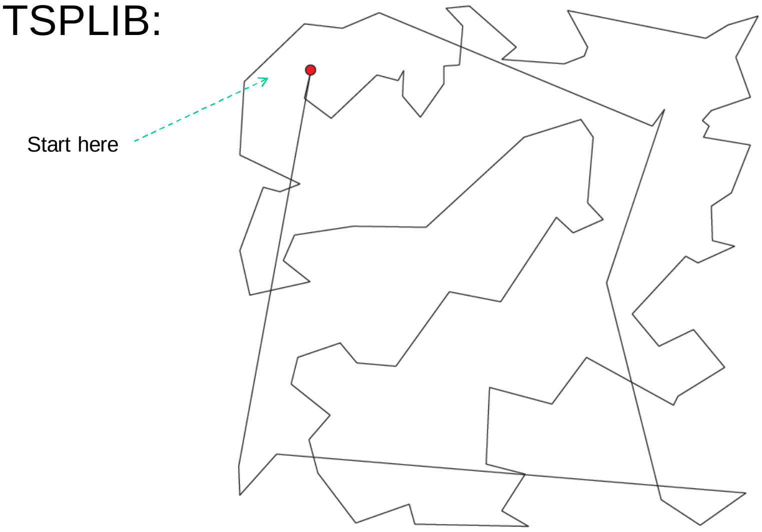
NP Completeness

- Traveling Salesperson Problem (*TSP*):
  - Given a weighted undirected graph $G = (V, E)$, find a cycle containing every vertex exactly once such that the sum of weights of edges in the cycle is a minimum.
  - A greedy algorithm does not work (see next slide).
  - An exhaustive search will "work" but takes much too long for even moderate values of n.
    - Exhaustive search: Trying each permutation of the cities involves testing $(n - 1)!$ sequences.
  - There is no known polynomial time algorithm that is guaranteed to find the minimal cost tour.
  - There are many polynomial time algorithms to generate *approximations* to the optimal solution.

TSP & Greedy (Near Neighbour)

- Consider NN "solution" for rd100 taken from TSPLIB:



TSPLIB: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/

Practical vs. Impractical

- A simple characterization of algorithms:
  - We will consider polynomial time algorithms to be "practical" or efficient
    - (even with large exponents)
  - Algorithms that require exponential time (or worse) are regarded as "impractical".

The Question of Efficiency

- What do we do if we have a problem like TSP that has no polynomial time solution?

  If we could *prove* that no polynomial time algorithm can be designed to solve the problem,

  then we could go about looking for approximate solutions or heuristics with knowing that no practical algorithm exists to find the optimal solution.
  - There is a set (class) of problems that **require exponential solutions** and this can be proven.
  - But there are many other problems (e.g. TSP) for which such a proof is **not** known despite considerable effort by researchers.

- So instead we must go for a lesser prize:
  - We prove that the problem is "NP-hard".
    - That is: it is essentially **equivalent** to other problems for which no known polynomial time solution exists.
    - Such a proof does not show that a polynomial time solution does not exist but only that thousands of other experts have never found an efficient solution either.
  - Why is this important?
    - If you have an NP-hard problem you have two choices:

    1. You can decide that all the experts have missed some vital point and so you set about looking for a polynomial time solution…
    2. You decide that you are not likely to do better than the experts and set about looking for an approximation or heuristic algorithm that gives a reasonable solution.

Optimization vs. Decision Problems

- Optimization problems:
  - Find a value, object or configuration that optimizes some function.
    - For example: in TSP we are looking for a least cost tour.
- Decision problems:
  - The solution of a decision problem is "yes" or "no".
  - Some examples:
    - Is a given number a prime number?
    - In the TSP-D problem we are given a value B and we want to know whether there is a tour with length $\leq B$.
  - **NP-completeness theory** deals with **decision problems**.

D for Decision

- Observation:
  - If the cost function is easy to evaluate, then the decision problem can be no harder than the corresponding optimization problem.
    - For example: If we can find the minimum length of a TSP tour, then we can compare it to value B and thus solve the decision problem.
  - Often the reverse is also true:
    - If we can solve the decision problem in polynomial time then we can solve the optimization problem in polynomial time as well.

Class P (Polynomial)

- Review:
  - What does "running time" mean?
    - The running time of an algorithm A is a function $T_A(n)$ where n is the size of the input instance.
    - $T_A(n)$ = worst case time to solve an instance of size n.
  - What is the size of an instance?
    - Size = number of bits needed to encode the problem instance.
- Definition:
  - A decision problem Q belongs to the class of problems P if and only if there exists a polynomial time algorithm solving problem Q.

- Example 1:
  - Recall Bentley's problem. Is it in class P?
  - No. It is not a decision problem.
- Example 2:
  - Reformulate Bentley's problem as a decision problem:
    - Given an array $A[1..n]$ of integers and an integer B, is there a subarray with sum $\geq B$?
  - Is this in class P? Yes.

- Example 3:
  - The Decision Coin Changing Problem:
    - Given a denominational system with n different coins, a sum S, and an integer B, is it possible to pay out sum S with $\leq B$ coins?
  - Is this problem in class P?
  - How many bits are needed to encode the input?

$$O\left(\log n + \sum_{i=1}^{n} \log c_i + \log S + \log B\right)$$

Storage space for S.

  - What is the time for the best known algorithm? $O(n \cdot S)$.
  - Is it polynomial in the size of the input? No.
    - Polynomial in S means exponential in $\log(S)$.
    - So we are unable to prove that the problem is in P.

Non-deterministic Algorithms

- Non-deterministic algorithms are only defined for decision problems.
- Think of them as algorithms that encode massive searches.
  - For example in the TSP problem we can try all possible tours in the graph.
  - These searches could be performed in parallel.
    - For example we could have a large cluster of Linux boxes (exponentially many of them) with each box responsible for trying a few choices.

Non-deterministic Algorithms

- We will make use of the following special statements to be used in pseudocode representing non-deterministic algorithms:
  - **Accept**: finish the computation with an answer of "yes".
  - **Reject**: finish the computation with an answer of "no".
  - **Try** $\boldsymbol{k = \{i, \ldots, j\}}$:
    - Try all possibilities for k, ranging from i to j in parallel (as if each possibility is tested on its own computer).
      - In this statement k is an important variable that must be set with some particular value if the computation is to finish with 'accept', (assuming the decision is "yes").
        - » There may be several such values to be tried.
      - If, instead, the decision is "no" then the setting of k is arbitrary.
    - Note that 'Try' takes a long time on a single computer.

A TSP-D “Solution”

- Given $V = \{1, 2, \ldots, n\}$ and an adjacency matrix representation, consider TSP-D solved with a **non-deterministic** algorithm:

```
function TSP_D
  visited[i] := false for all vertices
  last_visited := 1; visited[1] := true; length := 0;
  repeat n – 1 times
     try next_visited between 1 and n  //Going parallel here!
       if (visited[next_visited]) then reject;
       // we cannot visit a single vertex twice
       visited[next_visited] := true;
       length := length + w(last_visited, next_visited);
       last_visited := next_visited;
       length := length + w(last_visited, 1); // Finish tour
       if length ≤ B then accept;   else reject;
```

Some Important Issues

- Be sure to understand the following concepts associated with the **`try`** statement:

  `try next_visited between 1 and n`
  `//Going parallel here!`

  – An algorithm that can solve a problem in polynomial time does not need a **`try`** statement.

  – As demonstrated in the last slide, the **`try`** statement can be used to solve TSP but (so far) no one has been able to prove that it is **necessary** for the solution of TSP.

  – However, there are problems of “exponential nature” such that you can prove that **`try`** is needed.

  – Finally, there are “non-computable” problems that cannot be solved even if the **`try`** statement was available!

Running Time of a Non-deterministic Algorithm

- Definitions:
  - An "accepting computation" is a computation that ends on the "accept" statement.
  - The running time of a non-deterministic algorithm A on instance x is the shortest time taken for an accepting computation (producing a "yes").
    - Recall that each "try" is executed on its own computer.
    - It is undefined if x leads to a "no" decision.
  - Running time of a non-deterministic algorithm A is denoted by $T_A(n)$ and is the longest running time over all "yes" instances of size n.

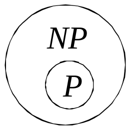
Class *NP* (Non-deterministic Polynomial)

- Definition:
  - A decision problem Q belongs to the class of problems NP if and only if there exists a polynomial time non-deterministic algorithm solving the problem Q.
  - Note:
    - Non-determinism is one of the most important concepts in computer science.
    - It was first introduced in formal languages.
    - It is impractical as the computer cluster would require an exponentially large number of computers.

$P \subseteq NP$

- All problems in P are also in NP.
  - They simply never use the “try” statement in the pseudocode.

- Are there problems that are in NP but not in P?
  - In other words, do we have: $P \neq NP$?



  - Most famous unsolved problem in computer science.
    - The general view is that it is true but nobody has been able to prove it so far.

Reductions (1)

- Reduction is a general technique for showing that one problem is no harder than another.
  - Suppose we have two decision problems: P_x and P_y such that P_y can be solved using algorithm A_y.
  - Suppose further that we do not have an algorithm A_x that can solve P_x but after some clever observations we suspect that we can solve P_x by making it “look like” P_y.

Reductions (2)

– More precisely:

Suppose x is a problem instance of P_x.

- We need a function f that can transform x into an instance y of problem P_y.
- We then use A_y to solve y.
- Our function f must work in such a way that the decision made by A_y on problem y is taken to be the answer for decision problem x.

Reductions (3)

– In other words:
  - For **every** instance of x in P_x, the decision made when solving the problem x is true if and only if the decision made when solving the problem $f(x)$ is true.

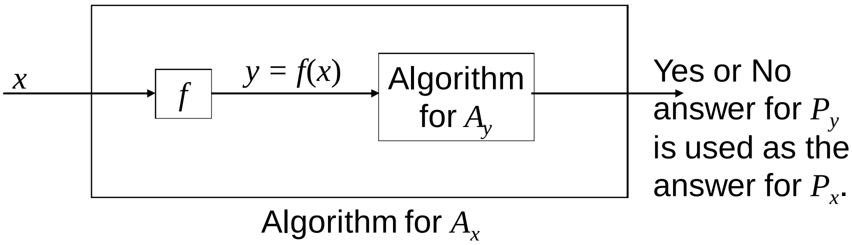
– We will say that problem P_x has been reduced to problem P_y.
  - This is written as: $P_x \leq P_y$.

– Of course all this depends on whether a reasonable function f can be found.
  - If the function f can be found we say that P_x is reducible to P_y.

– What does "reasonable" mean here?
  - The execution time required for f to do its transformation must be polynomial in the size of the instance x.

Polynomial Reductions (1)

- Suppose P_x is reducible to P_y.
  - That is: $P_x \leq P_y$.
  - We can diagram the situation as follows:



Polynomial Reductions (2)

- It should be reasonably clear that A_x, the algorithm for P_x is using A_y as a “subroutine”.
  - So, if f is polynomial in execution time and **if** A_y is polynomial in execution time **then** so is A_x.
- It should also be clear that P_x is no harder than P_y.
  - (within a polynomial factor)
  - For example, A_x would not be exponential in execution time if A_y was polynomial in execution time.
    - Now note the significance of the symbol “$\leq$”.
  - Often, to emphasize the polynomial aspect of the transformation we may write: $\leq_p$.
- Observation:
  - The reducibility relation is transitive.

Hamiltonian Circuits (1)

- Definition:
  - Given an undirected graph $G = (V, E)$, a Hamiltonian circuit is a tour in G passing through every vertex of G once and only once.
  - The decision problem *HAM* asks if a graph G has a Hamiltonian circuit.

Hamiltonian Circuits (2)

- Reduction of $HAM \leq_P TSP\text{-}D$:
  - We want to show that *TSP-D* can be used to solve *HAM*.
    - How do we do this? We need the function f.
  - Solution:
    - Suppose we have a graph G to be used as an input instance for *HAM*.
    - We want $f(G)$ to be an input for *TSP-D*.
    - In other words, we need f to create an undirected graph G_{TSP} and a threshold B that act as input to *TSP-D*.

Hamiltonian Circuits (3)

Our description of what f does:

– We create a complete graph G_{TSP} with the following edge weights:

- $w(u, v) = 0$ if (u, v) is an edge in G.
- $w(u, v) = 1$ otherwise.

– Note that graph G has a Hamiltonian circuit iff graph G_{TSP} has a tour of total length at most 0.

(Our B value is 0.)

- So we have used *TSP-D* with input $(G_{TSP}, 0)$ to solve problem *HAM*.

3-*SAT*

- 3-Satisfiability:
  - Consider a set of Boolean variables $U = (u_1, u_2, \ldots, u_m)$ and a logical formula of the form:

$$\left(a_{1,1} \vee a_{1,2} \vee a_{1,3}\right) \wedge \left(a_{2,1} \vee a_{2,2} \vee a_{2,3}\right) \wedge \ldots \wedge \left(a_{n,1} \vee a_{n,2} \vee a_{n,3}\right)$$

  where $a_{i,j}$ is either a variable u_k from U or its negation $\neg u_k$.

    - $a_{i,j}$ is called a *literal*.

Note: m variables
n clauses

  - Question:
    - Is there an assignment of *True* and *False* values for the u_k variables so that the formula is **satisfied**, that is, it evaluates to True?

Note: Each clause must evaluate to *True*.

- Examples:
  - Given the formula:

$$(u_1 \vee \neg u_3 \vee \neg u_4) \wedge (\neg u_1 \vee u_2 \vee \neg u_4)$$

    - We see that this is satisfiable, for example, by making the assignment $u_1 = True$, $u_2 = True$, $u_3 = True$, $u_4 = True$.

  - Given the formula:

$$(\neg u_1 \vee \neg u_1 \vee \neg u_1) \wedge (u_1 \vee \neg u_2 \vee \neg u_2) \wedge (u_1 \vee u_2 \vee \neg u_3) \wedge (u_1 \vee u_2 \vee u_3)$$

    - There is no assignment that will make this $True$.

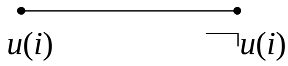
Vertex Cover (1)

- Vertex Cover:
  - We are given a pair (G, K) where $G = (V, E)$ is an undirected graph and K is a number.
  - Does there exist a subset of at most K vertices V_c such that for each edge (x, y) in E, either x is in V_c or y is in V_c?
    - That is, every edge is "covered" by the set V_c.
- Our task:
  - Show that $3\text{-}SAT \leq_P VC$.
    - In other words:
      We are given a $3\text{-}SAT$ Boolean formula and we want to construct a graph G and define a value K such that the graph G has a vertex cover of size K iff the formula is satisfiable. (This last statement is defining what the transformation f should do!)

Vertex Cover (2)

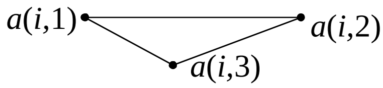
– Construction of the graph G involves three types of edges defined as follows:

- Type (1): For every variable u_i: (2 vertices, 1 edge):



  – To cover the edge, at least one of the vertices u_i or $\neg u_i$ must be in the cover.

- Type (2): For every clause $(a_{i,1} \vee a_{i,2} \vee a_{i,3})$:



  – To cover edges in the triangle, at least two of the vertices $a_{i,1}$, a_{i2}, $a_{i,3}$ must be in the cover.

- Type (3): If literal $a_{i,j} = u_k$ then connect $(a_{i,j}, u_k)$.
  If literal $a_{i,j} = \neg u_k$ then connect $(a_{i,j}, \neg u_k)$.

Every vertex cover of this graph must have at least $m + 2n$ vertices.

Notes on Construction of Graph G

– Any vertex cover for the constructed graph must have at least $m + 2n$ vertices where m is the number of variables u_i and n is the number of clauses.

– Our reduction strategy will require the vertex cover to have at most $m + 2n$ vertices so the only possibility is that there are exactly $m + 2n$ vertices in the cover.

- This means that Type 1 edges are covered by only one vertex and Type 2 edges (in the triangles) are covered by exactly 2 vertices.
- Type 3 edges will be covered by vertices that are already covering Type 1 edges or Type 2 edges.

– Note: the construction can be done in polynomial time.

Proof that $3\text{-}SAT \leq_P VC$ (1)

- We need to show that the $3\text{-}SAT$ problem is satisfiable *if and only if* there is a cover of G with exactly $m + 2n$ vertices.
  - We will prove this in the following order:
    - satisfiability $\Rightarrow$ existence of the cover
    - existence of a cover $\Rightarrow$ satisfiability.

Proof that $3\text{-}SAT \leq_P VC$ (2)

- Proof that satisfiability $\Rightarrow$ existence of the cover:
  - Select the vertex of a Type 1 edge that will be in the cover according to the truth assignment.
    - That is, we select the vertex corresponding to u_i if the satisfying assignment sets u_i true otherwise we select the "not u_i" vertex.
  - Now, consider any Type 3 "group" of edges corresponding to a particular clause (triangle).
    - A vertex selected in the previous point will cover at least one of the three edges (because we have satisfiability).
  - We can include in the cover the 2 endpoints in a triangle that are on the other two edges of the group (which may or may not be covered by vertices already covering a Type 1 edge).
  - So: we have derived a cover with $m + 2n$ vertices.

Proof that 3-$SAT \leq_P VC$ (3)

– Proof that existence of a cover $\Rightarrow$ satisfiability:

- Since each Type 1 edge contains exactly one covering vertex we can use it to specify the satisfiability assignment:
  – If the vertex corresponding to u_i is in the cover then we have the assignment $u_i = true$ otherwise $u_i = false$.
- We now wish to demonstrate that this assignment will give a true value to each of the clauses in the 3-SAT formula.
- Consider three edges in a Type 3 “group” corresponding to a clause (triangle).

Proof that 3-$SAT \leq_P VC$ (4)

- Consider three edges in a Type 3 “group” corresponding to a clause (triangle).
- Only two of these edges can be covered by the two vertices that are covering the edges in a triangle so that means that one of the edges is covered by a vertex on a Type 1 edge.
  – Note that this is the vertex leading to our specified assignment described in the first point.
- But that means that it has an entry in the clause with a true value and so it is satisfied.
- Since this is true for all groups (i.e. all clauses) we have satisfiability.

Some Cautionary Notes

- Be sure to understand the following:

- We have just shown that the availability of an algorithm for VC allows us to do $3\text{-}SAT$.
  Nothing more is intended!
- This was expressed as: $3\text{-}SAT \leq_P VC$.
  (read: $3\text{-}SAT$ reduces to VC).
  - The proof involved both "if and only if" parts but this was to ensure that the "yes" decision for a VC instance maps to "yes" decision for the given $3\text{-}SAT$ and a "no" from VC to a "no" in $3\text{-}SAT$.

Some Cautionary Notes

- It did *not* give us any proof of $VC \leq_P 3\text{-}SAT$.
  - This requires its own proof.
    - After all: we showed that the existence of a vertex cover of a *particular* graph G can be used to solve a given $3\text{-}SAT$ problem.
      - However, it was possible to be given **any** general 3-SAT problem.
  - We would expect that a proof of $VC \leq_P 3\text{-}SAT$ to involve the VC of an arbitrary graph G (with the possible use of an assignment satisfying some particular $3\text{-}SAT$ expression).
    - In other words, we have shown that some particular graphs correspond to $3\text{-}SAT$ problems.
      - There might be other graphs that do not correspond to any $3\text{-}SAT$ problems.

NP Hard

- Definition:

  A problem Q is NP-hard if and only if for <u>any</u> problem R in NP we have $R \leq_P Q$.

- Recall that this means that Q is at least as hard as R.

NP-Complete (1)

- Definition:

  If a problem T is NP-hard (i.e. $R \leq_P T$ for all R in NP) and if T is in NP then T is NP-complete.

- NP-complete problems are the “hardest” problems in NP.

NP-Complete (2)

- Consider two problems A and B with $A \leq_P B$. Recall that if we can solve B in polynomial time then we can solve A in polynomial time.
  – So: if someone was to solve an NP-complete problem in polynomial time, then we have $P = NP$.
- If we prove a problem is NP-complete, we typically give up on a polynomial time solution.

Satisfiability

- Definition:
  – Given a set of Boolean variables $U = (u_1, u_2, \ldots, u_m)$ and a logical formula f we seek to find an assignment of the variables that will give f a true value.
  – The formula for f is Boolean (ANDs, ORs, NOTs) without any further restriction (3-SAT is a special case logical function).

Cook's Theorem

Theorem: *SAT* is *NP*-complete.

Proof (sketch):

- First claim: $SAT \in NP$
  - The following non-deterministic algorithm solves *SAT* in polynomial time:

```
for i := 1 to m do
   try u[i] = 0, 1   //1 = true, 0 = false
evaluate f with assignment
           (u1 := u[1], u2 := u[2],…,un := u[n]);
if f is satisfied then ACCEPT
                  else REJECT;
```

- Second claim: SAT is *NP*-hard
  - Consider any problem $Q \in NP$.
  - There is a polynomial time non-deterministic algorithm A solving Q.
    - Consider a computer with registers $(R_1, R_2, \ldots,)$ each containing a number of fixed size.
    - Our program has a constant number of lines.
    - For simplicity each line does one of the following:

      Perform a basic arithmetic operation

      (e.g.: $R_w := R_u + R_v$, $R_w := R_u * R_v$, etc. including some operations involving indirect addressing)

      IF R_w = 0 THEN GOTO m

      GOTO j

      ACCEPT

      REJECT

      TRY R_w BETWEEN 0 AND 1

– At the beginning of execution, the input is stored in the first n registers.
– Program running time is bounded by a polynomial $p(n)$.
– During execution, the program only uses a number of registers bounded by a polynomial $q(n)$.
– The program itself does not change.
– Consider a formula with the following variables:
  • $L[i, k]$ – at time i the program is on line k.
  • $V[i, j, k]$ – at time i register j has value k.
– We will construct a large SAT formula that "simulates" our program.
  • Our formula will be a big conjunction of the following formulas:

1. At each time i, the program is on exactly one line:
   $\neg\left(L[i,k]\wedge L\left[i,k^*\right]\right)$ for all combinations $k \neq k^*$.
2. At each time i, each register contains a single value:
   $\neg\left(V[i,j,k]\wedge V\left[i,j,k^*\right]\right)$ for all combinations j and $k \neq k^*$.
3. At time 0, the program is on line 1, and the first n registers hold values $(x_1, x_2, \ldots, x_n)$.
   (Input and others are 0).
   $L[0,1]\wedge V[0,1,x_1]\wedge V[0,2,x_2]\wedge\ldots\wedge V[0,n,x_n]\wedge\ldots\wedge V\left[0,q(n),0\right]$.
4. After $p(n)$ time steps, the program has entered the line with the "*ACCEPT*" command:
   $L[p(n), k]$ where k is one of the lines containing "ACCEPT".

5. For each time $0 \leq i < p(n)$, the state of the computer changes between time i and $i+1$ according to the program:
   - If line k contains "ACCEPT" or "REJECT"

     $$L[i,k] \Rightarrow L[i+1,k].$$

   - If line k contains "GOTO m"

     $$L[i,k] \Rightarrow L[i+1,m].$$

   - If line k contains "IF R_w = 0 THEN GOTO m"

     $$L[i,k] \wedge V[i,w,0] \Rightarrow L[i+1,m]$$

     $$L[i,k] \wedge \neg V[i,w,0] \Rightarrow L[i+1,k+1].$$

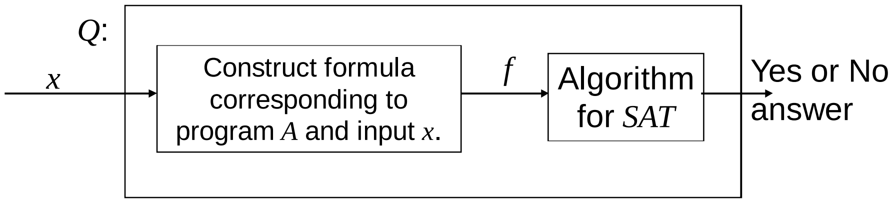
   - If line k contains "TRY R_w = 0, 1"

     $$L[i,k] \Rightarrow L[i+1,k+1] \wedge \left(V[i+1,w,0] \vee V[i+1,w,1]\right).$$

     … and so on for the rest of the instructions in the instruction set.

– This yields a large Boolean formula f that is:
- Of polynomial size and can be constructed in polynomial time.
- It is satisfiable if and only if the original program has an accepting computation for a given input.

– So, we have the following reduction demonstrating that $Q \leq_P SAT$:



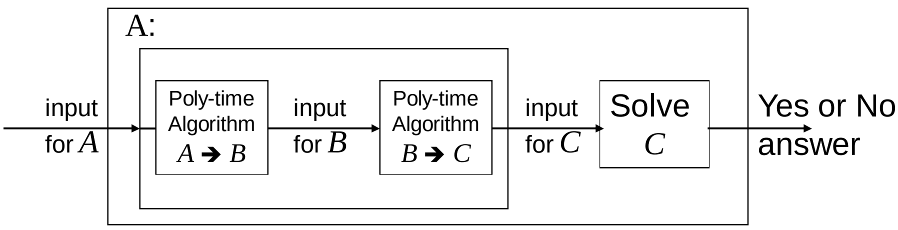
As required we showed that for any $Q \in NP$, $Q \leq_P SAT$.

Proving Other Problems to be *NP*-Complete (1)

- We proved *SAT* is *NP*-Complete by reducing any problem $Q \in NP$ to *SAT* $(Q \leq_P SAT)$.
- Thus if *SAT* can be solved in polynomial time then <u>any</u> problem in *NP* can be solved in polynomial time.

Proving Other Problems to be *NP*-Complete (2)

- Lemma: "Reduces to" ($\leq_P$) is a transitive relation.
  - If $A \leq_P B$, and $B \leq_P C$, then $A \leq_P C$.
  - "Picture" proof:



- Recall definition of problem Q being NP-complete:
  1. Q is NP-hard (i.e.: $R \leq_P Q$ for all $R \in NP$) and
  2. $Q \in NP$.

  – This definition has R represent <u>any</u> problem in NP.
    – We can relax this requirement and work with a single problem from NP, as follows:

  – Corollary:
  If N is NP-complete and $N \leq_P Q$, then Q is NP-hard.
    – $R \in \mathrm{NP} \Rightarrow R \leq_P N$ and $N \leq_P Q$ so transitivity gives $R \leq_P Q$.

Polynomial Time Verification

- Certificates:
  – Suppose you have a list of vertices that specify the order of nodes in the solution of a Hamiltonian Circuit (HAM) problem.
    - Recall that this would guarantee a "YES" answer for our HAM decision problem.
    - The character string that specifies this list of vertices is called a certificate.
    - The verification (i.e. correctness) of the certificate can be accomplished in polynomial time.
      – (linear time in this case).
    - More formally:

Verification Algorithms

- Definition:
  - A verification algorithm for a problem Q is a two argument algorithm $A(x, y)$ where
    - x is an instance of a problem Q (let $|x| = n$).
    - y is a string (the certificate) with a size that is bounded by a polynomial $q(n)$.
    - The running time of A is polynomial in n.
    - If x is a "NO" instance of the problem Q, $A(x, y)$ = "NO" for any certificate.
    - If x is a "YES" instance of the problem Q, then there exists a certificate y such that $A(x, y)$ = "YES".
      - Given y, $A(x,y)$ should verify that certificate y does indeed produce a "YES" in polynomial time.

Certificates, Verification, and NP

- Lemma:
  - Given a problem Q,
    $Q \in NP \Leftrightarrow$ there is a verification algorithm for Q.
  - Proof sketch:
    - ($\Rightarrow$): Take all "try" command choices and make them into a certificate.
    - ($\Leftarrow$): Non-deterministically generate a certificate and run the verification algorithm

Proving Q is NP-complete

- To prove that a problem Q is NP complete:
  1. Choose a problem N known to be NP-complete
     - for example, SAT.
  2. Show that $N \leq_P Q$:
     - Give a polynomial time algorithm transforming any instance x of N to an instance $f(x)$ of Q.
     - Show: If x is a "YES" instance of N then $f(x)$ is a "YES" instance of Q.
     - Show: If x is a "NO" instance of N then $f(x)$ is a "NO" instance of Q or if $f(x)$ is a "YES" instance of Q then x is a "YES" instance of N.
  3. Conclude that since N is NP-complete, Q must also be NP-Hard.

But we need more…

4. To finish the NP-completeness proof, we must also show that $Q \in NP$.
   - Go with either of the following two approaches:
   1. Design a polynomial time non-deterministic algorithm that solves Q.

      OR:

   2. Define a polynomial size certificate and give a polynomial time verification algorithm.

   - Before proceeding with further NP-completeness proofs, we need an "arsenal" of basic NP-complete problems:

Seven Basic NP-Complete Problems (1)

- SAT:
  - Instance: Boolean formula f
  - Problem: Can f be satisfied?
- 3-SAT:
  - Instance: Boolean formula f in conjunctive normal form with each "or" clause containing at most 3 variables.
  - Problem: Can f be satisfied?

Seven Basic NP-Complete Problems (2)

- VC:
  - Instance: Graph $G = (V, E)$ and a number K.
  - Problem: Is there a set of vertices V_c with set size $\leq K$ such that for any edge $e = (u, v)$, either $u \in V_c$ or $v \in V_c$?
- HAM:
  - Instance: Graph $G = (V, E)$.
  - Problem: Is there a Hamiltonian cycle in G (a tour containing all the vertices)?

Seven Basic *NP*-Complete Problems (3)

- *TSP-D*:
  - Instance: Weighted graph $G = (V, E)$ and a number K.
  - Problem: Is there a Hamiltonian cycle in G with total weight $\leq K$?
- *CLIQUE*:
  - Instance: Graph $G = (V, E)$ and a number K.
  - Problem: Does G contain a complete subgraph with $\geq K$ vertices?
- *SUBSET-SUM*:
  - Instance: n numbers $s_1, s_2, \ldots, s_n$ and a target number t.
  - Problem: Is there a subset of the numbers $\{s_1, s_2, \ldots, s_n\}$ with sum t?

SUBSET-SUM is *NP*-Complete

- *SUBSET-SUM* problem:
  - Instance: We are given a set S of positive integers, and a target integer t.
  - Question: Does there exist a subset of S adding up to t?
    - Example: $\{1, 3, 5, 17, 42, 391\}$, target 50.
    - *Subset-Sum* is a good problem to use when proving *NP*-completeness for problems defined on sets of integers.
  - We will show that $3\text{-}SAT \leq_P SUBSET\text{-}SUM$.
    - So: We are given an arbitrary $3\text{-}SAT$ formula and we wish to derive a set S of integers and a target integer t.
    - Then prove that $3\text{-}SAT$ is satisfiable iff a subset of S adds up to t.

3-*SAT* $\leq_P$ *SUBSET-SUM*

- Idea: we use "bit fields" to encode the 3-*SAT* formula.
  - The integers in S will be derived from these fields.
  - As before, we assume that there are m Boolean variables and n clauses.
  - Our description uses bit fields that represent different aspects of the 3-*SAT* formula: it has two lines for each logic variable:
    - One line specifies which clauses use the true version of a variable.
    - Another line describes which clauses use the false version of the variable.

  - For every variable u_i we create a line T_i corresponding to the true value of u_i and another line F_i for the false value of u_i as follows:

	u_1	u_2	...	u_i	...	u_m	c_1	c_2	...	c_n
...	...	...	...	...	...	...	...	...	...	...
T_i	0	0	0	1	0	0	.	.	.	.
F_i	0	0	0	1	0	0	.	.	.	.
...	...	...	...	...	...	...	...	...	...	...

There is a 1 under c_j in row T_i iff the j^{th} clause contains u_i. (Similarly for the F_i row).

– Example description for:

$$(u_1 \vee \neg u_3 \vee \neg u_4) \wedge (\neg u_1 \vee u_2 \vee \neg u_4)$$

	u_1	u_2	u_3	u_4	c_1	c_2
T_1	1	0	0	0	1	0
F_1	1	0	0	0	0	1
T_2	0	1	0	0	0	1
F_2	0	1	0	0	0	0
T_3	0	0	1	0	0	0
F_3	0	0	1	0	1	0
T_4	0	0	0	1	0	0
F_4	0	0	0	1	1	1

– The numbers for S are built by considering the rows as integers.

– When adding the integers carry poses a problem, so we make the fields a few bits wider

	u_1	u_2	u_3	u_4	c_1	c_2
T_1	001	000	000	000	001	000
F_1	001	000	000	000	000	001
T_2	000	001	000	000	000	001
F_2	000	001	000	000	000	000
T_3	000	000	001	000	000	000
F_3	000	000	001	000	001	000
T_4	000	000	000	001	000	000
F_4	000	000	000	001	001	001

– We wish to have a target t that forces a selection of T_i or F_i rows (but not both) for each value of i such that it satisfies the formula.

	u_1	u_2	u_3	u_4	c_1	c_2
T_1	001	000	000	000	001	000
F_1	001	000	000	000	000	001
T_2	000	001	000	000	000	001
F_2	000	001	000	000	000	000
T_3	000	000	001	000	000	000
F_3	000	000	001	000	001	000
T_4	000	000	000	001	000	000
F_4	000	000	000	001	001	001
t	001	001	001	001		

– What about the sums of entries in the columns c_i for clauses?

- Note that the sum in a c_i column could be 0, 1, 2, or 3, depending on the number of true literals in the clause.
- Hence the clause is true if the sum = 1, 2, or 3.
- Problem: For any column, the target sum must be one specific value, not one out of three possible values.
- Idea: for every clause column, we add two “slack” integers: one with a 1 and another with a 10_2 in that column and 0 everywhere else.
  – Later, we will show that this does not destroy our ability to do an appropriate selection from the T_i, F_i rows.
- Make the target have a 100_2 in the digits corresponding to the clause columns.
  – Note: For any column, the nonzero digits in all integers add up to at most 110_2 (so there’s never a carry-over).

– Going back to our example:

$$(u_1 \vee \neg u_3 \vee \neg u_4) \wedge (\neg u_1 \vee u_2 \vee \neg u_4)$$

	u_1	u_2	u_3	u_4	c_1	c_2
T_1	001	000	000	000	001	000
F_1	001	000	000	000	000	001
T_2	000	001	000	000	000	001
F_2	000	001	000	000	000	000
T_3	000	000	001	000	000	000
F_3	000	000	001	000	001	000
T_4	000	000	000	001	000	000
F_4	000	000	000	001	001	001
$S1_1$	000	000	000	000	001	000
$S2_1$	000	000	000	000	010	000
$S1_2$	000	000	000	000	000	001
$S2_2$	000	000	000	000	000	010
T	001	001	001	001	100	100

– Going back to our example:

$$(u_1 \vee \neg u_3 \vee \neg u_4) \wedge (\neg u_1 \vee u_2 \vee \neg u_4)$$

	u_1	u_2	u_3	u_4	c_1	c_2	
T_1	001	000	000	000	001	000	32776
F_1	001	000	000	000	000	001	32769
T_2	000	001	000	000	000	001	4097
F_2	000	001	000	000	000	000	4096
T_3	000	000	001	000	000	000	512
F_3	000	000	001	000	001	000	520
T_4	000	000	000	001	000	000	64
F_4	000	000	000	001	001	001	73
$S1_{C1}$	000	000	000	000	001	000	8
$S2_{C1}$	000	000	000	000	010	000	16
$S1_{C2}$	000	000	000	000	000	001	1
$S2_{C2}$	000	000	000	000	000	010	2
T	001	001	001	001	100	100	37476

- Suppose the formula is satisfiable:
  - We need to show that there is a subset of S with target sum exactly t.
  - Choose the one integer from the T_i, F_i rows corresponding to true literals, giving sums of exactly 001 in the literal-digit positions.
  - Using only the T_i, F_i rows, we get sums that are 1, 2, or 3 in the clause columns.
  - Choose appropriate "slack integers" (from the $S1_i$ and $S2_i$ rows to make those sums equal to 100_2).
  - The final sum matches the target in all digits, as required.

- Suppose a set of integers sum to target t:
  - We need to show that we can get a satisfying assignment of the given 3-SAT formula.
  - There must be exactly one integer (i.e. row) selected from each pair of the T_i, F_i rows, as there is no other way to get a 1 in the initial columns .
  - The selection thus defines the obvious assignment to variables, but is it a satisfying assignment?
  - For each clause, the "slack integers" in the chosen set can only add up to at most 3 in that clause column.
  - There must be at least one 1 contributed from some integer corresponding to the T_i, F_i rows.
  - That corresponds to a true literal in that clause and the formula is satisfied.

- Finishing our Proof that SUBSET-SUM is NP-Complete:
  - Since 3-*SAT* is *NP*-complete, we have just demonstrated that *SUBSET-SUM* is *NP*-hard.
  - But is it in *NP*? Yes, because:
    - We have a certificate: the subset achieving the target sum.
    - A verification algorithm would verify that the numbers specify a subset and furthermore that they add up to the target.
  - Finally:
    - *SUBSET-SUM* is *NP*-hard + *SUBSET-SUM* in *NP* $\Rightarrow$ *SUBSET-SUM* is *NP*-complete.

The Knapsack Problem

- Problem specification:
  - We are given n objects and a knapsack.
  - Each object i has a positive weight w_i and a positive value v_i.
  - The knapsack can carry a weight of at most W.
  - Fill the knapsack so that the value of objects in the knapsack is maximized.

KNAPSACK

- In the dynamic programming section we studied a solution for knapsack that takes time $O(nW)$ where n is the number of items and W is the capacity of the knapsack.
  - Input is of size $n + \log W$.
  - Time is $O(nW)$.
  - W is exponentially larger than $\log W$.
  - Hence DP solution is reasonable only when W is small.

KNAPSACK is *NP*-complete

- Problem specification:
  - We are given n objects and a knapsack.
  - Each object i has a positive weight w_i and a positive value v_i.
  - The knapsack can carry a weight of at most W.
  - Fill the knapsack so that the value of objects in the knapsack at least C.

$SUBSET\text{-}SUM \leq_P KNAPSACK$ (1)

- Given a $SUBSET\text{-}SUM$ problem, that is a set $S = \{s_1, s_2, \ldots, s_n\}$ and a target T we create a knapsack problem as follows:
  - There are n items.
  - Each object i has a value v_i equal to its weight w_i equal to s_i.
  - The knapsack can carry a weight of at most $W = T$.
  - Fill the knapsack so that the value of objects in the knapsack at least $C = T$.

$SUBSET\text{-}SUM \leq_P KNAPSACK$ (2)

Solution to knapsack ⇒ solution to subset-sum

Proof:

- Since the value equals the weight, a value of at least C means a weight of at least C.
- On the other hand, the capacity of the knapsack is at most C.
- Hence any valid solution to knapsack has value exactly C and is a solution to the subset-sum problem.

$SUBSET\text{-}SUM \leq_P KNAPSACK$ (3)

Solution to subset-sum $\Rightarrow$ solution to knapsack

Proof:

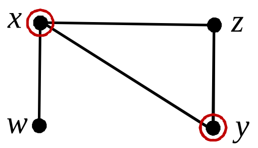
- If the subset-sum problem has a solution, then clearly the corresponding knapsack has value C as $v_i = s_i$ and $$\sum_i s_i = C.$$
- The solution also fits in the knapsack since $w_i = s_i$ and $\sum_i s_i = C.$

$KNAPSACK$ is in NP

- $KNAPSACK$ is in NP since it has a certificate, namely the specific items adding to weight no larger than W and value larger than C, which can easily be checked.
- This completes the proof that $KNAPSACK$ is NP-complete.

A *VC* to *HAM* Reduction (1)

- $VC \leq_p HAM$ Our last reduction! ☹
  - But, more to come (on the exam). ☺
- Given an instance of *VC* (*VERTEX_COVER*): $G = (V, E)$ and $k > 0$, we construct a graph H such that H has a Hamiltonian circuit iff G has a vertex cover of size k.
  - Consider a small example for G:

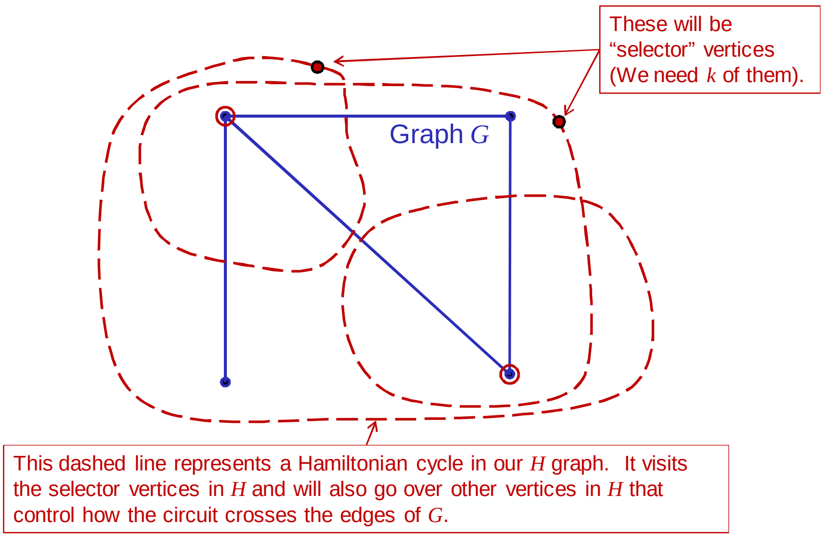


We need a graph H such that *HAM* circuits somehow identify the cover x and y within G.

- If the circuits in H use the edges of G it is not clear how we proceed... (we give up on this thinking).

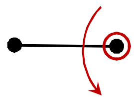
A *VC* to *HAM* Reduction (2)

- **Idea:** Suppose that H is a separate graph that overlays G and the circuit in H "goes around" each of the vertices in the cover.

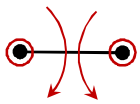


A *VC* to *HAM* Reduction (3)

- Repeating: Suppose that H is a separate graph that overlays G and the circuit in H "goes around" each of the vertices in the cover.
  - We need the circuit to go over an edge of G **once** if only one of its vertices are in the cover:



    Otherwise, the circuit goes over an edge **twice** if both vertices are in the cover:

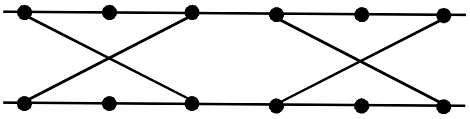


A *VC* to *HAM* Reduction (4)

- Since the Hamiltonian circuit will have to choose between these two scenarios in the designation of a cover, both possibilities must be available for each edge.
  - A clever approach is to use a "gadget".
    - We have already seen gadgets when doing the *3-SAT* to *VC* reduction. This reduction employed a "variable gadget" and a "clause gadget".
  - For $VC \leq_P HAM$ we need an "edge gadget".

A *VC* to *HAM* Reduction (5)

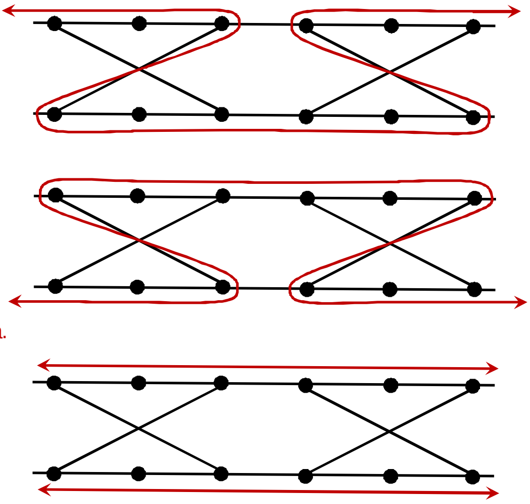
- Each edge of *G* will be associated with an edge gadget in *H*:



  - Our edge gadgets will be part of the graph *H* that will be our instance of the *HAM* problem.
- The requirement that **all** vertices must be part of a Hamiltonian cycle means that *there are only a few ways* for a path to go through the gadget (as illustrated on the next slide):

A *VC* to *HAM* Reduction (6)

- From previous slide: The requirement that **all** vertices must be part of a cycle means that there are only a few ways for a path to go through the gadget:



- Other attempts to go through the vertices will cause some vertices to be left out or will cause vertices to be visited twice!
  - You should try a few of these to get the idea.

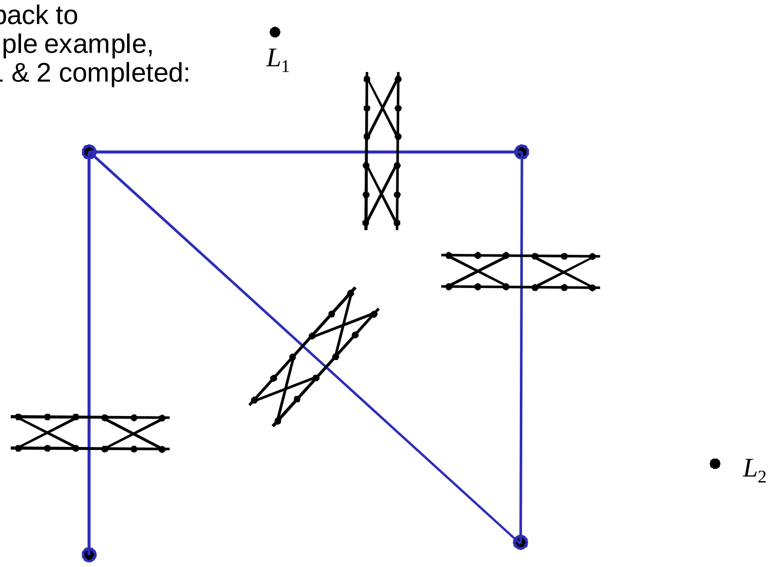
A *VC* to *HAM* Reduction (7)

- So, given an instance of *VC* : $G = (V, E)$ and $k > 0$, how do we construct a graph H such that H has a Hamiltonian circuit iff G has a vertex cover of size k?
- Follow these steps to construct H:
  1. Start with k “selector” vertices labeled $L_1, L_2, \ldots, L_k$.
  2. Put an edge gadget across each edge of G.
     (G is not part of construction. Think of it as being in the background with H overlaying it.)
  3. Chain together all the edges of the gadgets “closest” to a vertex to form a partial loop.
  4. Each end of a partial loop is connected to each L_i selector vertex.

  Note that all of this can be done in time $O(|E|)$.

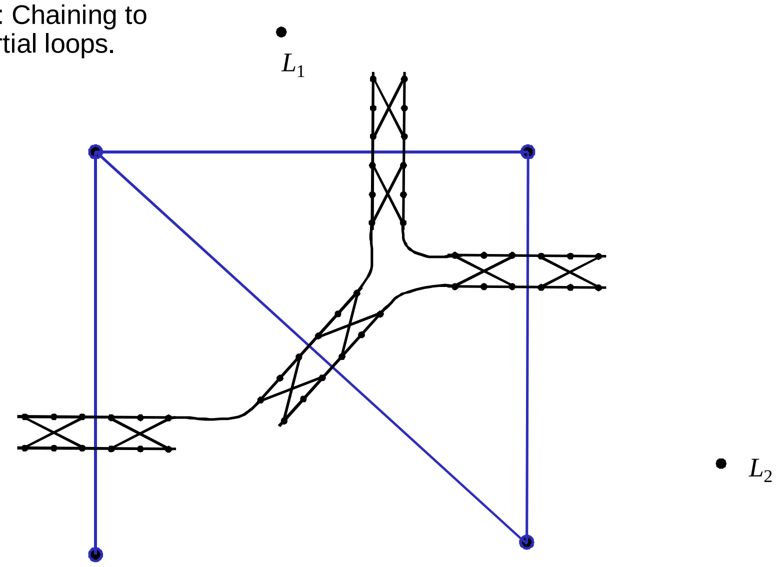
A *VC* to *HAM* Reduction (8)

- Going back to our simple example, Steps 1 & 2 completed:



A *VC* to *HAM* Reduction (9)

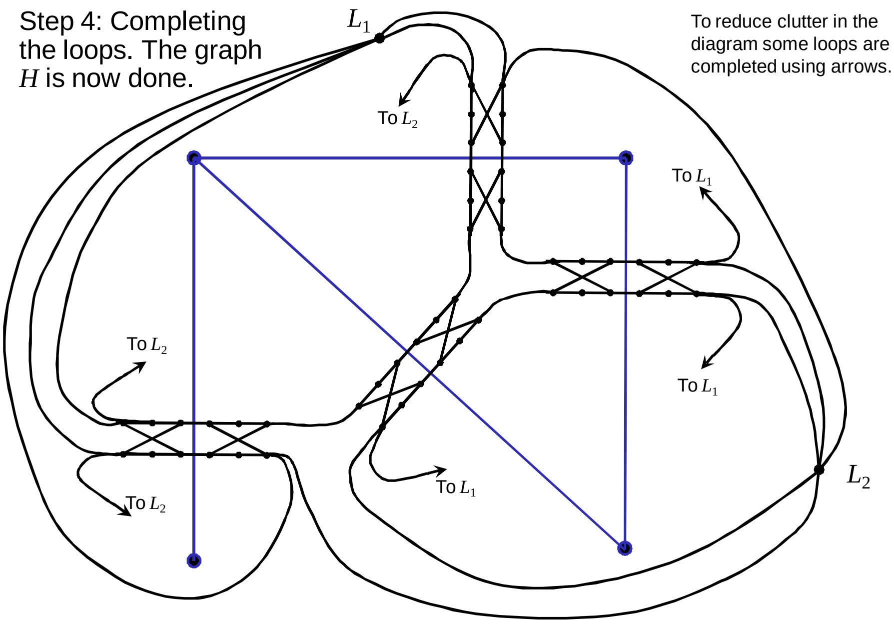
- Step 3: Chaining to get partial loops.



A *VC* to *HAM* Reduction (10)

- Step 4: Completing the loops. The graph H is now done.

To reduce clutter in the diagram some loops are completed using arrows.



A *VC* to *HAM* Reduction (11)

- Assume we have a size k vertex cover for G.
  Then there exists a Hamiltonian circuit in graph H:
  - We start at L_1 and go through a loop that surrounds a vertex of the cover, say u in G.
    - When going through a gadget that overlays edge (u, v) in G, do all 12 vertices if only one of u or v is in the cover.
    - Otherwise, go through 6 of the 12 vertices in the gadget.
  - After passing through all the gadgets overlaying the edges coming out of u, we go to L_2.
  - Repeat with another vertex in the cover until all the cover vertices are circled and then go to L_1 thus getting back to where we started.
    - Note that all vertices are visited once and only once so we have a Hamiltonian circuit.

A *VC* to *HAM* Reduction (12)

- Assume there is a Hamiltonian circuit in H.
  Then there exists a vertex cover for G with k vertices:
  - If a Hamiltonian circuit exists, it will have to go through all of the L_i for $i = 1, 2, \ldots, k$.
  - The circuit *must* alternate between selector vertices and vertices in the gadgets.
  - When going through the gadget vertices it will be going around a G vertex which will be in the cover.
    - We can check to see which vertices are being circled by the partial loops and these will specify the vertex cover.
  - Some supporting observations:
    - Every (u, v) edge in G contains a gadget so all edges will be taken into account by the k partial loops of the Hamiltonian circuit.
    - At least one of the vertices u or v of edge (u, v) will be circled.
      So we do get a vertex cover.

A *VC* to *HAM* Reduction (13)

- The Hamiltonian circuit corresponding to the vertex cover: