Algorithm Design and Analysis (Fall 2022) Final Exam Cheat Sheet
Xiangyuan Xue (521030910387)

Time Complexity

Definition of big O notation:

- $T(n) = O(g(n))$:
  $$\exists C, n_0, \forall n > n_0 : T(n) \le C \cdot g(n)$$
- $T(n) = \Omega(g(n))$:
  $$\exists C, n_0, \forall n > n_0 : T(n) \ge C \cdot g(n)$$
- $T(n) = \Theta(g(n))$:
  $$T(n) = O(g(n)) = \Omega(g(n))$$

Master Theorem (Generalized)

Given $a \ge 1, b > 1, d \ge 0, w \ge 0$, suppose $T(n) = 1$ for $n < b$ and $T(n) = aT(\frac{n}{b}) + n^d \log^w n$, then

$$T(n) = \begin{cases} O(n^d \log^w n) & \text{if } a < b^d \\ O(n^{\log_b a}) & \text{if } a > b^d \\ O(n^d \log^{w+1} n) & \text{if } a = b^d \end{cases}$$

Just let $w = 0$, it falls back to original version.

Karatsuba

Let $z = (a+b)(c+d)$, $ad + bc = z - ac - bd$, thus

$$\begin{aligned} x \cdot y &= \left(a \cdot 10^{\frac{n}{2}} + b\right) \cdot \left(c \cdot 10^{\frac{n}{2}} + d\right) \\ &= ac \cdot 10^n + (ad + bc) \cdot 10^{\frac{n}{2}} + bd \\ &= ac \cdot 10^n + (z - ac - bd) \cdot 10^{\frac{n}{2}} + bd \end{aligned}$$

where time complexity is $O(3^{\log_2 n}) \approx O(n^{1.6})$.

Median of medians

An algorithm for selecting k-th element in S:

1. Partition S evenly into $\frac{n}{5}$ groups and directly find their $\frac{n}{5}$ medians.
2. Pick median of these $\frac{n}{5}$ medians as pivot and divide S into L, M, R by comparing to pivot.
3. If $k \le |L|$, select k-th element in L.
   If $|L| < k \le |L| + |M|$, pivot is the answer.
   If $|L| + |M| < k$, select $k - |L| - |M|$-th element in R.

It holds that $T(n) = T(0.2n) + T(0.7n) + O(n)$, by induction we can prove its complexity is $O(n)$.

Closest Pair

An algorithm to find the closest pair in 2-D plane:

1. Sort the points S by x and draw a vertical line l with $\frac{n}{2}$ points falling on each side.
2. Recursively find the closest pair on each side, where the distance is δ_L and δ_R.
3. Let $\delta = \min\{\delta_L, \delta_R\}$, drop points further than δ from l and sort remaining points S' by y.
4. For each $a \in S'$, check 7 points above in S' to find the closest pair, which is the answer.

Correctness relies on the limit of δ. Since $T(n) = 2T(\frac{n}{2}) + O(n \log n)$, its complexity is $O(n \log^2 n)$.

Fast Fourier Transform

Suppose p's degree is $D - 1$ and $\omega = e^{\frac{2\pi}{D} i}$, function $\text{FFT}(p, \omega)$ is described as follow:

1. If $\omega = 1$, return $\{p(1)\}$.
2. $p_e(x) = a_0 + a_2 x + a_4 x^2 + \cdots + a_{D-2} x^{\frac{D-2}{2}}$.
   $p_o(x) = a_1 + a_3 x + a_5 x^2 + \cdots + a_{D-1} x^{\frac{D-2}{2}}$.
3. $\{p_e(\omega^0), \ldots, p_e(\omega^{D-2})\} = \text{FFT}(p_e, \omega^2)$.
   $\{p_o(\omega^0), \ldots, p_o(\omega^{D-2})\} = \text{FFT}(p_o, \omega^2)$.
4. For $t = 0, 1 \ldots, D - 1$, we have
   $p(\omega^t) = p_e(\omega^{2t}) + \omega^t \cdot p_o(\omega^{2t})$.
5. Return $\{p(\omega^0), p(\omega^1), \ldots, p(\omega^{D-1})\}$

Complexity $T(n) = 2T(\frac{D}{2}) + O(D) = O(D \log D)$, which makes polynomial multiplication $O(n \log n)$. Inverse transformation is just $\text{FFT}(r, \omega^{-1})$.

Topological Sort

Run DFS on DAG and record finishing time. Sort vertices by descending order of finishing time, which is also the topological order. Just append the vertex as soon as it is finished, which is $O(|V| + |E|)$.
To prove correctness, we claim that if $finish[v] > finish[u]$, there will be no edge (u, v). By DFS properties, (u, v) won't be tree edge, forward edge or cross edge, and it can't be back edge on DAG, which proves the claim and promises correctness.

Algorithm Design and Analysis (Fall 2022) Final Exam Cheat Sheet
Xiangyuan Xue (521030910387)

Strongly Connected Component

We can find SCCs by DFS as follow:

1. Construct reversed graph G^R.
   DFS G^R and record finishing time.
2. Choose v with the largest finishing time.
   DFS from v in G and form new SCC.
3. Remove new SCC from both G and G^R.
   Repeat until G and G^R are empty.

Note that the vertex with largest finishing time in G^R is in tail SCC in G, and successive vertices in G won't be visited again because they are removed. This algorithm is proved to be $O(|V| + |E|)$.

Single Source Weighted Shortest Path

Dijkstra works without negative weights:

1. $T = \{s\}$, $dist[s] = 0$, $dist[v] = \infty$ $(\forall v \neq s)$.
   $dist[v] = w(s, v)$ $(\forall (s, v) \in E)$.
2. Find $v \notin T$ with smallest $dist[v]$.
   $T = T + \{v\}$.
3. $dist[u] = \min\{dist[u], dist[v] + w(v, u)\}$ $(\forall (v, u) \in E)$.
   Repeat until $T = V$.

Correctness relies on step 2. Just prove such v is optimal. Simple realization takes $O(|V|^2+|E|)$. By binary heap it can be $O((|V|+|E|)\log|V|)$, and by Fibonacci heap it can be $O(|E| + |V|\log|V|)$.

Negative Weighted Shortest Path

Bellman-Ford is slower but works:

1. $dist[s] = 0$, $dist[v] = \infty$ $(\forall v \neq s)$.
2. For each $(u, v) \in E$, update $dist[v] = \min \{dist[v], dist[u] + w(u, v)\}$
3. Repeat until none of $dist[v]$ is updated.

After k rounds, $dist[v]$ is the shortest distance of k-edge paths. It terminates as long as the shortest path includes no longer than $|V| - 1$ edges. Otherwise, a negative cycle exists and there is no shortest path, which can be figured out by Bellman-Ford. Time complexity is obviously $O(|V| \cdot |E|)$.

Minimum Spanning Tree

Kruskal and Prim are two typical greedy algorithms to solve MST. They are simple, so description is omitted. Both of them works with negative weights and can be proved by induction, replacement and contradiction.
Time complexity of Kruskal is $O(|E|\log|E|) = O(|E|\log|V|)$, with find-union set optimization.
Time complexity of Prim is $O(|E| + |V|\log|V|)$, with Fibonacci heap optimization.

Huffman Encoding

This greedy algorithm repeats $n-1$ rounds. In each round, it finds two minimized elements and merge them into a new element, which forms a hyper node on Huffman tree. It is $O(n\log n)$ with heap optimization. If elements are initially sorted, it can be $O(n)$ by maintaining two increasing lists.

To prove its correctness, reformulate the cost as

$$S = \sum_{k=1}^{n-1}(w_{k,x} + w_{k,y}) = \sum_{i=1}^{n} x_i d_i = \sum_{i\in C} w_i d_i$$

For any nodes w_i and w_j such that $w_i < w_j$ and $d_i < d_j$, swapping them will reduce S (Rearrangement Inequality). Therefore, S is minimum when nodes with smaller weight have larger depth.

Makespan Minimization

This problem is NP-hard, so consider an approximation algorithm LPT, where we:

- Choose the job with longest processing time.
- Insert the job into earliest finished machine.

It does not promise optimal solution S^*, but its solution S is no larger than $\frac{4}{3}$ of S^*.
To prove that, consider a stronger proposition. Suppose the shortest job is p_n, we have $3p_n \leq S^*$.
Assume $S^* < 3p_n$, we know every machine in optimal solution has no more than 2 jobs. However, LPT will find this optimal solution in this case, namely $S = S^* < \frac{4}{3}S^*$. Otherwise, we have $S \leq S^* + p_n \leq \frac{4}{3}S^*$. Now the ratio has been proved.

Algorithm Design and Analysis (Fall 2022) Final Exam Cheat Sheet
Xiangyuan Xue (521030910387)

Shortest Path on DAG
Find shortest path on DAG as follow:

1. Find a topological order of V and $dist[s] = 0$.
2. Solve $dist[u]$ by topological order.
   $$dist[u] = \min_{(v,u)\in E} dist[v] + w(v,u)$$

Time complexity of this algorithm is $O(|V| + |E|)$.

Potential Largest List
To find the largest number in k consecutive numbers, potential largest list can be used to reduce time complexity.

1. Construct a double-ended queue Q.
2. Sequentially traverse the list $a_1, a_2, \dots, a_n$.
3. If $Q.front.index \le i - k$, pop $Q.front$.
4. While $Q.back.value \le a_i$, pop $Q.back$.
5. Push back a_i and $Q.front$ is the largest number of $a_{i-k+1}, a_{i-k+1}, \dots, a_i$.

Charge the cost of maintaining to each a_i, time complexity of this algorithm is proved to be $O(n)$.

Longest Increasing Sequence
Find shortest path on DAG as follow:

1. Initialize increasing list $seq[0] = 0$.
2. For each a_i, find the first element in seq that is no smaller than a_i and replace it by a_i.
3. The final length of seq is the answer.

Time complexity of this algorithm is $O(|V| + |E|)$.

Minimizing Manufacturing Cost
Given a sequence of items $a_1, a_2, \dots, a_n$, minimize the cost to manufacture all the items where the cost $c(l,r) = C + \left(\sum_{i=l}^{r} a_i\right)^2$.

Let $s(x) = \sum_{i=1}^{x} a_i$, $g(x) = f(x) + s^2(x)$, By mathematical derivation, for any i and $x < y$, y is better than x if and only if $k_g(x,y) < s(i)$. Then any concave point cannot be optimal, so we can maintain a convex hull.

1. Let $i = 1, 2, \dots, n$ and calculate $f(i)$.
2. Let $j_1, j_2, \dots, j_m$ denote the convex hull.
3. While $k_g(j_k, j_{k+1}) \le s(i)$, kick j_k.
4. While $k_g(j_{k-1}, j_k) \ge k_g(j_k, i)$, kick j_k.
5. Now j_1 is the best choice.

Charge the cost of maintaining to each f_i, time complexity of this algorithm is proved to be $O(n)$.

All Pair Shortest Path
Floyd solves that with dynamic programming.

1. Initialize $dist(u,v) = w(u,v)$.
2. For each $k \in V$, for each $u, v \in V$, $dist(u,v) = \min\{dist(u,v), dist(u,k) + dist(k,v)\}$.

Time complexity of Floyd algorithm is $O(|V|^3)$.

Traveling Salesman Problem
Given a positive-weighted complete graph G. Find the cycle of $|V|$ vertices with the minimum weight.

Let $f(S,u,v)$ denote the shortest path from u to v via $S \subset V$ except u and v. The answer will be $\min_{u\in V} f(V,u,u)$. Then state transition equation is

$$f(S,u,v) = \min_{k\in S} \{f(S - \{k\}, u, k) + d(k,v)\}$$

For simplification, we fix u and calculate $f(S,v)$. Time complexity of this algorithm can be proved $O(n^2 \cdot 2^n)$, much better than $O(n!)$ brute force.

Network Flow
Given a directed graph $G = (V,E)$ with a source $s \in V$ and a sink $t \in V$. Each $(u,v) \in E$ is assigned a capacity $c(u,v)$. A flow $f : E \to \mathbb{R}_{\ge 0}$ satisfies

- $f(e) \le c(e)$
- $\sum_{v:(v,u)\in E} f(v,u) = \sum_{w:(u,w)\in E} f(u,w)$

namely capacity constraint and flow conservation.

Ford-Fulkerson Algorithm
Ford-Fulkerson algorithm solves max flow as follow:

1. Initialize empty flow f and residual flow G_f.
2. Find an s-t path p on G_f.
3. Push maximum amount of flow on G_f.
4. Update f and G_f.
5. Repeat until there is no such s-t path on G_f.

The correctness can be proved together with Max-Flow-Min-Cut Theorem. Its time complexity is $O(|E| \cdot f^*)$, which is not polynomial.

Algorithm Design and Analysis (Fall 2022) Final Exam Cheat Sheet
Xiangyuan Xue (521030910387)

Max-Flow-Min-Cut Theorem

The value of the maximum flow is exactly the value of the minimum cut

$$\max_{f} v(f) = \min_{\{L,R\}} c(L,R)$$

To prove this theorem, there are two main points:

- For any flow f and any cut $\{L,R\}$, we have $v(f) \leq c(L,R)$, namely any cut $c(L,R)$ provides an upper bound for max flow.

- There exists a cut $\{L,R\}$ such that $v(f) = c(L,R)$, namely some upper bound is reached by max flow with Ford-Fulkerson algorithm.

Integrality Theorem

If each $c(e)$ is an integer, then there exists a maximum flow f such that $f(e)$ is an integer for each $e \in E$.
The proof is trivial with Ford-Fulkerson algorithm. This theorem also promises the termination of Ford-Fulkerson algorithm with integral and rational capacity. Ford-Fulkerson algorithm does not always terminates with irrational capacity.

Edmonds-Karp Algorithm

Edmonds-Karp algorithm is exactly the same with Ford-Fulkerson algorithm, except for finding an s-t path p on G_f with BFS.
In this algorithm, an edge (u,v) becomes critical if the flow pushed along p is exactly $c(u,v)$. The distance of any vertex u is non-decreasing throughout the algorithm, so when (u,v) becomes critical again, the distance of u increases by at least 2. Note that at least 1 edge becomes critical in one iteration, its time complexity is $O(|V| \cdot |E|^2)$.

Dinner Table Assignment

Decide if it is possible to make an arrangement such that each table is shared by students from different universities.
Construct a vertex u_i for each university, a vertex v_i for each table. For each u_i, construct an edge (s, u_i) with capacity r_i. For each v_i, construct an edge (v_i, t) with capacity c_i. Any u_i and c_j are connected by an edge with capacity 1.
By capacity constraint and Integrality Theorem, such plan exists if and only if max flow is $\sum_{i=1}^{m} r_i$.

Tournament Prediction

Decide whether a team has a chance for champion.
Exclude the team we are considering. For any team u and v, construct a vertex u-v and an edge $(s, u\text{-}v)$, whose capacity is the number of competitions left between u and v. For each team, construct a vertex u and an edge (u,t), whose capacity is the number of winning allowed for u. Any u-v and w are connected by an edge with infinity capacity.
By capacity constraint and Integrality Theorem, such chance exists if and only if max flow is $\sum_{i=1}^{m} c_i$.

Maximum Bipartite Matching

Maximum matching on a bipartite graph can be solved by network flow similarly.
Even on a general graph, maximum matching can be solved in $O(|V|^2 \cdot |E|)$ time by Edmonds Blossom algorithm.

Independent Set and Vertex Cover

Given a graph $G = (V,E)$, S is an independent set if and only if $V - S$ is a vertex cover. Therefore, the maximum independent set is equivalent to the minimum vertex cover.
Both maximum independent set and minimum vertex cover are NP-hard, but they can be solved in polynomial time on a bipartite graph.
Construct a typical flow network for the bipartite graph. A cut $\{L,R\}$ divides the graph into four parts denoted by L_A, L_B, R_A, R_B, then $L_A \cup R_B$ is an independent set and $R_A \cup L_B$ is a vertex cover.
For a general graph, there is a 2-approximation algorithm to solve minimum vertex cover, where we find a maximal matching and choose all the endpoints.

Hall Marriage Theorem

Given a bipartite graph $G = (A,B,E)$ with $|A| = |B| = n$, G contains a perfect matching if and only if $|N(S)| \geq |S|$ for any $S \subseteq A$, where $N(S) = \{b \in B \mid \exists a \in S \subseteq A : (a,b) \in E\}$ is the set of all the neighbors of the vertices in S.

Linear Programming

The standard form of linear programming is

$$\begin{aligned} \max_{\boldsymbol{x}} \quad & \boldsymbol{c}^T \boldsymbol{x} \\ \text{s.t.} \quad & \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b} \\ & \boldsymbol{x} \geq \boldsymbol{0} \end{aligned}$$

The dual programming in standard form is

$$\begin{aligned} \min_{\boldsymbol{y}} \quad & \boldsymbol{b}^T \boldsymbol{y} \\ \text{s.t.} \quad & \boldsymbol{y}^T \boldsymbol{A} \geq \boldsymbol{c} \\ & \boldsymbol{y} \geq \boldsymbol{0} \end{aligned}$$

The original one is called the primal programming.

Algorithm Design and Analysis (Fall 2022) Final Exam Cheat Sheet
Xiangyuan Xue (521030910387)

Weak Duality Theorem
If $\boldsymbol{x}$ is a feasible solution to primal programming and $\boldsymbol{y}$ is a feasible solution to dual programming, then $\boldsymbol{c}^T\boldsymbol{x} \le \boldsymbol{b}^T\boldsymbol{y}$.

Strong Duality Theorem
Let $\boldsymbol{x}^*$ be the optimal solution to primal programming and $\boldsymbol{y}^*$ be the optimal solution to dual programming, then $\boldsymbol{c}^T\boldsymbol{x}^* = \boldsymbol{b}^T\boldsymbol{y}^*$.

from LP-Duality to Max-Flow-Min-Cut
Step 1: write down the linear program describing max-flow problem.

$$\begin{aligned}
\max_f \quad & \sum_{(s,u)\in E} f_{su} \\
\text{s.t.} \quad & \sum_{(v,u)\in E} f_{vu} - \sum_{(u,w)\in E} f_{uw} \le 0 \quad && \forall u \in V\backslash\{s,t\} \\
& \sum_{(u,w)\in E} f_{uw} - \sum_{(v,u)\in E} f_{vu} \le 0 \quad && \forall u \in V\backslash\{s,t\} \\
& f_{uv} \le c_{uv} && \forall (u,v) \in E \\
& f_{uv} \ge 0 && \forall (u,v) \in E
\end{aligned}$$

Step 2: write down the dual program describing fractional min-cut problem.

$$\begin{aligned}
\min_y \quad & \sum_{(u,v)\in E} c_{uv} y_{uv} \\
\text{s.t.} \quad & y_{su} + z_u \ge 1 && \forall u : (s,u) \in E \\
& y_{vt} - z_v \ge 0 && \forall v : (v,t) \in E \\
& y_{uv} - z_u + z_v \ge 0 && \forall (u,v) \in E : u \ne s, v \ne t \\
& y_{uv} \ge 0 && \forall (u,v) \in E
\end{aligned}$$

where $y_{uv} \in \{0,1\}$ describes whether edge (u,v) is cut and $z_u \in \{0,1\}$ describes whether vertex u is on the s-side.

Step 3: prove the dual program has integral optimal solution.
A matrix $\boldsymbol{A}$ is totally unimodular if every square submatrix D has determinant $|D| \in \{0,1,-1\}$. If $\boldsymbol{A}$ is totally unimodular and $\boldsymbol{b}$ is integral, then $P = \{\boldsymbol{x} : \boldsymbol{A}\boldsymbol{x} \le \boldsymbol{b}\}$ has integral vertices.
The constraints of the dual program can be written as

$$\begin{pmatrix} \boldsymbol{Y} & \boldsymbol{Z} \end{pmatrix} \begin{pmatrix} \boldsymbol{y} \\ \boldsymbol{z} \end{pmatrix} \ge \boldsymbol{b}$$

where $\boldsymbol{Y}$ is identity matrix and $\boldsymbol{Z}$ is totally unimodular, which can be proved by induction. Therefore, an integral optimal solution exists for the dual program. Since $y_{uv} \ge 2$ cannot be optimal, this optimal solution exactly solves min-cut problem.
Step 4: apply Strong-Duality Theorem to prove Max-Flow-Min-Cut Theorem.
By Strong-Duality Theorem, the primal program and the dual program have the same optimal value, so the size of max-flow equals the size of min-cut.

Zero-Sum Game
There two players A and B. Each player has a set of actions $\boldsymbol{a}$ and $\boldsymbol{b}$. For each pair of actions (a_i, b_j), an utility $u_A(a_i, b_j)$ and $u_B(a_i, b_j)$ is assigned to each player, where $u_A(a_i, b_j) + u_B(a_i, b_j) = 0$. They can also be expressed by payoff matrix $\boldsymbol{G} \in \mathbb{R}^{m\times n}$.
A strategy is a probability distribution of actions. A pure strategy specifies one of $a_1, a_2, \ldots, a_m$ with probability 1. Otherwise, it is a mixed strategy. The best response maximizes the player's utility.
Let $\boldsymbol{x}$ and $\boldsymbol{y}$ denote the strategies of A and B. The expected utility for A is $U_A(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{x}^T\boldsymbol{G}\boldsymbol{y}$. The expected utility for B is $U_B(\boldsymbol{x}, \boldsymbol{y}) = -\boldsymbol{x}^T\boldsymbol{G}\boldsymbol{y}$.

Minimax Theorem
Minimax Theorem shows that which player chooses strategy first does not matter, namely

$$\max_{\boldsymbol{x}} \min_{\boldsymbol{y}} \sum_{i,j} G_{ij} x_i y_j = \min_{\boldsymbol{y}} \max_{\boldsymbol{x}} \sum_{i,j} G_{ij} x_i y_j$$

Similarly, the left hand side can be formulated as the primal program and the right hand side can be formulated as the dual program. By Strong Duality Theorem, we can prove that Minimax Theorem holds.

The Complexity Class P
A decision problem $f : \sum^* \to \{0,1\}$ is in P if there exists a polynomial time algorithm that gives a correct decision. Finding an s-t path, deciding a k-flow and judging a prime number are all in P.

The Complexity Class NP
A decision problem $f : \sum^* \to \{0,1\}$ is in NP if there exists a polynomial time algorithm that verifies a hint correctly. SAT, VertexCover, IndependentSet, SubsetSum, HamiltonianPath are all in NP.
It is a trivial theorem that P $\subseteq$ NP.

NP Completeness
A decision problem f is NP-hard if $g \le_k f$ for any problem $g \in$ NP.
A decision problem f is NP-complete if $f \in$ NP and $g \le_k f$ for any problem $g \in$ NP.

Cook-Levin Theorem
Cook-Levin Theorem promises that SAT is NP-complete.
Note that a CNF formula is sufficient to simulate the execution of a Turing Machine, so SAT is weakly harder than any problem in NP.

Algorithm Design and Analysis (Fall 2022) Final Exam Cheat Sheet
Xiangyuan Xue (521030910387)

Basic NP-Complete Problem

Basic NP-complete problems are as follow:

- SAT (Boolean Satisfiability Problem): Given a CNF formula ϕ, decide if there is a value assignment to the variables to make ϕ true.
- 3SAT: A 3-CNF formula is a CNF formula where each clause contains at most 3 literals. Given a 3-CNF formula ϕ, decide if there is a value assignment to the variables to make ϕ true.
- IndependentSet: Given a simple graph $G = (V, E)$ and $k \in \mathbb{Z}^+$, decide if the graph has an independent set of size k.
- VertexCover: Given a simple graph $G = (V, E)$ and $k \in \mathbb{Z}^+$, decide if the graph has a vertex cover of size k.
- Clique: Given a simple graph $G = (V, E)$ and $k \in \mathbb{Z}^+$, decide if the graph contains clique (a complete subgraph) of size k.
- SubsetSum: Given a collection of integers $S = \{a_1, a_2, \dots, a_n\}$ and $k \in \mathbb{Z}^+$, decide if there is a sub-collection $T \subseteq S$ such that $\sum\limits_{a_i \in T} a_i = k$.
- SubsetSum+: Given a collection of positive integers $S = \{a_1, a_2, \dots, a_n\}$ and $k \in \mathbb{Z}^+$, decide if there is a sub-collection $T \subseteq S$ such that $\sum\limits_{a_i \in T} a_i = k$.
- VectorSubsetSum: Given a collection of integral vectors $S = \{\boldsymbol{a}_1, \boldsymbol{a}_2, \dots, \boldsymbol{a}_n : \boldsymbol{a}_i \in \mathbb{Z}^m\}$ and $\boldsymbol{k} \in \mathbb{Z}^m$, decide if there is a sub-collection $T \subseteq S$ such that $\sum\limits_{\boldsymbol{a}_i \in T} \boldsymbol{a}_i = \boldsymbol{k}$.
- Partition: Given a collection of integers $S = \{a_1, a_2, \dots, a_n\}$, decide if there is a partition of S to A and B such that $\sum\limits_{a \in A} a = \sum\limits_{b \in B} b$.
- Partition+: Given a collection of positive integers $S = \{a_1, a_2, \dots, a_n\}$, decide if there is a partition of S to A and B such that $\sum\limits_{a \in A} a = \sum\limits_{b \in B} b$.
- DominatingSet Given a simple graph $G = (V, E)$, a dominating set is a subset of vertices S such that for any $v \in V \backslash S$, there is a vertex $u \in S$ that is adjacent to v. Given an integer $k \in \mathbb{Z}^+$, decide if the graph contains a dominating set of size k.
- DirectedHamiltonianPath Given a simple graph $G = (V, E)$, a source $s \in V$ and a sink $t \in V$, decide if there is a Hamiltonian path from s to t.
- HamiltonianPath Given a simple graph $G = (V, E)$, decide if G contains a Hamiltonian path.
- HamiltonianCycle Given a simple graph $G = (V, E)$, decide if G contains a Hamiltonian cycle.

Key Points for Reduction

Some key points for reduction are as follow:

- SAT $\leq_k$ 3SAT: A long clause can be broken into shorter ones: $(x_1 \vee x_2 \vee \neg x_3 \vee \neg x_4) = (x_1 \vee x_2 \vee y_1) \wedge (\neg y_1 \vee \neg x_3 \vee \neg x_4)$.
- 3SAT $\leq_k$ IndependentSet: Construct a triangle for each clause and connect every x_i with $\neg x_i$.
- VertexCover $\leq_k$ VectorSubsetSum: For each $v_i \in V$, construct a vector $\boldsymbol{a}_i$ where $a_i[0] = 0$ and $a_i[j] = 1$ if v_i is on edge e_j. For each $e_j \in E$, construct a vector $\boldsymbol{b}_j$ where only $b_j[j] = 1$. Then $\boldsymbol{k} = (k, 2, 2, \dots, 2)$.
- SubsetSum+ $\leq_k$ Partition+: Append a positive integer $b = \left|2k - \sum\limits_{i=1}^{n} a_i\right|$.
- VertexCover $\leq_k$ DominatingSet: For each $e \in E$, construct a intermediate vertex v_e representing e. For the original vertices, connect them into a clique.
- 3SAT $\leq_k$ DirectedHamiltonianPath: For each variable x_i, construct a gadget S_i. The direction we pass S_i determines the value of x_i. For each clause c_j, create a vertex v_j and connect v_j with S_i if x_i is in c_j.
- DirectedHamiltonianPath $\leq_k$ HamiltonianPath: Break each vertex into multiple vertices on a single path.