Data Structures and Algorithms
CS245-2017S-23
NP-Completeness and Undecidability

David Galles

Department of Computer Science
University of San Francisco
Some algorithms take exponential time
- Simple version of Fibonacci
- Faster versions of Fibonacci that take linear time

Some *Problems* take exponential time
- *All* algorithms that solve the problem take exponential time
- Towers of Hanoi
23-1: Towers of Hanoi

- Move one disk at a time
- Never place a larger disk on a smaller disk
23-2: Towers of Hanoi

- Move one disk at a time
- Never place a larger disk on a smaller disk

Moves = 1
23-3: **Towers of Hanoi**

- Move one disk at a time
- Never place a larger disk on a smaller disk

Moves = 2
23-4: **Towers of Hanoi**

- Move one disk at a time
- Never place a larger disk on a smaller disk

Moves = 3
Towers of Hanoi

- Move one disk at a time
- Never place a larger disk on a smaller disk

Moves = 4
23-6: Towers of Hanoi

- Move one disk at a time
- Never place a larger disk on a smaller disk

Moves = 5
Towers of Hanoi

- Move one disk at a time
- Never place a larger disk on a smaller disk

Moves = 6
Towers of Hanoi

- Move one disk at a time
- Never place a larger disk on a smaller disk

Moves = 7
23-9: **Towers of Hanoi**

- Move one disk at a time
- Never place a larger disk on a smaller disk

Moves = 8
23-10: **Towers of Hanoi**

- Move one disk at a time
- Never place a larger disk on a smaller disk

Moves = 9
Towers of Hanoi

- Move one disk at a time
- Never place a larger disk on a smaller disk

Moves = 10
23-12: Towers of Hanoi

- Move one disk at a time
- Never place a larger disk on a smaller disk

Moves = 11
Towers of Hanoi

- Move one disk at a time
- Never place a larger disk on a smaller disk

Moves = 12
23-14: **Towers of Hanoi**

- Move one disk at a time
- Never place a larger disk on a smaller disk

Moves = 13
23-15: Towers of Hanoi

- Move one disk at a time
- Never place a larger disk on a smaller disk

Moves = 14
23-16: Towers of Hanoi

- Move one disk at a time
- Never place a larger disk on a smaller disk

Moves = 15
Towers of Hanoi

- Move one disk at a time
- Never place a larger disk on a smaller disk

Moves = 15

- Moving $n$ disks requires $2^n - 1$ moves
Towers of Hanoi

- Move one disk at a time
- Never place a larger disk on a smaller disk

Moves = 15

- Moving $n$ disks requires $2^n - 1$ moves
- Completely impractical for large values of $n$
Reductions

A reduction from Problem 1 to Problem 2 allows us to solve Problem 1 in terms of Problem 2

- Given an instance of Problem 1, create an instance of Problem 2
- Solve the instance of Problem 2
- Use the solution of Problem 2 to create a solution to Problem 1
• Example Problem: Pairing
  • Given two lists of integers of size \( n \)
  • Match the smallest element of each list together
  • Match the second smallest element of each list together
  • .. etc.
### 23-21: Reductions

<table>
<thead>
<tr>
<th>List 1</th>
<th>List 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>30</td>
</tr>
<tr>
<td>21</td>
<td>14</td>
</tr>
<tr>
<td>5</td>
<td>26</td>
</tr>
<tr>
<td>47</td>
<td>19</td>
</tr>
<tr>
<td>93</td>
<td>87</td>
</tr>
<tr>
<td>25</td>
<td>54</td>
</tr>
<tr>
<td>14</td>
<td>23</td>
</tr>
<tr>
<td>6</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>8</td>
</tr>
</tbody>
</table>
23-22: Reductions

List 1

13 → 30
21 → 14
5 → 26
47 → 19
93 → 87
25 → 54
14 → 23
6 → 11
12

List 2

8
Reductions

- Reduction from Pairing to Sorting
  - Can we reduce the pairing problem to a sorting problem
  - That is, how can we use the sorting problem to solve the pairing problem?
Reductions

- Reduction from Pairing to Sorting
  - Lets us solve the Pairing problem by solving Sorting problem
  - Given any two lists L1 and L2 that we wish to pair:
    - Sort L1 and L2
    - Pair L1[i] with L2[i] for all i
Reductions

- Reduction from Pairing to Sorting
  - Reduction takes very little time
  - Time to solve Pairing (using this reduction) is the time to solve Sorting
  - We can solve Pairing in time $O(n \log n)$ using sorting.
• Reduction from Sorting to Pairing
  • Given an instance of Sorting, create an instance of pairing problem
  • Solve the pairing problem
  • Use the solution of pairing problem to solve the sorting problem
Given an list $L_1$:
- Create a new list $L_2$, such that $L_2[i] = i$
- Solve the paring problem, pairing $L_1$ and $L_2$
- Use counting sort to sort $L_1$, using the paired element from $L_2$ as the key
Given an list L1:

- Create a new list L2, such that L2[i] = i
- Solve the paring problem, pairing L1 and L2
- Use counting sort to sort L1, using the paired element from L2 as the key

How long does this take?
Given an list $L_1$:

- Create a new list $L_2$, such that $L_2[i] = i$
- Solve the paring problem, pairing $L_1$ and $L_2$
- Use counting sort to sort $L_1$, using the paired element from $L_2$ as the key

How long does this take?

- $O(n + \text{time to do pairing})$
We can reduce Sorting to Pairing, such that:

- Time to do Sorting takes $O(n + \text{time to do pairing})$

Sorting takes $\Omega(n \lg n)$ time

Thus, the pairing problem must take at least $\Omega(n \lg n)$ time as well
We can use a Reduction to compare problems. If there is a reduction from problem $A$ to problem $B$ that can be done quickly, Problem $A$ is known to be hard (cannot be solved quickly). Problem $B$ cannot be solved quickly, either.
A problem is NP if a solution can be verified easily
- Given a potential solution to the problem, verify that the solution does solve the problem
- Verification takes polynomial (not exponential!) time
- (Pretty low bar for “easily”)

NP Problems
A problem is NP if a solution can be verified easily

- **Traveling Salesman Problem (TSP)**
  - Given a graph with weighted vertices, and a cost bound $k$
  - Is there a cycle that contains all vertices in the graph, that has a total cost less than $k$?

- Given any potential solution to the TSP, we can easily verify that the solution is correct
A problem is NP if a solution can be verified easily

- Graph Coloring
  - Given a graph and a number of colors $k$
  - Can we color every vertex using no more than $k$ colors, such that all adjacent vertices have different colors?

- Given any potential solution to the Graph Coloring problem, we can easily verify that the solution is correct.
A problem is NP if a solution can be verified easily.

**Satisfiability**
- Given a boolean formula over a set of boolean variables $a_1 \ldots a_n$
  
  $(a_1 || !a_2) && (a_2 || a_5 || !a_1) && \ldots$

- Can we give a truth value to all variables $a_1 \ldots a_n$ so that the value of the formula is true?

Given any potential solution to the Satisfiability problem, we can easily verify that the solution is correct.
NP Problems

• A problem is NP if a solution can be verified easily
  • Sorting
    • Given a list of elements $L$ and an ordering of the elements $\leq$
    • Create a permutation of $L$ such that $L[i] \leq L[i + 1]$
  • Given any potential solution to the Sorting problem, we can easily verify that the solution is correct
NP Problems

- If we can guess an answer, we can verify it quickly.
- NP stands for Non-Deterministic Polynomial
  - Non-Deterministic = we can guess
  - Polynomial = “quickly”
- NP problem: If we could guess an answer, we could verify it in polynomial ($n$, $n^2$, $n^5$ – not exponential) time.
Two Definitions of Non-Deterministic Machines:

- “Oracle” – allows machine to magically make a correct guess
- Massively parallel – simultaneously try to verify all possible solutions
  - Try all permutations of vertices in a graph, see if any form a cycle with cost $< k$
  - Try all colorings of a graph with up to $k$ colors, see if any are legal
  - Try all permutations of a list, see if any are sorted
A problem is NP if a non-deterministic machine can solve it in polynomial time
  - Of course, we have no real non-deterministic machines

A problem is in P (Polynomial), if a deterministic machine can solve it in polynomial time
  - Sorting is in P – can sort a list in polynomial time
  - All problems in P are also in NP
    - Ignore the oracle
An NP problem is “NP-Complete” if there is a reduction from *any* NP problem to that problem.

For example, Traveling Salesman (TSP) is NP-Complete:

- We can reduce *any* NP problem to TSP.
- If we could solve TSP in polynomial time, we could solve *all* NP problems in polynomial time.

Is TSP unique in this way?
There are many NP-Complete problems

- TSP
- Graph Coloring
- Satisfiability
- .. many, many more

If we could solve *any* of these problems quickly, we could solve *all* of them quickly

All known solutions take exponential time
If a problem is NP-Complete, it almost certainly cannot be solved quickly (polynomial time)

- If it could, then all NP problems could be solved quickly
- Many people have tried for many years to find polynomial solutions for NP complete problems, all have failed

- However, no proof that NP-Complete problems require exponential time – open problem
If we could solve any NP-Complete problem quickly (polynomial time), we could solve all NP problems quickly.

If that is the case, then $\text{NP} = \text{P}$.

- $\text{P}$ is set of problems that can be solved by a standard machine in polynomial time.

Most everyone believes that $\text{NP} \neq \text{P}$, and all NP-Complete problems require exponential time on standard computers – not yet been proven.
Why is NP-Completeness important?

- If a problem is NP-Complete, no point in trying to come up with an algorithm to solve it
- What can we do, if we need to solve a problem that is NP-Complete?
What can we do, if we need to solve a problem that is NP-Complete?

- If the problem we need to solve is very small (< 20), an exponential solution might be OK
- We can solve an \textit{approximation} of the problem
  - Color a graph using an non-optimal number of colors
  - Find a Traveling Salesman tour that is not optimal
Impossible Problems

- Some problems are “easy” – require a fairly small amount of time to solve
  - Sorting
- Some problems are “probably hard” – believed to require exponential time to solve
  - TSP, Graph Coloring, etc
- Some problems are “hard” – known to require an exponential amount of time to solve
  - Towers of Hanoi
- Some problems are impossible – cannot be solved
Halting Problem

- Program is running – seems to be taking a long time
- We’d like to know if the program will eventually finish, or if it is in an infinite loop
- Great debugging tool:
  - Takes as input the source code to a program $p$, and an input $i$
  - Determines if $p$ will run forever when run on $i$
23-48: Halting Problem

• Program is running – seems to be taking a long time

• We’d like to know if the program will eventually finish, or if it is in an infinite loop

• Great debugging tool:
  • Takes as input the source code to a program $p$, and an input $i$
  • Determines if $p$ will run forever when run on $i$

• No such tool can exist!
23-49: **Halting Problem**

- We will prove that the halting problem is unsolvable by contradiction
  - Assume that we have a solution to the halting problem
  - Derive a contradiction
  - Our original assumption (that the halting problem has a solution) must be false
boolean halt(char [] program, char [] input) {

    /* code to determine if the program
    halts when run on the input */

    if (program halts on input) 
        return true;
    else 
        return false;

}
boolean selfhalt(char [] program) {
    if (halt(program, program))
        return true;
    else
        return false;
}
23-52: Halting Problem

boolean selfhalt(char [] program) {
    if (halt(program, program))
        return true;
    else
        return false;
}

void contrary(char [] program) {
    if (selfhalt(program)
        while(true); /* infinite loop */
}
boolean selfhalt(char [] program) {
    if (halt(program, program))
        return true;
    else
        return false;
}

void contrary(char [] program) {
    if (selfhalt(program)
        while(true); /* infinite loop */
}

• what happens when we call contrary, passing in its own source code as input?
23-54: Reduction Example

- Hamiltonian Cycle:
  - Given an unweighted, undirected graph $G$, is there a cycle that includes every vertex exactly once?

- Traveling Salesman Problem (TSP)
  - Given a complete, weighed, undirected graph $G$ and a cost bound $k$, is there a cycle that includes every vertex in $G$, with a cost $< k$?
**23-55: Reduction Example**

- If we could solve the Traveling Salesman problem in polynomial time, we could solve the Hamiltonian Cycle problem in polynomial time.
  - Given any graph $G$, we can create a new graph $G'$ and limit $k$, such that there is a Hamiltonian Circuit in $G$ if and only if there is a Traveling Salesman tour in $G'$ with cost less than $k$.
  - Vertices in $G'$ are the same as the vertices in $G$.
  - For each pair of vertices $x_i$ and $x_j$ in $G$, if the edge $(x_i, x_j)$ is in $G$, add the edge $(x_i, x_j)$ to $G'$ with the cost 1. Otherwise, add the edge $(x_i, x_j)$ to $G'$ with the cost 2.
  - Set the limit $k = \#$ of vertices in $G$. 
23-56: Reduction Example

Limit = 4
If we could solve TSP in polynomial time, we could solve Hamiltonian Cycle problem in polynomial time

- Start with an instance of Hamiltonian Cycle
- Create instance of TSP
- Feed instance of TSP into TSP solver
- Use result to find solution to Hamiltonian Cycle
Given any instance of the Hamiltonian Cycle Problem:

- We can (in polynomial time) create an instance of Satisfiability
- That is, given any graph $G$, we can create a boolean formula $f$, such that $f$ is satisfiable if and only if there is a Hamiltonian Cycle in $G$

If we could solve Satisfiability in Polynomial Time, we could solve the Hamiltonian Cycle problem in Polynomial Time
Given a graph $G$ with $n$ vertices, we will create a formula with $n^2$ variables:

- $x_{11}, x_{12}, x_{13}, \ldots x_{1n}$
- $x_{21}, x_{22}, x_{23}, \ldots x_{2n}$
- $\ldots$
- $x_{n1}, x_{n2}, x_{n3}, \ldots x_{nn}$

Design our formula such that $x_{ij}$ will be true if and only if the $i$th element in a Hamiltonian Circuit of $G$ is vertex # $j$. 
For our set of $n^2$ variables $x_{ij}$, we need to write a formula that ensures that:

- For each $i$, there is exactly one $j$ such that $x_{ij} = \text{true}$
- For each $j$, there is exactly one $i$ such that $x_{ij} = \text{true}$
- If $x_{ij}$ and $x_{(i+1)k}$ are both true, then there must be a link from $v_j$ to $v_k$ in the graph $G$
For each $i$, there is exactly one $j$ such that $x_{ij} = \text{true}$

- For each $i$ in $1 \ldots n$, add the rules:
  - $\left( x_{i1} \mid \mid x_{i2} \mid \mid \ldots \mid \mid x_{in} \right)$

This ensures that for each $i$, there is at least one $j$ such that $x_{ij} = \text{true}$

(This adds $n$ clauses to the formula)
For each $i$, there is exactly one $j$ such that $x_{ij} = \text{true}$ for each $i$ in $1 \ldots n$
for each $j$ in $1 \ldots n$
for each $k$ in $1 \ldots n$ $j \neq k$
Add rule $(\neg x_{ij} || \neg x_{ik})$

This ensures that for each $i$, there is at most one $j$ such that $x_{ij} = \text{true}$

(this adds a total of $n^3$ clauses to the formula)
Reduction Example #2

- If $x_{ij}$ and $x_{(i+1)k}$ are both true, then there must be a link from $v_i$ to $v_k$ in the graph $G$

  for each $i$ in $1 \ldots (n - 1)$
  for each $j$ in $1 \ldots n$
  for each $k$ in $1 \ldots n$
  if edge $(v_j, v_k)$ is not in the graph:
  Add rule $(!x_{ij} \lor !x_{(i+1)k})$

- (This adds no more than $n^3$ clauses to the formula)
If $x_{nj}$ and $x_{0k}$ are both true, then there must be a link from $v_i$ to $v_k$ in the graph $G$ (looping back to finish cycle)

for each $j$ in $1 \ldots n$
for each $k$ in $1 \ldots n$
if edge $(v_n, v_0)$ is not in the graph:
Add rule $(!x_{nj} \| !x_{0k})$

(This adds no more than $n^2$ clauses to the formula)
In order for this formula to be satisfied:

- For each $i$, there is exactly one $j$ such that $x_{ij}$ is true
- For each $j$, there is exactly one $i$ such that $x_{ji}$ is true
- If $x_{ij}$ is true, and $x_{(i+1)k}$ is true, then there is an arc from $v_j$ to $v_k$ in the graph $G$

Thus, the formula can only be satisfied if there is a Hamiltonian Cycle of the graph
• Exact Cover Problem
  • Set of elements $A$
  • $F \subseteq 2^A$, family of subsets
  • Is there a subset of $F$ such that each element of $A$ appears exactly once?
• Exact Cover Problem
  • \( A = \{a, b, c, d, e, f, g\} \)
  • \( F = \{\{a, b, c\}, \{d, e, f\}, \{b, f, g\}, \{g\}\} \)
  • Exact cover exists:
    \( \{a, b, c\}, \{d, e, f\}, \{g\} \)
• **Exact Cover Problem**
  
  - $A = \{a, b, c, d, e, f, g\}$
  - $F = \{\{a, b, c\}, \{c, d, e, f\}, \{a, f, g\}, \{c\}\}$
  - No exact cover exists
Exact Cover is \( \mathbf{NP} \)-Complete

- Reduction from Satisfiability
- Given any instance of Satisfiability, create (in polynomial time) an instance of Exact Cover
- Solution to Exact Cover problem tells us solution to Satisfiability problem
- Satisfiability is \( \mathbf{NP} \)-Complete \( \implies \) Exact Cover is \( \mathbf{NP} \)-Complete
Given an instance of SAT:

- \( C_1 = (x_1 \lor \overline{x_2}) \)
- \( C_2 = (\overline{x_1} \lor x_2 \lor x_3) \)
- \( C_3 = (x_2) \)
- \( C_4 = (\overline{x_2} \lor \overline{x_3}) \)

Formula: \( C_1 \land C_2 \land C_3 \land C_4 \)

Create an instance of Exact Cover

- Define a set \( A \) and family of subsets \( F \) such that there is an exact cover of \( A \) in \( F \) if and only if the formula is satisfiable
23-71: Exact Cover is NP-Complete

\[ C_1 = (x_1 \lor \overline{x_2}) \quad C_2 = (\overline{x_1} \lor x_2 \lor x_3) \quad C_3 = (x_2) \quad C_4 = (\overline{x_2} \lor \overline{x_3}) \]

\[ A = \{x_1, x_2, x_3, C_1, C_2, C_3, C_4, p_{11}, p_{12}, p_{21}, p_{22}, p_{23}, p_{31}, p_{41}, p_{42}\} \]

\[ F = \{\{p_{11}\}, \{p_{12}\}, \{p_{21}\}, \{p_{22}\}, \{p_{23}\}, \{p_{31}\}, \{p_{41}\}, \{p_{42}\}\} \]

\[ X_1, f = \{x_1, p_{11}\} \]

\[ X_1, t = \{x_1, p_{21}\} \]

\[ X_2, f = \{x_2, p_{22}, p_{31}\} \]

\[ X_2, t = \{x_2, p_{12}, p_{41}\} \]

\[ X_3, f = \{x_3, p_{23}\} \]

\[ X_3, t = \{x_3, p_{42}\} \]

\[ \{C_1, p_{11}\}, \quad \{C_1, p_{12}\}, \quad \{C_2, p_{21}\}, \quad \{C_2, p_{22}\}, \quad \{C_2, p_{23}\}, \quad \{C_3, p_{31}\}, \]

\[ \{C_4, p_{41}\}, \quad \{C_4, p_{42}\} \} \]
Directed Hamiltonian Cycle

- Given any directed graph $G$, determine if $G$ has a Hamiltonian Cycle
- Cycle that includes every node in the graph exactly once, following the direction of the arrows
Given any directed graph $G$, determine if $G$ has a Hamiltonian Cycle.

Cycle that includes every node in the graph exactly once, following the direction of the arrows.
The Directed Hamiltonian Cycle problem is \textit{NP}-Complete

Reduce Exact Cover to Directed Hamiltonian Cycle

Given any set $A$, and family of subsets $F$:

Create a graph $G$ that has a hamiltonian cycle if and only if there is an exact cover of $A$ in $F$
• Widgets:
  • Consider the following graph segment:

  ![Graph Diagram]

  • If a graph containing this subgraph has a Hamiltonian cycle, then the cycle must contain either $a \rightarrow u \rightarrow v \rightarrow w \rightarrow b$ or $c \rightarrow w \rightarrow v \rightarrow u \rightarrow d$ — but not both (why)?
• Widgets:
  • XOR edges: Exactly one of the edges must be used in a Hamiltonian Cycle
Directed Hamiltonian Cycle

- Widgets:
  - XOR edges: Exactly one of the edges must be used in a Hamiltonian Cycle
Directed Hamiltonian Cycle

- Add a vertex for every variable in $A$ (+ 1 extra)

$A = \{a_0, a_1, a_2, a_3\}$

$F_1 = \{a_1, a_2\}$
$F_2 = \{a_3\}$
$F_3 = \{a_2, a_3\}$
Directed Hamiltonian Cycle

- Add a vertex for every subset $F$ (+ 1 extra)

$F_0 = \{ a_3 \}$
$F_1 = \{ a_1, a_2 \}$
$F_2 = \{ a_3 \}$
$F_3 = \{ a_2, a_3 \}$
Directed Hamiltonian Cycle

- Add an edge from the last variable to the 0th subset, and from the last subset to the 0th variable

\[
\begin{align*}
F_0 &= \{ a_1, a_2 \} \\
F_1 &= \{ a_3 \} \\
F_2 &= \{ a_2, a_3 \}
\end{align*}
\]
Directed Hamiltonian Cycle

- Add 2 edges from $F_i$ to $F_{i+1}$. One edge will be a “short edge”, and one will be a “long edge”.

$F_1 = \{ a_1, a_2 \}$
$F_2 = \{ a_3 \}$
$F_3 = \{ a_2, a_3 \}$
Directed Hamiltonian Cycle

- Add an edge from \( a_{i-1} \) to \( a_i \) for each subset \( a_i \) appears in.

\[
\begin{align*}
F &= \{a_1, a_2\} \\
F_1 &= \{a_3\} \\
F_2 &= \{a_2, a_3\}
\end{align*}
\]
Directed Hamiltonian Cycle

- Each edge \((a_{i-1}, a_i)\) corresponds to some subset that contains \(a_i\). Add an XOR link between this edge and the long edge of the corresponding subset.

\[
F = \{ a_1, a_2 \}
\]

\[
F_1 = \{ a_1, a_2 \}
\]

\[
F_2 = \{ a_3 \}
\]

\[
F_3 = \{ a_3, a_2, a_3 \}
\]

XOR edge
Directed Hamiltonian Cycle

\[ F = \{ a_1, a_2 \} \]
\[ F_2 = \{ a_3 \} \]
\[ F_3 = \{ a_2, a_3 \} \]

XOR edge
23-85: Directed Hamiltonian Cycle

- **F₀**: A directed graph with vertices labeled a₀ to a₄.
- **F₁**: Set of vertices {a₂, a₄}.
- **F₂**: Set of vertices {a₂, a₄}.
- **F₃**: Set of vertices {a₁, a₃}.
- **F₄**: Set of vertices {a₂}.

The graph shows a Hamiltonian cycle where each vertex is connected to the next in a sequence, and there are XOR edges indicated by circles on the diagram.