(S)}$, and accept iff we find C_{acc} is reachable.

□

2.2 Savitch's Theorem

In the previous subsection, we saw how to simulate NSPACE$(S(n))$ with time $2^{O(S(n))}$ and space $2^{O(S(n))}$.[4]

A natural question is the following: Is the $2^{O(S)}$ space bound necessary when simulating a nondeterministic space $S(n)$ machine? Surprisingly, the answer is no!

Theorem 2.3 (Savitch)**.** *For all $S(n) \geq \log n$, we have that*

$$\mathsf{NSPACE}(S(n)) \subseteq \mathsf{SPACE}(S(n)^2).$$

Proof. Let N be the nondeterministic machine running in space $S(n)$. Our idea will be to look for a path in $G_{N,x}$ *without* writing down the whole graph. Note that if a path from C_0 to C_{acc} exists in $G_{N,x}$, then its length is at most $2^{O(S)}$.

Towards this end, let us define the following useful function.

Definition 2.4. Define the function $\text{CANYIELD}_t : V(G_{N,x}) \times V(G_{N,x}) \to \{0,1\}$ as

$$\text{CANYIELD}_t(C, D) = \begin{cases} 1 & \text{if } D \text{ is a reachable configuration after at most } t \text{ steps starting from } C \\ 0 & \text{otherwise.} \end{cases}$$

[3] This requires adding a timer, but this will only be an additional $O(S(n))$ bits.
[4] We needed to build and store the configuration graph

Let's start with some basic observations. First, when $t = 0$, $\text{CANYIELD}_0(C, D) = 1$ iff $C = D$. Second, we are interested in $\text{CANYIELD}_t(C_0, C_{acc})$ for $t = 2^{O(S(n))}$.

Most crucially, we have that

$$\text{CANYIELD}_t(C, D) = 1 \iff \exists C_{mid} \text{ s.t. } \Big(\text{CANYIELD}_{t/2}(C, C_{mid}) = 1\Big) \wedge \Big(\text{CANYIELD}_{t/2}(C_{mid}, D) = 1\Big).$$

Let us now write down a recursive definition of CANYIELD, which also happens to be an algorithm! The recursive nature of the algorithm allows us to reuse space. Let's see the algorithm now, and then analyze its space usage more in depth after.

Algorithm 2.5. $\text{CANYIELD}_{2^{O(S(n))}}$ = "on input C and D:

```
1. if $t = 1$, then accept iff there is an edge from $C$ to $D$ in $G_{N,x}$.
2. For each possible $C_{mid} \in V(G_{N,x})$):
3.     Run $\text{CANYIELD}_{t/2}(C_0, C_{mid})$
4.     Run $\text{CANYIELD}_{t/2}(C_{mid}, C_{acc})$
5.     If both of the previous steps accept, then accept.
6. if we have not yet accepted, reject."
```

Our deterministic (low space) machine is thus as follows. Let d be some constant such that the number of possible configurations for N is at most $2^{dS(n)}$.

M = "on input x:

Output the result of $\text{CANYIELD}_{2^{dS(n)}}(C_0, C_{acc})$"

Note that C_0 and C_{acc} (because of our assumption) just depend on x, so M knows what they are.

There are at most $dS(n)$ recurisive levels, and we must store t, C_1, C_2 (which each require $O(S(n))$ space) in a stack while going down the recursive tree. This gives us an $O(S(n)^2)$ bound on our space usage.

Crucially, we reuse space – meaning when we reach the end of one of our recursive branches, we erase everything we had recorded, and start anew down a new recursive branch. Our total space is thus $O(S(n)^2)$.

There is one small detail to handle: on input x, M does not know what $S(n)$ is. The solution is simply to try all $S(n) = 1, 2, 3, ...$! Of course, if we ever accept, then that is great. But how do we know when to stop? We can use CANYIELD_i to check if *any* configuration of length $i + 1$ is reachable from the start configuration. If not, we can halt and reject. □

We have the following immediate corollary.

Corollary 2.6. *We have that*

$$\mathsf{PSPACE} = \mathsf{NPSPACE}.$$