\[
\text{Circuit SAT} \leq \text{3-SAT} \leq \text{Subset Sum} \\
\text{Independent Set} \leq \text{Vertex Cover} \leq \text{3-SAT} \\
\text{clique} \leq \text{Directed Hamiltonian Cycle} \leq \text{Undirected Hamiltonian Cycle} \leq \text{TSP} \\
\text{Vertex Cover} \leq \text{Undirected Hamiltonian Cycle} \leq \text{Hamiltonian Path}
\]
Subset Sum

- **Input:** numbers $w_1, w_2, \ldots w_n, W$
- **Question:** is there a subset $S \subseteq \{1, 2, \ldots n\}$ such that $\sum_{i\in S} w_i = W$

- **Theorem.** Subset Sum is NP-complete.
- **Proof.**
  1. Subset Sum is in NP. (done in previous lecture)
  2. $3$-SAT $\leq_p$ Subset Sum

    - Assume we have a polynomial time algorithm for Subset Sum. Make a polynomial time algorithm for $3$-SAT
    - Input: A $3$-SAT formula $\varphi$ with clauses $C_1 \ldots C_m$ on variables $x_1 \ldots x_n$
    - Question: Is $\varphi$ satisfiable?
      1. Construct an instance of subset sum, $(S, W)$, such that

         $S, W$ has a subset $S'$ which sums to $W$ iff $\varphi$ is satisfiable.

      2. Run the subset sum algorithm
      3. Return its answer
3-SAT $\leq_P$ Subset Sum

- **Input:** A 3-SAT formula $\varphi$ with clauses $C_1 \ldots C_m$ on variables $x_1 \ldots x_n$
- **Construct** an instance of Subset Sum $(S, W)$ such that it has a subset $S'$ that sums to $W$ iff $\varphi$ is satisfiable
  - **Assumption:**
    - No clause contains a variable and its negation
      - Otherwise, the clause will be automatically satisfied by any value assigned to the literal
    - Each variable appears in at least one clause
  - **Idea:** Create set $S$ so that
    - It has two numbers for each variable $x_i$ and two numbers for each clause
      - In total: $2n + 2m$ numbers
      - Each number has $n+m$ digits
      - Each digit correspond to a variable or a clause
3-SAT $\leq_P$ Subset Sum

- $\phi = C_1 \land C_2 \land C_3 \land C_4$
- $C_1 = (x_1 \lor \neg x_2 \lor \neg x_3)$
- $C_2 = (\neg x_1 \lor \neg x_2 \lor \neg x_3)$
- $C_3 = (\neg x_1 \lor \neg x_2 \lor x_3)$
- $C_4 = (x_1 \lor x_2 \lor x_3)$
- For each variable $x_i$, set $S$ contains two integers $v_i$ and $v'_i$
  - All $v_i$ and $v'_i$ values are unique
- For each clause, set $S$ contains two integers $s_j$ and $s'_j$
  - In the column $C_j$, $s_j = 1$ $s'_j = 2$, and 0 on the rest of columns

<table>
<thead>
<tr>
<th></th>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$C_1$</th>
<th>$C_2$</th>
<th>$C_3$</th>
<th>$C_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_1$</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$v'_1$</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$v_2$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$v'_2$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$v_3$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$v'_3$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$s_1$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$s'_1$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$s_2$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$s'_2$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$s_3$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$s'_3$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>$s_4$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$s'_4$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>$W$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
3-SAT ≤ₚ Subset Sum

- Each row is a number in set S
- Add two numbers by adding up rows
- ensure we don’t choose row $v_i$ and row $v_i'$$$
- Target sum ≥ 1 to ensure we pick at least one literal in each clause
  - What can the sum be? 1 or 2 or 3.
  - Add slack rows of 1 and 2 so sum can always be 4.
- Each row is a number in S
  - 2n+2m numbers each with n+m digits (base 10)
3-SAT $\leq_p$ Subset Sum

Claim. $\varphi$ is satisfiable iff there is a subset of the numbers with sum $W$.

Proof.

- Suppose $\varphi$ is satisfiable. If $x_i$ is True, pick row $v_i$. If $x_i$ is False, pick row $\neg v_i$.
- Then column $v_i$ adds up to its target 1, and column $C_j$ adds to 1, 2, or 3.
- Next we choose some slack rows $s_j$ or $s'_j$ to increase the sum to 4.
- This gives a set of rows that sum to 4.

Example: $\varphi$: $x_1 = 0$, $x_2 = 0$, $x_3 = 1$
3-SAT ≤ₚ Subset Sum

\( \varphi = C_1 \land C_2 \land C_3 \land C_4 \)

\( C_1 = (x_1 \lor \neg x_2 \lor \neg x_3) \)

\( C_2 = (\neg x_1 \lor \neg x_2 \lor \neg x_3) \)

\( C_3 = (\neg x_1 \lor \neg x_2 \lor x_3) \)

\( C_4 = (x_1 \lor x_2 \lor x_3) \)

\( \varphi: x_1 = 0, x_2 = 0, x_3 = 1 \)

\( S = \{v_1, v'_1, v_2, v'_2, v_3, v'_3, s_1, s'_1, s_2, s'_2, s_3, s'_3, s_4, s'_4\} \)

<table>
<thead>
<tr>
<th></th>
<th>( x_1 )</th>
<th>( x_2 )</th>
<th>( x_3 )</th>
<th>( C_1 )</th>
<th>( C_2 )</th>
<th>( C_3 )</th>
<th>( C_4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( v_1 )</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>( v'_1 )</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( v_2 )</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>( v'_2 )</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( v_3 )</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( v'_3 )</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( s_1 )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( s'_1 )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( s_2 )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( s'_2 )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( s_3 )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>( s'_3 )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>( s_4 )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>( s'_4 )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>( W )</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
3-SAT $\leq_P$ Subset Sum

$\varphi$: $x_1 = 0$, $x_2 = 0$, $x_3 = 1$

$S' = \{v'_1, v'_2, v_3, s_1, s'_1, s'_2, s_3, s_4, s'_4\}$

$S'$ matches the target sum ($W$), and no carries can occur, the values of $S'$ sum to $W$
3-SAT $\leq_P$ Subset Sum

**Claim.** $\varphi$ is satisfiable iff there is a subset of the numbers with sum $W$.

**Proof.**

- Suppose there is a subset $S'$ with sum $W=1114444$
- Because $x_i$ column sum is 1, we must have chosen row $x_i$ or row $\neg x_i$ (not both) — set the variable accordingly.
- Because column $C_j$ sum is 4 and slacks sum to $\leq 3$, we must have chosen a literal to satisfy clause $C_j$. Thus $\varphi$ is satisfiable.
Approximation Algorithms
Approximation Algorithms

- Many practical problems are NP-complete — no one knows a polynomial time algorithm, nor can we prove that none exists.
- What to do?
  - If the input is small, use an algorithm with exponential time
  - Special cases can be solved in polynomial time
  - Efficient exhaustive search (backtracking, branch-and-bound).
    - Exponential time in the worst case, but can be useful.
  - Heuristics - there might be no guarantee on run-time nor on quality of solution.
    - Local search — start with some solution and try to improve it via small “local” changes.
    - Hill climbing, simulated annealing
    - Particle swarm, evolutionary algorithms
  - Approximation algorithms
    - Near optimal solutions
    - Polynomial time and a guarantee on the quality of the solution
    - E.g. for a minimization problem, might guarantee a solution $\leq 2 \cdot \text{min}$
Approximation Algorithms

- The cost of the approximate solution: $C$
- The cost of the optimal solution: $C^*$
- Approximation ratio of an approximation algorithm: $\rho(n)$
  - Maximization problem: $C^* \leq \rho(n) C$
  - Minimization problem: $C \leq \rho(n) C^*$
  - $\rho(n) \geq 1$
- $\rho(n)$-approximation algorithm
  - An algorithm that achieves an approximation ratio of $\rho(n)$
Approximation algorithms for Vertex Cover

- A vertex cover is a set $S \subseteq V$ such that every edge $(u,v) \in E$ has $u$ or $v$ (or both) in $S$.
- Optimization problem: find a minimum size vertex cover
- Recall that the decision version is NP-complete.
Greedy Algorithm 1

C := ∅

repeat

    C := C ∪ {vertex of maximum degree}

    remove covered edges

Until no edges remain

Examples:

• Runtime: polynomial
Greedy Algorithm 2

\[ C := \emptyset \]
\[ F = E \quad // \text{F is uncovered edges} \]
While \( F \neq \emptyset \)
  pick \( e = (u,v) \) from \( F \)
  add \( u \) and \( v \) to \( C \)
  remove \( (u,v) \) from \( F \)
  remove edges incident to \( u \) from \( F \)
  remove edges incident to \( v \) from \( F \)

- Example:
- Runtime: polynomial
- \( C \) is a vertex cover, since the algorithm loops until every edge in \( F \) has been covered by some vertex in \( C \)
Greedy Algorithm 1 v.s. Greedy Algorithm 2

- Which is better, Algorithm 1 or Algorithm 2?
  - On the example provided, Alg. 1 is better.
- Find an example where algorithm 2 is better
Greedy Algorithm 2: Analysis of approximation factor

- Let $C = \text{vertex cover found by Algorithm}$
- Let $C^* = \text{a minimum vertex cover}$
- **Claim:** $C \leq 2 \cdot C^*$
- **Proof:**
  - $A$: set of edges picked on the red line of the algorithm
  - $|C^*| \geq |A|$  
    - No two edges in $A$ share an endpoint, since one edge is picked all other edges incident on its endpoints are deleted from $F$  
    - No two edges in $A$ are covered by the same vertex from $C^* \rightarrow$ for every vertex in $C^*$, there is at most one edge in $A$
  - $|C| = 2|A|$  
    - Each execution of line 4 picks an edge for which neither of its endpoints is already in C
  - $|C| \leq 2|C^*|$
Approximation algorithms for Vertex Cover

- We say that Algorithm 2 has approximation factor 2 because it produces a vertex cover of size $\leq 2 \cdot \text{optimum}$
- Algorithm 1 has approximation factor $\Theta(\log n)$. It is worse than Algorithm 2.
- Recall that Vertex Cover and Independent Set are closely related. However, Independent Set has no good approximation algorithm unless $P = NP$.
  - Covered in CS466
Travelling Salesman Problem

**Input:** a graph G, weights on edges, number k

**Question:** does G have a TSP tour of length ≤ k

Euclidean TSP. For the complete graph on points in the plane, with weight = Euclidean distance.

**Note:** Euclidean TSP is NP-complete

key property of Euclidean case: triangle inequality

\[ w(c, b) \leq w(c, a) + w(a, b) \]
Approximation algorithm for Euclidean TSP

APPROX-TSP-TOUR (G, c)
1. select a vertex \( r \in G.V \) to be a “root” vertex
2. compute a minimum spanning tree \( T \) for \( G \) from root \( r \) using MST-PRIM (G,c,r)
3. let \( H \) be a list of vertices, ordered according to when they are first visited in a preorder tree walk of \( T \)
4. return the hamiltonian cycle \( H \)
Approximation algorithm for Euclidean TSP
Approximation algorithm for Euclidean TSP

- (a) select a to be the root
- (b) compute MST (min. spanning tree)
- (c) take a tour by walking around it (we visit every vertex but maybe more than once)
- (d) take shortcuts to avoid revisiting vertices note: by the triangle inequality, the short-cuts are shorter

(e) is the optimal solution, not found by the algorithm
Approximation factor for Euclidean TSP

- \( H^* \): optimal tour
- \( T \): minimum spanning tree computed in the algorithm
- **Claim.** \( c(H) \leq 2 \ c(H^*) \)
- **Proof.**
  - \( c(T) \leq c(H^*) \)
    - Deleting any edge from the optimal tour generates a spanning tree
  - \( c(W) = 2c(T) \)
    - \( W \) is the full walk (part e)
  - \( W \) is not a TSP tour since it visits some vertices more than once
  - \( H \) is a tour (part c)
  - \( c(H) \leq c(W) \)
    - \( H \) is obtained by deleting vertices from \( W \)

Combining the above three inequalities

\[
c(H) \leq c(W) = 2c(T) \leq 2 \ c(H^*) \quad \rightarrow \quad c(H) \leq 2 \ c(H^*)
\]
The general TSP

If $\text{P} \neq \text{NP}$, then for any constant $\rho \geq 1$, there is no polynomial-time approximation algorithm with approximation ratio $\rho$ for the general traveling salesman problem.