The pricing method: vertex cover :-

Given a graph $G=(V,E)$. The vertex cover of G is a subset $S \subseteq V$ that covers every edge in E.
i.e. for any edge in E, at least one end point is present in S.

→ In the vertex cover problem, we aim to find the minimum cardinality subset of V that covers every edge.

→ Here, we will consider a general version of the vertex cover problem in which each vertex $i \in V$ has a weight $w_i \geq 0$. (weighted vertex cover).
So, the weight of the vertex cover S can be defined as,

$$W(S) = \sum_{i \in S} w_i$$

→ Our goal is to find a vertex cover S for which $W(S)$ is as minimum as possible.

→ If we make all weights equal to 1, this problem corresponds to the original decision VC problem.

→ We know VC is a NPC problem. So, we cannot have a p-time algorithm for it.

Towards a sol^n:-

1. Approximation through reduction :-

→ In our previous discussion, we have shown an approximation sol^n for set cover problem.

→ We know vertex-cover can be reduced to set-cover

$$VC \leq_p SC.$$

→ So, we can use the above two statements to formulate a theorem.

claim:- We can use the approximation algorithm for Set Cover problem to develop an $H(d)$-approximation algorithm for the weighted vertex cover problem, where d is the maximum degree of any vertex in the graph.

If:-

Greedy-Vertex-Cover1 (G, w)

1. Construct a set cover instance as follows:

$U \leftarrow G.E$, $S \leftarrow \phi$

for each vertex $i \in V$

$S_i \leftarrow$ all edges incident on i

$S \leftarrow S \cup \{S_i\}$

$w_i \leftarrow w_i$

2. return the soln to Greedy-set-cover (U, S, w)

Now, the above algorithm can be used to achieve an approximate soln to the vertex cover with approximation ratio $H(d)$.

→ However, we can not achieve an approximation soln in every general case.

example:- Independent set $\leq_p$ VC.

let's say we have an α-approximation soln for VC, that means, if the optimal VC has size $|V|/2$, our approximation soln will have a size $|V|$. Now, if we use the reduction to generate an IS.

the size of IS = $|V|$ − size of VC

$= |V| - |V| = 0$ (not possible)

as optimal IS will have a size = $|V|/2$.

The pricing Method:- (primal dual method).

In the pricing method, solution for vertex cover, we think the weights associated with the vertices as their cost, and each edge has to pay for its share of the cost when getting covered in the vertex cover.

→ A similar analysis we did in case of set cover algorithm. Each set S_i was associated with some cost w_i and each element s has to pay its share of the cost C_s when they get selected.

So, the total cost of the set cover is, $\sum_{s \in U} C_s = \sum_{S_i \in C} w_i$.

We could achieve an approximation ratio of $H(d^*)$ due to some "fairness" property of the cost shares which put a bound on the cost share of a set S_k as —

$$\sum_{s \in S_k} C_s \leq H(|S_k|) \cdot w_k \quad \text{—— (1)}$$

→ When using pricing technique to solve VC problem, we think of the weight w_i of a vertex 'i' to be the cost of using 'i' in the VC.

→ The edge e is an "agent" that can pay an amount $p_e \geq 0$ to the vertex that covers it.

→ The algorithm for VC will find a set $S \subseteq V$ that covers every edge in E and also compute the prices value p_e for each $e \in E$.

so that $\sum_{e \in E} p_e \leq \sum_{i \in S} w_i$

Fairness property:- If we look into the fairness property of the set cover sol^n specified in eq^n (1), the cost of each element of S_k in total should correspond to the w_k. but it's not as some elements may get selected through some other set. However, in our vertex cover problem,

we try to address this issue by proposing some natural fairness rules for the prices.

→ Selecting a vertex 'i' covers all its ~~adjacent~~ incident edges. So, it would be unfair to charge these incident edges in total, more than the cost of vertex i.

→ Based on the above notion, we call prices p_e fair if, for each vertex 'i', the edges incident on 'i' don't have to pay more than the cost of the vertex:

$$\sum_{e=(i,j)} p_e \leq w_i \quad \forall i \in V \quad \text{---} (2)$$

This is an exact fairness condition while the equation (1) shows an approximate fairness condition.

→ As we consider the prices of the edges to be fair, it gives a lower bound on the cost of any set as -

Theorem 1:- for any vertex cover S^* and any non-negative and fair prices p_e, we have $\sum_{e \in E} p_e \leq w(S^*)$.

Proof:- from equation (2), the fairness property states that,

$$\sum_{e=(i,j)} p_e \leq w_i \quad \forall i \in V$$

Now, adding it for all nodes (vertices) in S^*, we get

$$\sum_{i \in S^*} \sum_{e=(i,j)} p_e \leq \sum_{i \in S^*} w_i = w(S^*) \quad \text{---} (3)$$

This sum includes p_e for each edge at least once as S^* covers every edges. S^* may have two vertices covering the same edge, so, p_e may get added more than once for such edges.

So $$\sum_{e \in E} p_e \leq \sum_{i \in S^*} \sum_{e=(i,j)} p_e \quad \text{---} (4)$$

$$\leq w(S^*) \quad \text{(proved)}$$

The algorithm:-

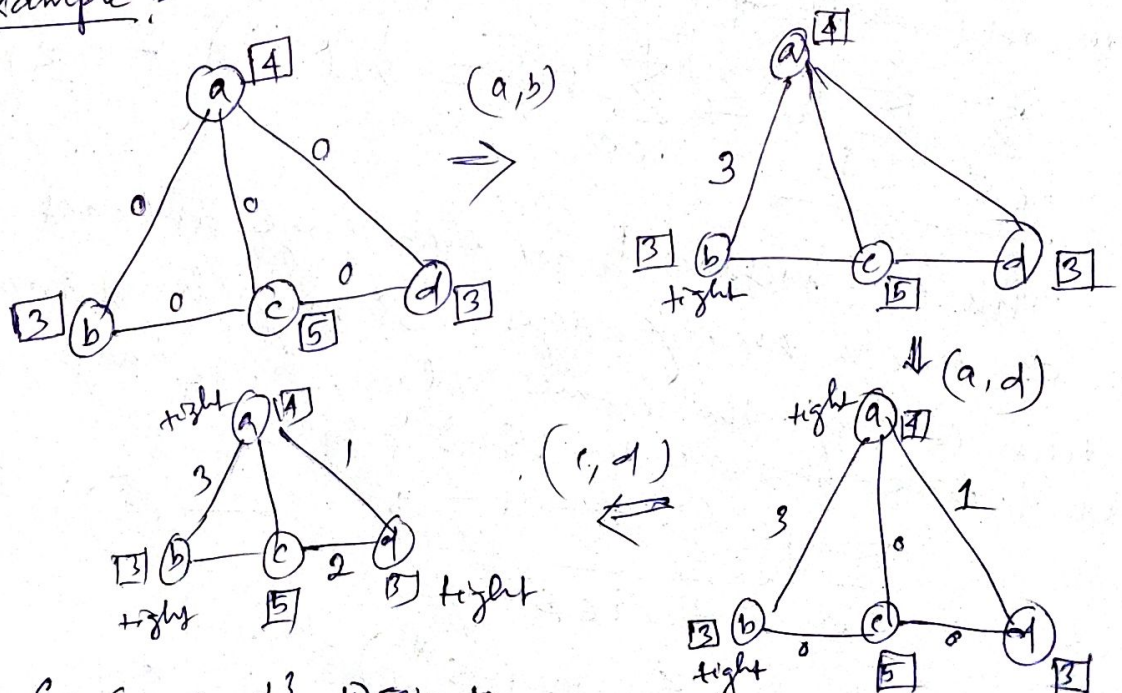
The algorithm simultaneously finds the vertex cover & sets the prices of edges (greedily).

→ It greedily sets the prices of the edges
→ Then it uses this prices to select the nodes of the vertex cover.
→ We define a node i to be "tight" (paid for) if $\sum_{e=(i,j)} p_e = w_i$

Approximate-vertex-cover(G, w)

1. Set $p_e = 0 \quad \forall e \in E$
2. while there is an edge $e=(i,j)$ such that neither i
3. nor j is tight
4. select such an edge e.
5. Increase p_e ~~without~~ as much as possible without violating fairness
6. end while.
7. S ← set of all tight nodes
8. return S as the VC

Example:-



S = {a, b, d} w(S) = 10

But this is clearly not optimal. The optimal vertex would be, $S = \{a, c\}$ $w = 9$

Analysis:-

In the algorithm and the discussed example, it may seem that the total cost of the vertex cover is fully paid by the prices of the edges of the graph. but it is not the case, as an edge e can be incident on more than one vertices. (i.e, both ends of e are in S) In this case, e has to pay twice for each of its end points.

→ In the example (a,b) and (a,d) are the edges with both end points in S.

So, Total cost of vertex cover = $\sum_{e_1 \in E} (2p_{e_1}) + \sum_{e_2 \in E} p_{e_2}$

where e_1 → edges with both end points are in S.

e_2 → edges with one end points are in S.

~~This gives us a bound~~

This may seem unfair to charge the e_1 type edge twice, but this unfairness can be bounded by a constant factor of 2.

i.e. each edge has to pay a maximum of 2 times its price.

Theorem 2 :- The weight/cost of the set cover S' is bounded in terms of the prices of the edges as, $w(S) \leq 2 \cdot \sum_{e \in E} p_e$

Proof:- As all the vertices in S are tight $w_i = \sum_{e=(i,j)} p_e \quad \forall i \in S$

Now, adding over all nodes in S, $w(S) = \sum_{i \in S} w_i = \sum_{i \in S} \sum_{e=(i,j)} p_e$

As, each edge has to pay a max^m of 2 times its price, $\boxed{w(S) \leq 2 \sum_{e \in E} p_e}$

Theorem 3: The algorithm returns a vertex cover S with cost atmost twice the minimum cost of any vertex cover. i.e. $W(S) \leq 2 \cdot W(S^*)$.

Proof: First, let prove that S is a VC.

If S is not a vertex cover, that means at least one edge $e = (i, j)$ is not covered by S.

As S covers all tight vertices, either i or j or both are not tight. This contradicts with the termination cond" of the while loop.

→ To prove the theorem 3, we can directly use the stmt of Theorem 2 and 1

As per Theorem 2,

$$W(S) \leq 2 \cdot \sum_{e \in E} p_e$$

As per Theorem 1, $\sum_{e \in E} p_e \leq W(S^*)$

$$\Rightarrow \boxed{W(S) \leq 2 \cdot W(S^*)}$$

The knapsack Problem

When we try to solve an NPC/NPH problem using approximation algorithms, we expect a soln which is close to the optimal soln.

→ But, the algorithms that we have studied so far, are not that close i.e. for confer selecting & VC: 9 is 100% more for SC even worse.

→ Though the NPC class problems are equivalent in terms of their complexities, but they differ significantly in the extent to which their soln can be approximated.

→ Now, we consider the knapsack problem, for which it is possible to achieve a strong approximation in polynomial time.

Definition: - Given n items each with its weight and value. we try to fill a knapsack of capacity W ~~such that the value~~ with a subset of items such that the total value is maximum and the aggregate weight is not more than W.

$$\text{maximize} \sum_{i \in S} v_i \quad \text{subject to.} \sum_{i \in S} w_i \leq W.$$

→ Here, we can achieve a ~~(1+ε)~~ polynomial-time approximation scheme.

Polynomial time approximation scheme: —

→ An approximation scheme is an approximation algorithm that takes as input, an instance of the problem and a value $\epsilon > 0$ such that for any fixed ϵ, the scheme is a $(1+\epsilon)$ - approximation algo.

→ If the scheme runs in polynomial time, it is called a PTAS. for any fixed $\epsilon > 0$.

→ As we keep decreasing the value of ϵ, the running time increases.

→ An approximation scheme is a fully polynomial-time approximation scheme, if ~~it is an appro~~ its running time is polynomial in both $(1/\epsilon)$ and the size of input instance n. [example: $O((1/\epsilon)^2 n^3)$

→ For a FullyPTAS, any constant factor decrease in ϵ results in a constant factor increase in the running time.

→ For our knapsack problem, the approximation soln will be less than the optimal soln by a factor $(1+\epsilon)$.

→ How can we get such close soln to the optimal for a NPH problem knapsack:-

→ knapsack is solved by DP-soln with complexity $O(nW)$

→ If the value of the W is small, then it behave like a polynomial soln.

But as W increases the running time also increases and may attain a value which is beyond polynomial limits in terms of no. of items (problem size) n.

So, this is a Pseudo polynomial soln.

↳ an algorithm whose running time is polynomial in the numeric value of the input rather than the size (no. of) of input.

→ Some NPC problems have Pseudo-polynomial soln's they are called as Weakly NPC.

As we already know, knapsack ~~runs in O(nW) time~~ can be solved by a DP-soln in O(nW) time optimizing the $\sum_{i \in S} w_i$ s.t $\sum_{i \in S} w_i \leq W$ for integer weights.

→ It is a pseudopolynomial solution as the running time depends on the ~~value~~ weight values.

→ If the weight are small (even if the values are big), we can achieve a polynomial soln

→ We can modify the DP-based soln for the case when the values are small even if the weights may be large.

→ Using this strategy, we can achieve a DP based soln with running time $O(n^2 v^*)$ where

n = no. of items and $v^* = \max_i v_i$ max value.

This is a Pseudo polynomial soln as the running time depends on the values of v_i.

Idea:- If the values are small, (v^* is small) then the soln is polynomial.

→ However, if v^* is large, we go for approximate solve

→ We will use a rounding parameter b to round off the values given in the problem to an integer multiple of b:-

for each item 'i', its rounded value = $\tilde{v}_i = \left\lceil \frac{v_i}{b} \right\rceil b$.

~~Then we will solve using DP on these rounded values~~ which are close to v_i

i.e $v_i \leq \tilde{v}_i \leq v_i + b$.

→ Then we solve using ~~these rounded~~ DP on values $\hat{v}_i$

$$\hat{v}_i = \tilde{v}_i / b = \left\lceil \frac{\tilde{v}_i}{b} \right\rceil \quad \text{for } \forall i$$

claim. The knapsack problem with values $\tilde{v}_i$ and the scaled problem with $\hat{v}_i$ have the same set of optimal solutions where the optimal values differ exactly by a factor of b, and the scaled values are integers.

→ Assume that all items have weight atmost W (capacity of knapsack)

→ Also assume that $1/\epsilon$ is an integer (simplicity)

Approximate-knapsack(n, W, v, ϵ)

set $b \leftarrow \left(\frac{\epsilon}{2n}\right) \cdot \max_i v_i$

Solve the knapsack problem with values $\hat{v}_i$ (equivalently $\tilde{v}_i$)

Return the set S of items found.

DP-based sol^n to knapsack ~~with problem as~~ to minimi

weight and achieve a particular value:

→ The DP sol^n we have studied for 0/1 knapsack form subproblems as OPT(i, W) which denotes the subproblem of finding the maximum value of any solution using a subset of the items 1, ..., i and a knapsack of weight W.

→ But, here we need ~~a~~ subproblems as: OPT(i, V).

finding the smallest weight of knapsack W so that one can obtain a sol^n using a subset of items 1, ..., i with a value of at least V.

In our subproblems: $i = 1, \ldots n$

$$V = 0, \ldots \sum_{j=1}^{i} v_j$$

→ If v^* denotes the $\max(v_i)$, then the largest V can get

in a subproblem is $\sum_{j=1}^{n} w_j \leq n V^*$.

→ As the values are integral, we can have atmost $O(n^2 \cdot v^*)$ subproblems.

From these subproblems, we can obtain the value of the original knapsack problem as follows:-
It is the largest value V such that

$OPT(n, V) \leq W$.

Recursive definition:-

$$OPT(i, V) = \begin{cases} 0 & \text{if } V = 0 \\ w_n + OPT(n-1, V-v_n) & \text{if } V > \sum_{i=1}^{n-1} v_i \\ \min\left(OPT(n-1, V),\ w_n + OPT\left(n-1, \max(0, V-v_n)\right)\right) & \end{cases}$$

$$OPT(i, V) = \begin{cases} 0 & \text{if } V \leq 0 \quad \text{—①} \\ w_i + OPT(i-1, V-v_i) & \text{if } V > \sum_{i=1}^{n-1} v_i \quad \text{—②} \\ \min\left\{OPT(i-1, V),\ w_i + OPT\left(i-1, \max(0, V-v_i)\right)\right. & \text{—③} \end{cases}$$

eq ② :- if $V > \sum_{i=1}^{n-1} v_i \Rightarrow$ the nth item has to be selected

eq 3 :- if not eq ②, nth item can be selected or not, depending upon the weight value.
first part ($OPT(i-1, V)$) correspond to a case where nth item is not selected
second part is when nth item is selected :-
two subcases - ① if nth item is the only item in OPT
② not the only item.

DP_knapsack(n)

M[0, ... n, 0 ... V]

for i ← 0 to n do

M[i, 0] = 0.

endfor

for i = 1 to n

for V = 1, to $\sum_{j=1}^{i} v_i$

if $V > \sum_{j=1}^{i-1} v_j$ then

$M[i, V] \leftarrow w_i + M[i-1, V - v_i]$.

else

$M[i, V] = \min(M[i-1, V], w_i +$

$M[i-1, \max(0, V - v_i)]$

endif

endfor

endfor

Return the max value V for which $M[n, V] \leq W$.

Theorem:- The algorithm Approx_knapsack runs in P time for a fixed $\epsilon > 0$.

$(n^2 v^*)$. j $\hat{v}_j = \left\lceil \frac{v_j}{b} \right\rceil = \frac{v_j \times n}{\epsilon * v_j}$

$= \frac{n}{\epsilon}$

$\left(n^2 \frac{n}{\epsilon}\right) = (n^3 \epsilon^{-1})$

Theorem:- $S \rightarrow$ approx
$S^* \rightarrow$ optimal

$$(1+\epsilon) \cdot \sum_{i \in S} v_i \geq \sum_{i \in S^*} v_i$$

$$\sum_{i \in S^*} v_i \leq \sum_{i \in S^*} \tilde{v}_i$$

$$\leq \sum_{i \in S} \tilde{v}_i$$

$$\leq \sum_{i \in S} (v_i + b) \qquad |S| \leq n$$

$$\leq \sum_{i \in S} v_i + nb$$

$$\leq \sum_{i \in S} v_i + n \cdot \frac{\epsilon}{2n} \cdot v_{max}$$

$$\leq \sum_{i \in S} v_i + \frac{1}{2} \epsilon v_{max}.$$

$$\frac{val(S^*)}{val(S)} \leq (1+\epsilon)$$

$$(1+\epsilon) \cdot \sum_{i \in S} v_i \geq \sum_{i \in S^*} v_i$$

$$\sum_{i \in S^*} v_i \leq \sum_{i \in S^*} \tilde{v}_i$$

$$\leq \sum_{i \in S} \tilde{v}_i$$

$$\leq \sum_{i \in S} (v_i + b)$$

$$\leq \sum_{i \in S} v_i + nb.$$

$$\leq \sum_{i \in S} v_i + n \cdot \left(\frac{\epsilon}{2n}\right) \cdot v_{max} \quad b = \frac{1}{2}$$

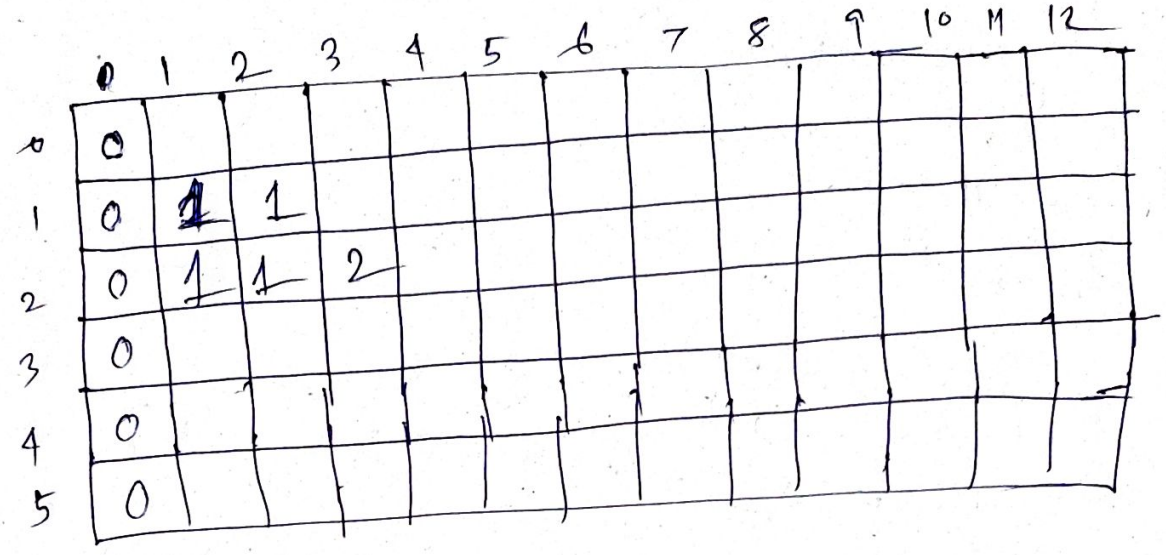
$$\leq \sum_{i \in S} v_i + \frac{1}{2} \cdot \epsilon \, v_{max}.$$

$v_m - \frac{1}{2} v_m \leq \sum_{i \in S} v_i$

$v_{max} \leq$

Example :=

items	1	2	3	4	5
v_i	2	3	1	4	2
w_i	1	2	5	6	7



$i=1$, $v=1$ to 2.

$v=1$ $M[1,1] = w_1 + M[0,1]$

$v=2$ $M[1,2] = w_1 + M[0,0] =$

$i=2$ $v=1$ to 5

$v=1$ $M[2,1] = 1$

$v_i = 1 > \sum_{j=1}^{i-1} v_j$ 2 X

$\min\{ M[i,1], 2 + M[1, \max(0, 1-3)] \}$

$\min(1, 2)$

$v=2$ $M[2,2] = 1$

$v=2 > 2$ X

$\min\left(M[1,2], 2 + \underbrace{M[1,-1]}_{0}\right)$

$\min(1, 2)$

$v=3$ $M[2,3] = w_2 + M[1,0] = 2$.

$v=3 > 2$ ✓

Knapsack (n, $W[1 \ldots n]$, $v[1 \ldots n]$, W)

$M[0 \ldots n, 0 \ldots V]$

for $i = 0$ to n
  for $j = (-v_{max})$ to 0
    $M[i, j] = 0$.

for $i = 1$ to n
  for $j = 1$ to $\sum_{j=1}^{i} v_i$
    if $V > \sum_{j=1}^{i-1} v_j$ then
      $M[i, j] = W_i + M[i-1, V - v_i]$
    else
      $M[i, j] = \min(M[i-1, V], w_i + M[i-1, \max(0, V - v_j)])$
    end if
  end for
end for
return the max m value of v for which $M[n, V] \leq W$.

$(n^2 \cdot v^*)$

$b = \frac{\epsilon}{2n} \cdot v_{max}$ $\quad \frac{2n}{\epsilon}$ $\quad n \cdot m \cdot v_{max}$

$\hat{v}_i = \left\lfloor \frac{v_j}{b} \right\rfloor$

$v_i \leq \hat{v}_i \leq v_i + b$ $\quad 2n$

$\hat{v} = \frac{v_i}{\epsilon \cdot v_{max}} \cdot 2n = \frac{2 \cdot n \cdot v_i}{\epsilon \cdot v_{max}} = \frac{2 \times 5 \times 934221}{2734399}$

Local Search :-

It is a general problem solving technique;

→ It describes algorithms that explore the space of possible solutions in sequential fashion, moving one step from a current soln to a nearby soln

→ It is so general and flexible that, it can be used to find soln's to almost any type of NPC/NP-H problems.

But it is not easy to establish the quality of the soln. So, we may not differentiate essentially a good local search technique from a bad one.

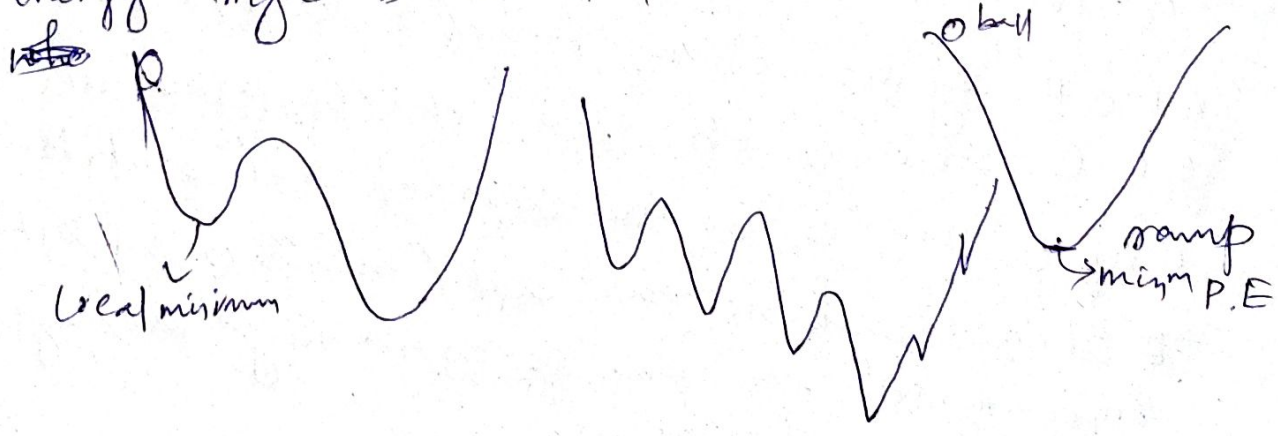
→ So, the local search algorithms are generally heuristics designed to find good, but not necessarily optimal solutions to computationally hard problems

→ The exploration of the solution space in local search resembles with the energy minimization principles in physics.

The landscape of an optimization problem :

Physical systems perform minimization, when they seek to minimize the potential energy.

→ If an object falls from a height 'h', its potential energy 'mgh' is converted into the kinetic energy



Connection to optimization. —

→ The physical system can be in one of the possible states from a large set of states.
→ Its energy is a function of its current state, and the states change to neighboring state through a small perturbation.
→ The energy landscape is a structure depicting how these states are linked together.

→ For a typical minimization problem, we have a large set C of possible solutions.
→ we have a cost function $c(\cdot)$ that measures the quality of each solution,
for a soln $S \in C$, we have the cost as $c(S)$.
The goal is to find a solution ~~State~~ $S^* \in C$ for which $c(S^*)$ is as small as possible.
→ A soln S' is the neighbor of another soln S if S' can be obtained by a small modification of S. ($S \sim S'$) :- S' is the neighbor of S.
$N(S)$: Set of neighborhood of S.
↳ symmetric

→ A local search algorithm works as follows:
At all time, it maintains a current soln $S \in C$. In a given step, it chooses a neighbor S' of S, then S' be the current soln and iterates.
→ It remembers the best (min cost) soln encountered so far., gradually moving towards better solutions.
→ The core of the local search algorithm is the neighbor selection procedure.

Gradient descent :- An application to vertex cover

VC :- Given a graph $G = (V, E)$, find a subset of nodes S of minimal cardinality such that for each $(u, v) \in E$, either u or v or both in S.

Neighbor rel^n :- $S \sim S'$ if S' can be reached from S by adding or deleting a single node.

→ Each vertex cover 'S' has at most 'n' neighbors

Gradient Descent :- Start with $S = V$. If there is a ~~vertex cover~~ neighbor S' which is a vertex cover and has lower cardinality, replace S with S'. otherwise terminate.

observation :- Algorithm terminates after atmost n steps. since each update decrease the size of the vertex cover by 1.

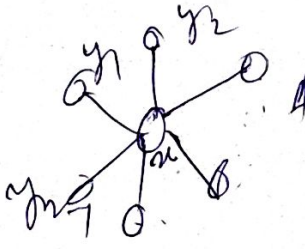
Analysis :-

We can relate the behaviour of Gradient descent method to the energy landscapes.

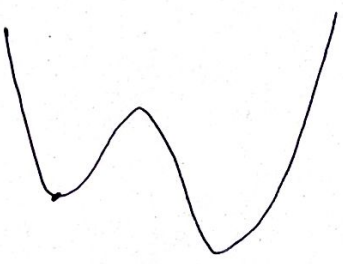
→ Easiest instance of VC is an n node graph with no edges.

Here V.C = ϕ.

we start at $C = V$ and reach at ϕ gradually deleting one node at a time.

→ 

star graph



The basic gradient descent was suffering from getting stuck at local minima. Now we will discuss some refined local search algorithms that use same type of neighbor selection, but include techniques for escaping local minima.

Metropolis Algorithm :- Metropolis gave the first idea for an improved local search algorithm, considering the problem of simulating the behaviour of a physical system according to the principle of statistical mechanics (SM)

→ As per the basic principle of SM, the probability of finding a physical system in a state with energy E is proportional to the Gibbs-Boltzmann function $e^{-E/(kT)}$ where $T > 0$ is the temperature and $k > 0$ is a constant. i.e. Probability $\propto \frac{1}{e^{E/(kT)}}$

Probability of a state $\propto \frac{1}{e^{E/(kT)}}$

→ for any temperature T, the function is a monotonically decreasing one with respect to E. i.e. if E is more probability is less
E is less probability is more

i.e. the physical system is more likely to be in a lower energy state than in a high energy state.

When T is small, probability for a low-energy state is significantly higher

T is large, high and low energy states have roughly same probabilities. i.e. the system is equally likely to be in any state.

M. Algo (Simulating of a system at a fixed T)

→ At all time, the simulation maintains a current state of the system and tries to produce a new reachable state by applying a perturbation.
→ Assume that there is a finite set C of reachable states.
→ In a single step, we generate a small random perturbation to the current state s and resulting a new state s'.
→ if $E(s') \leq E(s)$, update the current state to s'.
otherwise, update the current state to s' with probability $e^{-\Delta E/(kT)}$
otherwise leave s

Algorithm:-

Start with an initial solution S_0 and constants K and T.

In one step:

Let S be the current solution.

Let S' be choosen uniformly at random from the neighborhood of S ($N(S)$).

If $c(S') \leq c(S)$ then

Update $S \leftarrow S'$

Else

with a probability $e^{-(c(S') - c(S))/(KT)}$ (i.e. $e^{-\Delta E/(KT)}$.)

Update $S \leftarrow S'$

otherwise

Leave S unchanged.

endif

→ The algorithm is globally biased towards "downhill" steps but occasionally makes "uphill" steps to break out of local minima.

Theorem. Let $f_S(t)$ be the fraction of first t steps in which simulation is in state S. Then as t approaches ∞, with probability approaching 1, $\lim_{t \to \infty} f_S(t) = \frac{1}{Z} \cdot e^{-E(S)/(KT)}$.

where $Z = \sum_{S \in C} e^{-E(S)/(KT)}$.

i.e. the simulation spends roughly the right amount of time in each state, as per the Gibbs-Boltzmann eqn.

* The VC. instance with star graph, the metropolis algo. will quickly bounce out of the local min^m. even if x is deleted by adding x into the soln again with some +ve probability.

* But, for the first example of VC with n vertices and zero edges, where Gradient Descent solved without any problem, the M. Algo. may not exhibit as good as it considers both "downhill" as well as "uphill".

Simulated Annealing :- In order to fix the "flinching" issue arised in Metropolis algorithm, we can think of T as a one-D knob that control the extent of acceptable "uphill" movement.

→ T ∝ probability of accepting the "uphill" move.

T is large ⇒ Probability of accepting an uphill move is large

T is Small ⇒ uphill moves are almost never accepted.

↳ Metropolis algorithm behaves like a random walk. (too much time)

↳ " " " " Gradient-descent (local min")

→ However, neither of these extreme T values are effective to solve minimization problems in general as both random walk and gradient descent have several key issues.

→ In a physical system, if we heat a solid to a very high temperature (T is high), it cannot maintain a nice crystal structure (equilibrium). OR, if we suddenly freeze a molten solid, it too cannot maintain a perfect crystal structure.

→ So, scientists (Kirkpatrick et al) thought of using the Annealing method which is a process of guiding a material to a crystalline state (equilibrium) / low energy state).

→ In annealing, the material is cooled gradually from a high temp, allowing it to reach equilibrium. Thus escaping from local minima.

→ We can mimic the above process computationally, using a technique Simulated Annealing.

→ SA works by running the Metropolis algorithm while gradually decreasing the value of T over the course of execution. The way T is updated is called a cooling schedule, which is a function $\tau : \{1, 2, \dots\} \to R^+$.

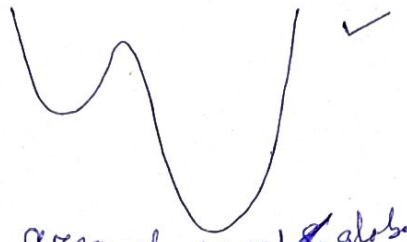
→ In iteration 'i' of the Metropolis algorithm, we use the temp. $T = \tau(i)$.

→ Simulated annealing allows for large changes in the solution in the early stages of its execution, when the temp. is high.

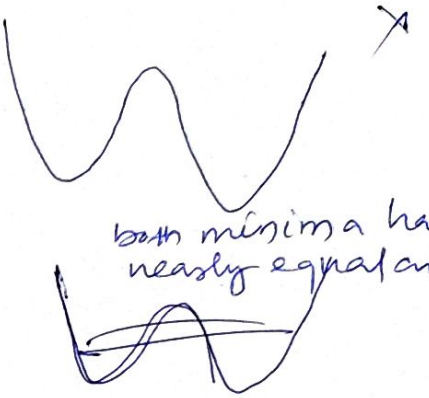
→ As the search proceeds, the temperature is lowered so that we will ~~get~~ less likely get "uphill" movement hence less likely to undo the progress made.

→ The simulated annealing method has no guarantee of finding the optimal solution.

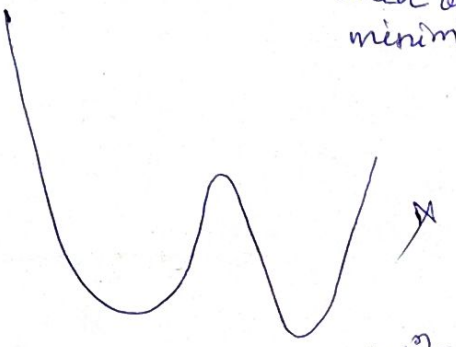
for example:-



area of local & global minima



both minima has nearly equal area.



no. optimal sol^n difficult.

Hopfield Neural Network :=

→ Here, no need to find the global optimum; any arbitrary local minimum will give a valid solution (acceptable).

Problem :- To find stable configurations in Hopfield Neural Networks.

→ Hopfield networks :- A simple model of an associative memory.

→ A Hopfield network can be viewed as an undirected graph $G_0 = (V, E)$ with integer-valued (+ve or -ve) weights we on each edge $e \in E$.

→ A configuration 'S' of the network is an assignment of the value -1 or +1 to each node $u \in V$, and we refer these values as the state s_u of node u.

→ The nodes represent the units of the neural network and they are trying to choose between two possible states (Yes (+1) or no (-1)). This choice is influenced by the incident edge and the neighbor vertex, as follows.

→ Each edge imposes a requirement on its endpoints. If (u, v) is an edge with -ve weight, then u and v want to have the same states.

→ If (u, v) is an edge with +ve weight, then they want to have opposite states.

→ The absolute value $|w_e|$ will indicate the strength of this requirement, and is called the absolute weight of e.

→ We may not always find a configuration that respects the requirements imposed by all edges.

For example, consider

→ So, instead of toying to achieve such a configuration, we can ask for something weaker (local minima)

→ We say that an edge $e=(u,v)$ is <u>good</u> with respect to a given configuration, if its requirement is satisfied by the states of its endpoints u and v.

i.e. if $w_e < 0 \quad s_u = s_v$

or, if $w_e > 0 \quad s_u = \overline{s_v}$

otherwise we call e is bad.

→ we can formulate the cond^n. to check for good edges as;- $w_e \cdot s_u \cdot s_v \leq 0 \Rightarrow e$ is good.

→ We say a node u is satisfied in a given configuration, if the total absolute weight of all good edges incident on u is at least as large as the total absolute weight of all bad edges incident on u.

$$W(\text{good edges of } u) \geq W(\text{bad edges of } u)$$

$$\sum_{e \in \text{good}} \underbrace{w_e s_u s_v}_{-ve} \geq \sum_{e=\text{bad}} \underbrace{w_e s_u s_v}_{+ve}$$

$$\Rightarrow \text{So, } \sum_{v:\, e=(u,v) \in E} w_e s_u s_v \leq 0.$$

→ A configuration is stable, if all nodes are satisfied.

<u>Why are say a configuration is stable when all nodes are satisfied?</u>

If we look at the n/w from the point of view of a node u, it has 2 choices, to take the state -1 or $+1$.

→ Every node want to respect as many edge requirements as possible.

For this, a natural condition is, to have good edges

i.e. $w_e < 0 \quad s_u = s_v$

$w_e > 0 \quad s_u \neq s_v$.

So, if the node u is satisfied, it achieves the natural cond^n and hence is more stable.

→ But, if the node u is not satisfied, that means it has more bad edge weight than good edge.

So, u can flip its state.

Then all good edges became bad and all bad edges became good. So, it became satisfied.

Analysis of state-flipping algo:-

Theorem:- The state-flipping algo terminates with a stable configuration after at most $W = \sum_e |w_e|$ iterations

Pf:- Consider a measure of progress of the algo as:-

$$\Phi(S) = \sum_{e \text{ is good}} |w_e|$$

[the algo. increases the good edges in every iteration]

$$0 \leq \Phi(S) \leq W$$

→ $\Phi(S)$ increases by at least 1 in every iteration.

→ Each iteration involve finding an unsatisfied node and flipping its state.

after we flip the state of a node u":-

- all good edges incident on u became bad
- all bad edges on u became good.
- all the other edges remain the same.

Before flipping, the weight of good edges is less as compared to the weights of bad edges on u.

After swapping, weight of good edges increases as compared to ~~bad~~ weight of bad edges.

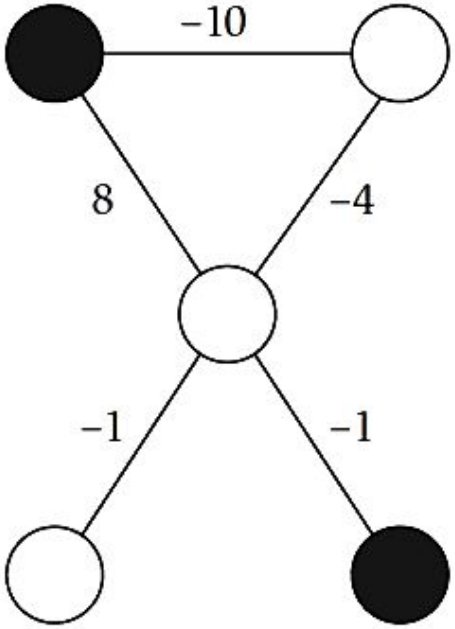
So, $\Phi(S') \geq \Phi(S) + 1$

→ As $\phi(S)$ increases in each iteration by at least 1 and the max^{m} possible value of $\phi(S)$ is W, ~~and~~ which is finite, the algo will terminate after W iterations of while loop. [time complexity $O(WN)$.

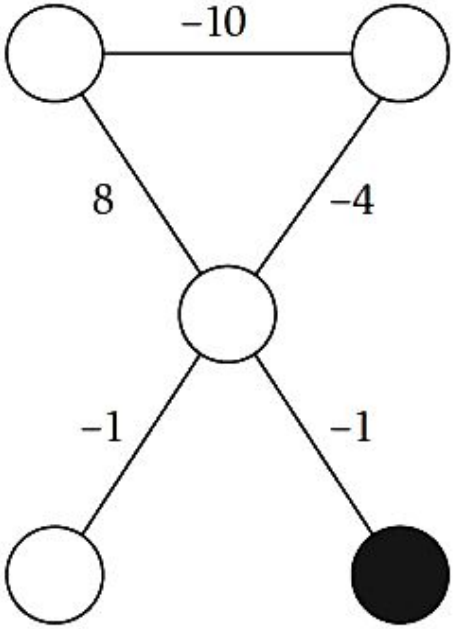
→ On termination, the configuration returned is stable as if it was not, the while loop would not have been terminated.

State-flipping algorithm :-

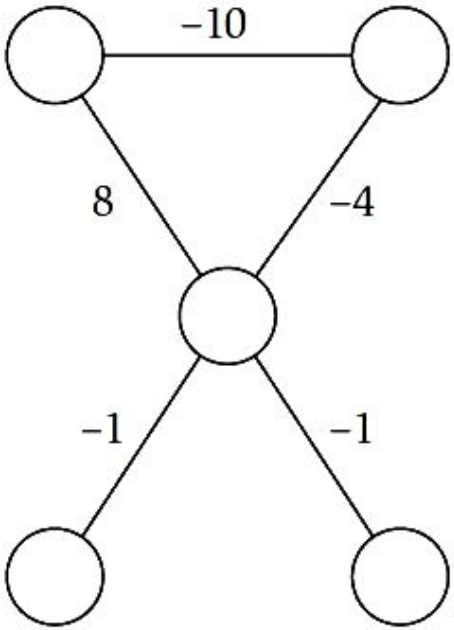
~~while the~~
Hopfield-flip (G, w)
  $S \leftarrow$ arbitrary configuration
  while the current configuration is not stable.
    $u \leftarrow$ unsatisfied node
    flip the state of u.
  end while.



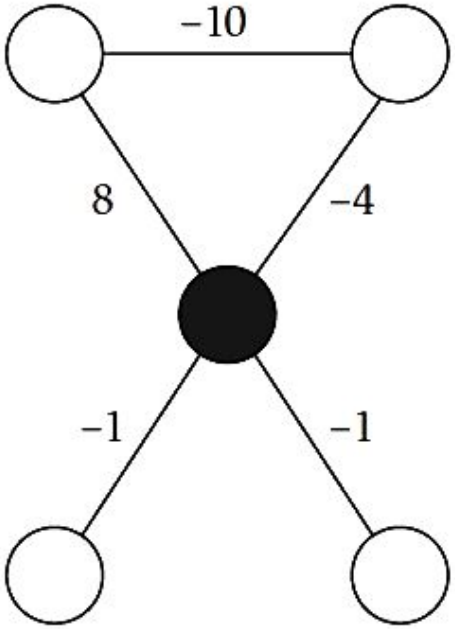
(a)



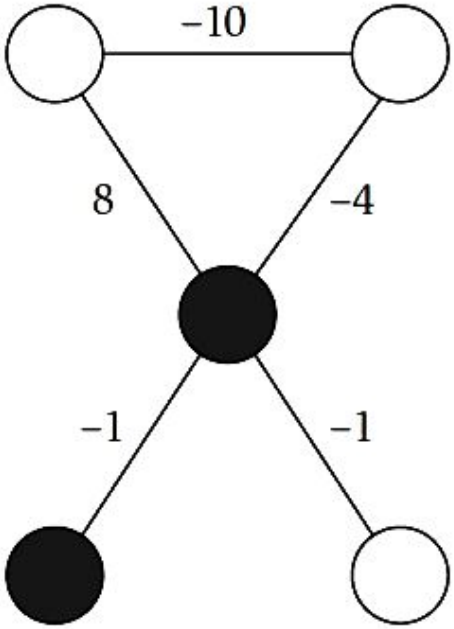
(b)



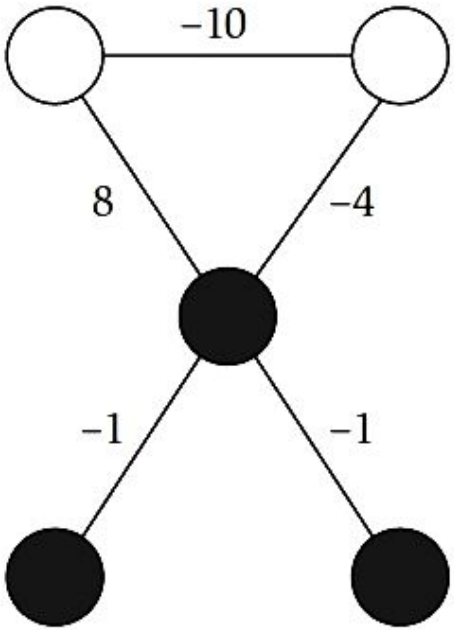
(c)



(d)



(e)



(f)