Algorithms

R4 Cheng

May 24, 2025

Def. Introduction of precise, umambiguous, and correct procedures for solving general problems

We care how many clicks in debug mode

Runtime Analysis

Runtime is expressed by counting the number of steps, as funtion of **the size of the input**

Asymptotic Notations:

- Big O: upper bound on the growth rate
- Little o: "strict" upper bound on the growth rate
- Big Ω: lower bound on the growth rate
- Θ: tight bound on the growth rate

$f(n)$: the runtime of an algorithm for input size n

$g(n)$: a simplified function without constants, lower order terms, e.g. n^2, $n \log n$

Big-O Definition: It is said $f(n) = O(g(n)) \iff$ there exists a constant $c > 0$ and $n_0 > 0$ such that

$$f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0$$

Little-o Definition:

$$f(n) = o(g(n)) \iff \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

E.g. $f(n) = 3900n + 57800$

$f(n) = o(n^2)$? T, because $\lim_{n\to\infty} \frac{3900n+57800}{n^2} = 0$

$f(n) = o(n)$? F, because $\lim_{n\to\infty} \frac{3900n+57800}{n} = 3900$

Big-Ω Definition: It is said $f(n) = \Omega(g(n)) \iff$ there exists a constant $c > 0$ and $n_0 > 0$ such that

$$f(n) \geq c \cdot g(n) \text{ for all } n \geq n_0$$

Relation between O, o, Ω, θ

- $f = \theta(g) \iff f = O(g), f = \Omega(g)$
- $f = o(g) \Rightarrow f = O(g)$
- $f = O(g) \iff g = \Omega(f)$
- $f = \theta(g) \iff g = O(f)$

Caveats about constants

For $\log_4(n)$, is 4 ignorable? No, it matters

> $log_a(x) = log_a b log_b(x)$

Common Efficiency Class

Class	Name	
1	constant	No reasonable examples, most cases infinite input size requires infinite run time
$\log n$	Logarithmic	Each operation reduces the problem size by half, Must not look at the whole input, or a fraction of the input, otherwise will be linear
n	linear	Algorithms that scans a list of n items (sequential search)
$n \log n$	linearithmic	Many D&C algorithms e.g., merge sort
n^2	quadratic	Double embedded loops, insertion sort
n^3	cubic	Three embedded loops, some linear algebra algo.
2^n	exponential	Typical for algo that generates all subsets of a n element set.
$n!$	factorial	Typical for algo that generates all permutations of a n-element set

Merge Sort

$$T(n) = 2T(\frac{n}{2}) + 2n + c$$

$>$ $2n$: compare and copy n elements $= 2n$

Proof

$$T(n) = 2T(\frac{n}{2}) + 2n + c$$
$$\Rightarrow T(\frac{n}{2}) = 2T(\frac{n}{4}) + (n + c)$$
$$\Rightarrow T(\frac{n}{4}) = 2T(\frac{n}{8}) + (\frac{n}{2} + c)$$

Bring in $T(\frac{n}{2})$

$$\Rightarrow T(n) = 2\{2T(\frac{n}{4}) + (n + c)\} + 2n + c$$
$$= 2^2T(\frac{n}{2^2}) + 2 \cdot 2n + c + 2c$$

Bring in $T(\frac{n}{4})$

$$= 2^2\{2T(\frac{n}{8}) + (\frac{n}{2} + c)\} + 2 \cdot 2n + c + 2c$$
$$= 2^3T(\frac{n}{2^3}) + 3 \cdot 2n + c + 2c + 4c$$
$$\Rightarrow T(n) = 2^kT(\frac{n}{2^k}) + k \cdot 2n + (2^k - 1)c$$

Since $T(1) = 1$, $\frac{n}{2^k} = 1 \Rightarrow k = \log_2(n)$

$$\Rightarrow T(n) = n \cdot 1 + \log_2(n) \cdot 2n + (n - 1)c$$
$$\Rightarrow T(n) = O(nlogn)$$

Master Theorem

$$T(n) = aT(\frac{n}{b}) + cn^d$$
$$\Rightarrow T(n) = (1 + \frac{a}{b^d}^2 + \cdots + \frac{a}{b^d}^k)cn^d$$

a: number of subproblems

b: factor by which the problem size is reduced

d: exponent in the running time of the "combine" step

Case 1: $a < b^d$, or equiv. $d > log_b a, T(n) = O(n^d) \Rightarrow$ the cost of the "combine" step dominates the cost of the "split" step

Case 2: $a = b^d$, or equiv. $d = log_b a, T(n) = O(n^d \log n)$

Case 3: $a > b^d$, or equiv. $d < log_b a, T(n) = O(n^{log_b a})$

Proof by induction

> Useful for proving correctness of recursive algorithms

Devide and Conquer

Inversion Counting

Input: an A comtaining numbers $1, 2, \ldots, n$ in some arbitrary order

Output: the number of inversions in A, i.e. the number of pairs (i, j) of array indices with $i < j$ and $A[i] > A[j]$

It is useful for recommender systems $\Rightarrow$ fewer inversions means better alignment with user preferences.

Quick Select

Worst case: $T(n) = T(n-1) + cn \Rightarrow O(n^2)$

Best case: $T(n) = T(n/2) + cn \Rightarrow O(n)$

Median

Input: A list of numbers S; an integer k

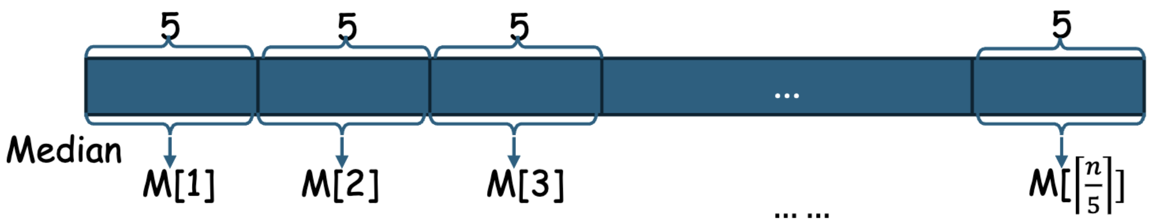
Output: the kth smallest element of S

0.0.1 A randomized divide-and-conquer algorithm

$$S_k =$$

Smart Selection of Pivot

Break array A into chunks of fives and find the median of each chunk, constructing a new array M of medians.



And **median** of M aka. MM is the pivot.

Claim: MM is at least $\geq \frac{3}{10}A$ and $\leq \frac{3}{10}A$

Proof: MM is the median of medians, so $\frac{1}{2}$ medians in M are $\leq MM$. $\because M = \frac{A}{5} \Rightarrow \frac{A}{10}$ medians are $\leq MM$. Moreover, For each $M[i]$, there are 3 elements $\leq M[i] \therefore \frac{3A}{10} \leq MM$

```
Select(A, k)
1. for i = 1, ..., n/5
2.     M[i] = median (a[5(i-1)+1], 5i)
3. MM = Select(M, n/10)
4. v = index of MM in A
5. r = partition(A, v)
6. if k=r: return A[r]
7. if k<r: Select(A[1,...,r-1], k)
8. if k>r: Select(A[r+1,...,n], k-r)
```

$$T(n) \leq O(n) + T(n/5) + T(7n/10) \Rightarrow T(n) = O(n)$$

Dynamic Programming

The main question in dynamic programming always is, what are the subproblems?

Fibonacci

$$\begin{gathered} T(n) = T(n-1) + T(n-2) + c \\ \Rightarrow T(n) \leq 2T(n-1) + c \\ \Rightarrow T(n) \leq 2^2 T(n-2) + 2c + c \\ \Rightarrow T(n) \leq 2^k T(n-k) + kc \\ k = n-1 \Rightarrow T(n) \leq O(2^n) \end{gathered}$$

$$T(n) \geq 2T(n-2) + c = \Omega(2^{\frac{n}{2}})$$

Applications of DP

- Unix diff for comparing files
- Bellman-Ford for shortest path
- CKY algorithm for natural language parsing
- etc.

Longest Increasing Subsequence

$$L(i) = \text{the length of a longest increasing subsequence ending at position i}$$
$$L(i) = (\max_{j:a_j<a_i} L(j)) + 1$$

Edit Distance

$$D(i,j) = \text{the edit distance between} s_1, s_2, \ldots, s_i \text{and} t_1, t_2, \ldots, t_j$$

Maximum Subarray

Because the decision at each index depends only on the previous index, it is uncessary to define $L(i, j)$ = the maximum subarray sum from index i to j. Instead, we can define $L(i)$ = the maximum subarray sum ending at index i.

Knapsack

> If the input is very sparse so that very few subproblems are solved, then top-down is a lot faster; if the input is very dense so that most subproblems are solved, then bottom-up is slightly faster (due to avoiding the overhead for recursion).

Q: For max W pounds, there are n items to pick from, of weight $w_1, \ldots, w_n$ and dollar value $v_1, \dot{;} v_n$. What is the most valuable combination of items he can fit into his bag?

E.g., take $W = 10$ and

Item	**Weight**	**Value**
1	6	\$30
2	3	\$14
3	4	\$16
4	2	\$9

0-1 knapsack problem

> go to local convenience store

> aka. the subset sum problem

Intuition: Go greedy

1. Greedily choose the most valuable item?
2. Greedily choose the most value/weight item?

Sadly, both can be found a counter example.

$K(w, j)$ = maximum value achievable using a knapsack of capacity w and items $1, 2, \ldots, j$

We should consider both capacity w and items j, where $0 \leq j \leq n$

$\Rightarrow$

The answer we seek is $K(W, n)$

Subproblems:

1. Choose the current item: $K(w - w_j, j - 1) + v_j$
2. Don't choose the current item: $K(w, j - 1)$

$$\Rightarrow K(w, j) = \max \begin{cases} K(w - w_j, j - 1) + v_j \\ K(w, j - 1) \end{cases}$$

$$(\text{The first case is invoked only if } w_j \leq w)$$

> we can express $K(w, j)$ in terms of subproblems $K(\cdot, j-1)$.

Time complexity: $O(nW)$

Unbounded knapsack problem

> aka. Knapsack with repetition

> like go to Costco (unlimited supply)

$$K(w) = \text{maximum value achievable with a knapsack of capacity } w$$

Express this in terms of smaller subproblems:

$$K(w) = \max_{i: w_i \leq w} \{K(w - w_i) + v_i\},$$

```
K(0) = 0
for w = 1 to W:
    K(w) = max {K(w - w_i) + vi: w_i <= w}
return K(W)
```

It is 1D dynamic programming. Each entry can take up to $O(n)$ time to compute, so the overall runtime is $O(nW)$

Optimal binary search tree

Q: what is the time complexity for this q?

TODO: study Longest common subsequence

Longest Common Subsequence

Greedy Algorithms

Greedy template:

```
Let T be the set of tasks H is the rule determinging the greedy algorithm

A = {}
While (T is not empty)
    Select a task t from T by a rule H
    Add t to A
    Remove t and all tasks imcompatible with t from T
```

Proving the optimality of a greedy algorithm

Greedy stays ahead

Exchange argument

Scheduling Theory

0.0.2 Interval Scheduling

```
f()
```

Prove:

the **Earlist-Finish-Time** algorithm is optimal.

We need to show that the solution chosen by greedy is as good as any other solution.

Usually, use the strategy of proof by contradiction

Minimum Spanning Tree

> aka. every node should be connected to every other node

We assume the graph is undirected in this class.

0.0.3 Applications of MST

- Network design
- Approximations to NP-hard problems like traveling salesman problem
- Clustering in machine learning

0.0.4 Greedy Template for MST

```
Init an empty set of edges T
While T is not yet a spanning tree (|T| <= |V| - 1)
    select $e \in E$ to add to $T$ according to a greedy criterion
Return T
```

Kruskal's Algorithm: sort edges by weight, add edge to T if it doesn't create a cycle

Prim's Algorithm: start at any node, add the lightest edge to T that connects a node in T to a node not in T

0.0.5 Proof of Correctness

Use **cut property**: If we can partition the graph into two parts, and an edge e is the cheapest edge connecting across the two parts, then e must be in the MST.

Huffman coding

Claim: The optimal code must be a full binary tree

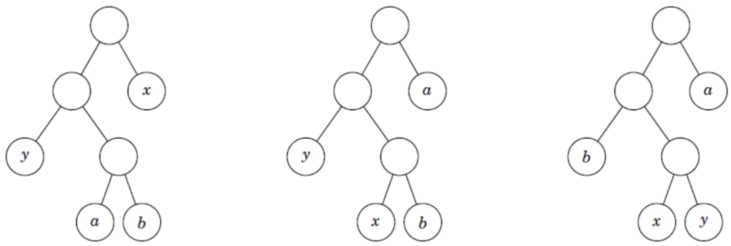
```
// C: the set of tokens, f: the frequency of each token
Huffman_recur(C, f)
    Identify the two least frequent tokens x, y in C
    C’ = C - {x, y} + {z} // z = x + y
    Assign frequency for z: f(z) = f(x) + f(y)
    T’ = Huffman_recur(C’, f)
    Append x and y as children of z in T’
    return T’
```

Proof Greedy Choice Property

Let x, y be the two least frequent tokens of C. There exists an optimal tree in which x and y are siblings.

$\Rightarrow$ use **the exchange argument**

- Let T be an optimal tree where x and y are not siblings.



T T' T''

- Identify the deepest sibling pair in T, let’s say they are a and b
- We can replace a with x, b with y, which will not increase the cost

Proof Corrrectness of Huffman’s Algorithm

Proof by induction:

Base case: $n = 2$ and 1 bit for each token

Inductive hypothesis:

Assume that the algorithm produce an optimal code for all token of $2, \ldots, k$

Inductive step:

Graph

Sparse vs. Dense

- Sparse: $|E| = \text{degree} \cdot |V| = O(|V|)$
  - The world is sparsely connected but well connected, which is known as the "small world phenomenon".
- Dense: $|E| = O(|V|^2)$
  - very small, e.g., family, single round robin tournament

Directed acyclic graph (DAG)

- E.g., software dependency, course prerequisite
- Each DAG has at least one **topological order**.

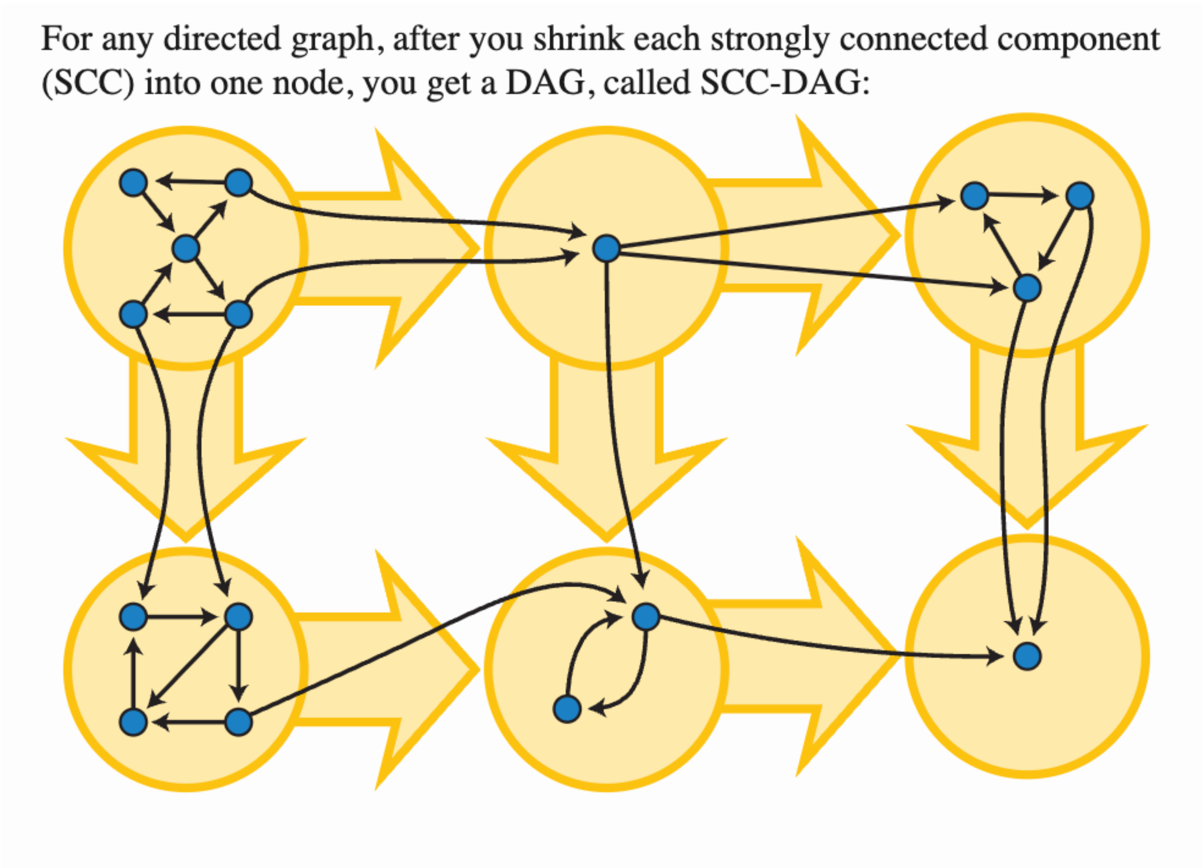


Figure 1: SCC DAG

0.1 Representations

Adjacency Matrix:

We use a $|V| \times |V|$ matrix A to represent the graph $G = (V, E)$, where each $A_{i,j} = 1$ or $c(i, j)$

Notes that:

- the matrix is symmetric for undirected graphs and asymmetric for directed graphs.
- the matrix Representation, using $O(|V|^2)$ space, is not efficient for sparse graphs.

Adjacency List:

```
0 -> 2 -> 3 -> 5 -> 6
    ^  \  |    ^  \
   /    \  v /     \
 1       > 4        > 7

adjacency list:    |    adjacency set:
0 -> [2]           |    0 -> {2}
1 -> [2]           |    1 -> {2}
2 -> [3, 4]        |    2 -> {3, 4}
3 -> [4, 5]        |    3 -> {4, 5}
4 -> [5]           |    4 -> {5}
5 -> [6, 7]        |    5 -> {6, 7}
6 -> []            |    6 -> {}
7 -> []            |    7 -> {}
```

```
   5      6
A ---> B ---> D
 \            ^
  \3        4/
   ---> C --/

adjacency list       | adjacency map
A -> [(B,5), (C,3)] | A: {B:5, C:3}
B -> [(D,6)]        | B: {D:6}
C -> [(D,4)]        | C: {D:4}
D -> []             | D: {}
```

In practice, if you don't need random edge access, you can use adjacency list, otherwise you should use adjacency set (for unweighted graph) or adjacency map (for weighted graph).

Graph Traversal

BFS:

- Time complexity of BFS is $O(|V| + |E|)$ because each vertex is visited once and each edge is traversed once.
- BFS is indeed the fastest algorithm for shortest-path on unweighted graphs. For example, for the coins problem (minimum number of coins to make up an amount), BFS would be much faster than Viterbi (or DP)
- Queue *Rightarrow* Priority Queue aka. Dijkstra's algorithm (for weighted graphs)

DFS:

Time complexity of DFS is $O(|V| + |E|)$ because each vertex is visited once and each edge is traversed once.

Linear Programming

A linear programming problem is a problem with a set of variables $x_1, x_2, \ldots, x_n$ that has

1. A linear objective function which can be minimized or maximized
   $\min \text{or} \max c_1x_1 + \cdots + c_nx_n$
2. A set of linear constraints $a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n \leq b_i$

> Linear Programming (LP) and Linear Regression have significant differences in mathematical tools and application objectives, but they may have indirect connections in certain specific contexts.

Different forms are equivalent

For example, for the question $2x \leq 5$, x is unrestricted in sign.

$\Rightarrow 2(x^+ - x^-) \leq 5, x^+ \geq 0, x^- \geq 0$

Example:

$$\max x_1 + x_2$$

s.t.

$$4x_1 - x_2 \leq 8$$
$$2x_1 + x_2 \leq 10$$
$$5x_1 - 2x_2 \geq -2$$
$$x_1 \geq 0, x_2 \geq 0$$

Turn this problem into a minimization problem with equality constraint

```
min -x_1 - x_2
1. 4x_1 - x_2 + s_1 = 8
2. 2x_1 + x_2 + s_2 = 10
3. 5x_1 - 2x_2 - s_3 = -2
4. x_1, x_2, s_1, s_2, s_3 >= 0
```

Problem: Maximum Flow

target: $\max f_{sa} + f_{sb} + f_{sc} \equiv \max f_{dt} + f_{et}$

Example of reduction

- Problems that don't look like a linear program sometimes can be <u>reduced</u> to LP
- Example:
  - You are given a set of n data points $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$
  - You want to learn a linear model that predicts y from x accurately: $y = ax + b$
  - In particular, you want to minimize the sum of absolute error:

$$\min_{a,b} \sum_{i=1}^{n} |ax_i + b - y_i|$$

- How can we solve this problem using LP?

To reduce the given problem to a linear programming (LP) problem, follow these steps:

Given Problem: We want to minimize the sum of absolute errors:

$$\min_{a,b} \sum_{i=1}^{n} |ax_i + b - y_i|$$

Step 1: Introduce Auxiliary Variables

Since absolute values are not directly handled in linear programming, introduce auxiliary variables t_i for each data point:

$$t_i \geq |ax_i + b - y_i|, \quad \forall i = 1, 2, \ldots, n$$

Step 2: Reformulate the Absolute Value Constraint

The absolute value constraint can be rewritten as two linear inequalities:

$$-t_i \leq ax_i + b - y_i \leq t_i, \quad \forall i = 1, 2, \ldots, n$$

Step 3: Formulate the LP Problem

The objective function can now be rewritten as:

$$\min_{a,b,t_1,t_2,\dots,t_n} \sum_{i=1}^{n} t_i$$

subject to the constraints:

$$ax_i + b - y_i \leq t_i, \quad \forall i$$
$$ax_i + b - y_i \geq -t_i, \quad \forall i$$
$$t_i \geq 0, \quad \forall i$$

This formulation is now a linear program since the objective function and constraints are all linear.

Conclusion:

By introducing auxiliary variables t_i, the problem is transformed into a standard LP problem. This LP can be solved using simplex or interior-point methods to find the best-fitting line $y = ax + b$ that minimizes the sum of absolute errors.

Reduction

Satisfiability

Literal: x_i or $\neg x_i$

Clause: A disjunction of clauses $c_j = x_1 \vee x_2 \vee \neg x_3$

CNF ϕ: A Boolean formula that is a conjunction of clauses $\phi = c_1 \wedge c_2 \wedge c_3 \wedge c_4$

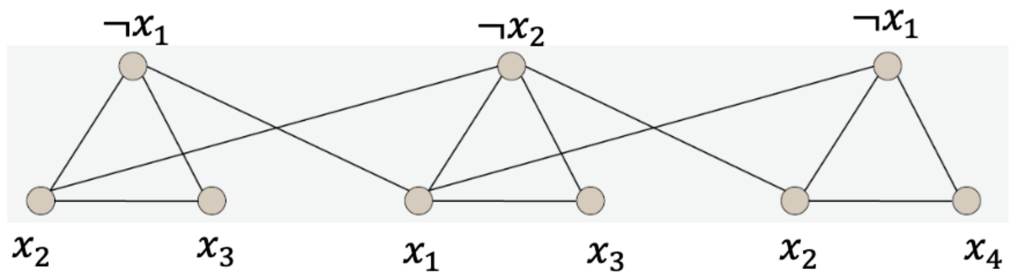
SAT: Given a CNF formula ϕ, does it have a satisfying truth assignment?

> Satisfying truth assignment: assign true/false to each variable such that the entire formula evaluates to true.

3-SAT to Independent Set

Construction from a formula ϕ to an independent set problem (G,k):

- For each clause of ϕ, construct 3 nodes for G, one for each literal
- Connect the 3 nodes in the same clause to each other into a triangle
- Connect literal to its negations in other clauses
- k = # of clauses in ϕ



$$\phi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$$

SAT reduces to 3-SAT

case where $k < 3$

$$k = 2, (l_1 \vee l_2) \Rightarrow (l_1 \vee l_2 \vee x) \wedge (l_1 \vee l_2 \vee \neg x)$$
$$k = 1, (l_1) \Rightarrow (l_1 \vee x \vee y) \wedge (l_1 \vee l_2 \vee \neg x)$$

case where $k > 3$

$$\Rightarrow (l_1 \vee l_2 \vee x_1) \wedge (\neg x \vee l_3 \vee x_2) \wedge (\neg x_2 \vee l_4 \vee x_3) \dots$$

if orignally is not satisfiable, new contruct is not satisfiable.

if $l_3 == T$ (statement is satisfiable), just set all free variables after l_3 to F and all free variables before l_3 to T.

3-SAT reduces to Vertex Cover

3-SAT: A set of clauses, each with exactly 3 literals. Identify whether there exists variables that can be set to true or false such that all clauses are satisfied.

Vertex Cover: Given graph G and integer k, does there exist a set of vertices V' such that $|V'| \leq k$ that covers all edges in G?

Computational Complexity

Clique

Certificate: A set of vertices $S \subseteq V$

Verification: Check if $|S| = k$ and that every pair of vertices in S is connected by an edge in E.

Reduction: $IS \leq_p Clique$

$G = (V, E) \Rightarrow G' = (V, E')$

E' is inverse of E.

Hamiltonian Cycle

Input: A undirected graph $G = (V, E)$

Question: Does there exist a cycle in G that visits each vertex exactly once?

Certificate: A candidate cycle C in G

Verification: Check if C is a cycle in G and that it visits each vertex exactly once.

Hamiltonian Cycle-2