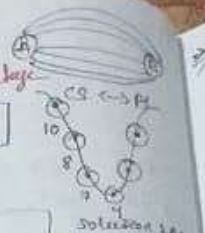
LOCAL SEARCH



- Search the space of all possible solutions in a sequential manner to find out the optimal solution.
- cannot guarantee the optimal solution.
- cannot say how much closer the sol is to optimal.

disadvantage

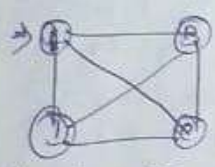
* no prediction for local optimal.

Given:-
- S = set of all possible solutions
  $S = \{S_1, S_2 \dots S_n\}$
- C(S) = cost function to compare two solutions
- Neighbours relation:- $S \leftrightarrow S'$
- Obj: solution with minimum cost

Vertex Covers :-

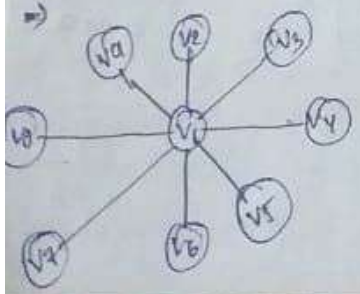
- G(V, E)
- Cost of VC set = size of vertex cover set
- Minimize the cost of VC set = find a VC set whose size is as small as possible.
- Neighbour Relation:- $S \leftrightarrow S'$
  If we can get S' from S either by deleting or adding a single element.

* Gradient Descent:
Start with S = V, then find a neighbouring sol S' and then replace S with S' if cost of S is strictly less than cost of S' (iff C(S') < C(S))



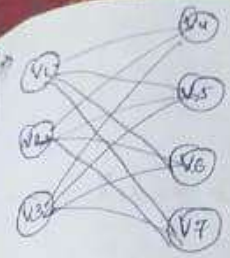
S = {A, B, C, D}
S' = S - A or S - B or S - C or S - D

S = V = {A to I} → S' = {B to I} (replace)
Now S'' = {C to I} (replace)
Finally S* = { } → optimal.



Start with S = V = {V1 to V9}
S' = {V2 to V9}

* (Here we cannot go further find a local solution thus here we get the sol as S' whose cost is but optimal has cost 1.)
* Stuck at local optimal solution



$S = \{v_1 \dots v_7\}$

$S' = \{v_2 \dots v_7\}$

$S^* = \{v_4 \dots v_9\}$

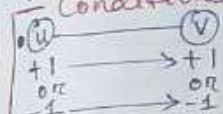
Best optimal is $\{v_1, v_2, v_3\}$

• $\{v_2, v_4, v_6, v_8\}$ • $\{v_1, v_3, v_5, v_7, v_9\}$ • $\{v_2, v_3, v_4, v_5, v_6, v_8, v_9\}$

Hopfield Neural Network :-

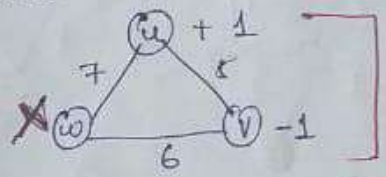
- Problem: to find a stable configuration in Hopfield neural network
- Hopfield Neural Network: - has a undirected graph $G(V, E)$ where edges can have weight $w_e < 0$ or $w_e > 0$. (Associative memory network)
- Configuration: to assign value to a vertex (+1/-1) also called state of a node u
  If $s_u = +1 \rightarrow$ on else $s_u = -1 \rightarrow$ off

Conditions :-

- 
  and weight $uv < 0 \Rightarrow$ state of u = state of v ($s_u = s_v$)

- 
  and weight $uv > 0 \Rightarrow$ state of u is opposite of state of v (s_u is opposite of s_v)

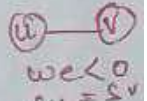
But everytime it is not possible to satisfy these conditions.



Weaker Condition :-

- An edge will be good if $w_e s_u s_v < 0$
  else it is a bad edge.
- Satisfied vertex/node: if weight of incident good edges $\geq$ weight of incident bad edges

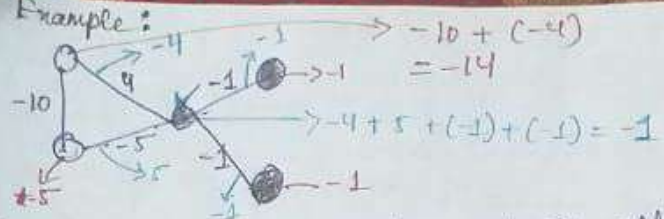
$$\sum_{e(u,v)} w_e s_v s_u \leq 0$$



$w_e < 0$

$s_u = s_v$

Example:



GOAL: Find a configuration where all the nodes are satisfied, if one exists

Algorithm:-

While the current configuration is not stable

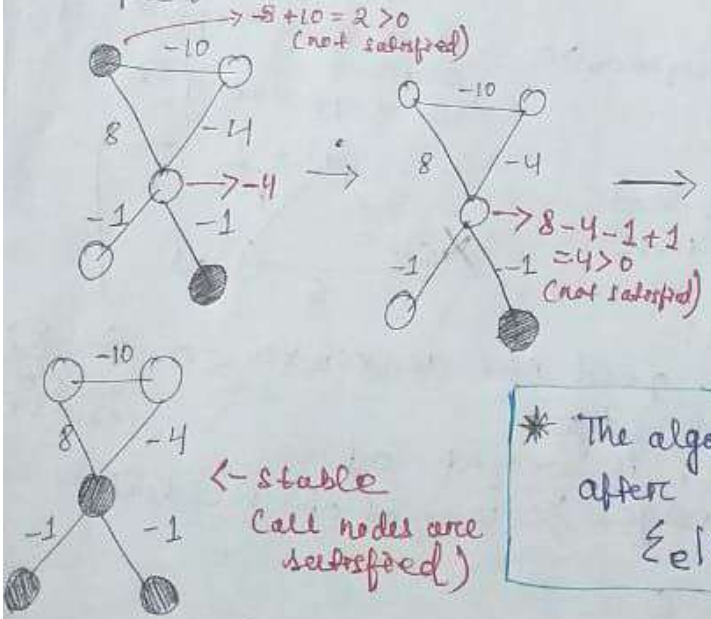
There must be an unsatisfied node

Choose an unsatisfied node u

Flip the state of u

End while.

Example:-



<- stable
(All nodes are satisfied)

* The algorithm terminates after at most $\leq |V| + |E|$ steps.

Centre Selection Problem :- (Continue)

minimum value of r : $r_{min} \geq 0 = r_0$

maximum value of r : maximum distance between any two sites. $= r_{max}$

* $r_{opt} = \frac{r_0 + r_{max}}{2}$ — First check for this then decrease r_{opt} and repeat the algorithm.

[Binary Search concept]

at most K centers found then opt sol lies in LHS

If K centers not found then opt sol found in RHS

* Greedy approach is a 2 approximation algorithm.

Algo :-

Assume $K \leq |S|$ (else define $C = S$)

Select any site s and let $C = \{s\}$

while $|C| < K$

  Select a site $s \in S$ that maximizes $dist(s, C)$

  Add site s to C

End while

Return C as the selected set of sites.

RANDOMIZED ALGORITHM

Algorithm that behaves randomly. Approach is precisely internal to the algorithm and does not require any assumptions about the nature of input.

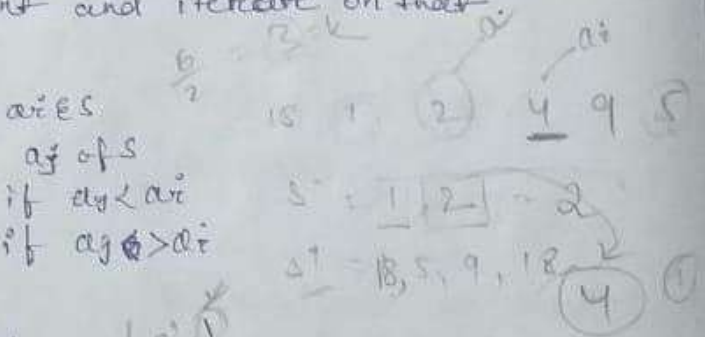
Median Finding:
Divide and Conquer works well along with randomization. In each case the divide step is performed using randomization.

Problem: Given a set of n numbers $S = \{a_1, a_2 \ldots a_n\}$. Their median is the number that would be in the middle position if we were to sort them. If n is even there is no such middle position.

* The median of $S = \{a_1, a_2 \ldots a_n\}$ is equal to the kth largest element in S, where $k = (n+1)/2$ if n is odd
  $k = n/2$; if n is even
* We will assume for the sake of simplicity that all numbers are distinct. Without this assumption, the problem becomes notationally more complicated.
* Time complexity is $O(n \log n)$ if we sort first.

Generic Algorithm based on splitter:
Select(S, k): returns kth largest element
We choose an element $a_i \in S$ as the splitter and form
$S^- = \{a_j : a_j < a_i\}$ and $S^+ \{a_j : a_j > a_i\}$
Then we can determine which of S^- or S^+ contains the kth largest element and iterate on that.

Algorithm:-
Select (S, k)
Choose a splitter $a_i \in S$
For each element a_j of S
  put a_j in S^- if $a_j < a_i$
  put a_j in S^+ if $a_j > a_i$
End for
If $|S^-| = k-1$ then
  The splitter a_i was in fact the desired answer.
Else if $|S^-| > k$ then
  the kth largest element lies in S^-
  Recursively call Select(S^-, k)



Else suppose $|S^-| = \ell < k-1$
The kth largest element lies in S^+
Recursively call Select-$(S^+, k-1-\ell)$
End if

* choosing a good splitter :-
The splitter should significantly reduce the size of the set being considered, so that we don't keep making passes through large sets of number of times. so a good splitter should produce sets S^- & S^+ that are approximately equal in size.

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

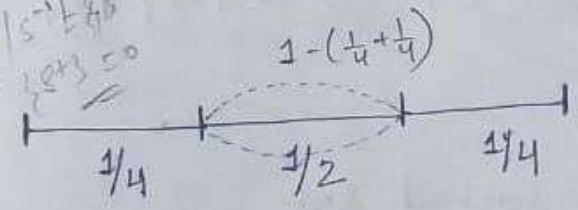
We must beware of off-center / bad splitter : If we choose the minimum element as the splitter, then we may end up with a set in the recursive call that is only one element smaller than before

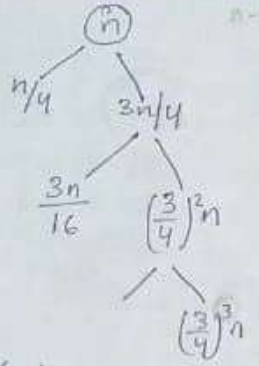
$$T(n) \leq T(n-1) + c \Rightarrow O(n^2)$$

Any 'well-centered' element can serve as a good splitter : If we had a way to choose splitters in such that there were at least εn elements both larger and smaller than a, for $\varepsilon > 0$, then size of the sets in the recursive call would shrink by factor $(1-\varepsilon)$ each time

$$T(n) \leq T((1-\varepsilon)n) + cn$$
$$= T((1-\varepsilon)^2 n) + (1-\varepsilon)cn + cn$$
$$T(n) = cn + (1-\varepsilon)cn + (1-\varepsilon)^2 cn + (1-\varepsilon)^3 cn + \cdots$$
$$= cn(1 + (1-\varepsilon) + (1-\varepsilon)^2 + \cdots)$$
$$= cn\left(\frac{a}{1-r}\right) = cn\left(\frac{1}{1-1+\varepsilon}\right) = \frac{cn \times 1}{\varepsilon}$$
$$\Rightarrow O(n)$$

Let :
$\varepsilon = 1/4$





after jth phase : $\left(\frac{3}{4}\right)^j cn$

Now,
Let x be random variable
X = no. of steps to execute the algorithm.
$= X_0 + X_1 + \dots$
X_j = no. of steps required in jth time phase

$$E(X) = \sum_j E[X_j] = \sum 2cn\left(\frac{3}{4}\right)^j = 2cn \sum_j \left(\frac{3}{4}\right)^j$$
$$= 2cn \times 4 = 8cn$$

$1 + \frac{3}{4} + \left(\frac{3}{4}\right)^2 + \dots$
$\frac{a}{1-r} = \frac{1}{1/4} = 4$

QUICK SORT :-

Average Time complexity : $O(n \log n)$
Worst case Time complexity : $O(n^2)$
To sort the list $S = \{a_1, a_2 \dots a_n\}$
we choose a splitter element $a_i \in S$ and the divide S as : S^- if $\{a_j : a_j < a_i\}$, S^+ if $\{a_j : a_j > a_i\}$
* If a_i is a good splitter or well centered splitter S is divided into two parts (equal halfs)
$T(n) = 2T(n/2) + cn \Rightarrow O(n \log n)$
* If a_i is a bad splitter or offcenter splitter
$T(n) = T(n-1) + cn \Rightarrow O(n^2)$
Condition : $|S^-| \geq |S|/4$ and $|S^+| \geq |S|/4$
Algorithm :-
Quick Sort (S)
IF $|S| \leq 3$ then
sorts
output the sorted list
Else
choose a splitter $a_i \in S$ uniformly at random

For each element a_j of S
put a_j in S^- if $a_j < a_i$
put a_i in S^+ if $a_j > a_i$
End for

Recursively call QuickSort(S^-) and QuickSort(S^+)
Output the sorted set S^- then a_i then sorted set S^+
End if

* Analysis of quicksort will now closely follow the procedure of median finding. The crucial definition is that of a central splitter - one that divides set so that each side contains at least a quarter of elements.

Algorithm :-

Modified QuickSort(S):

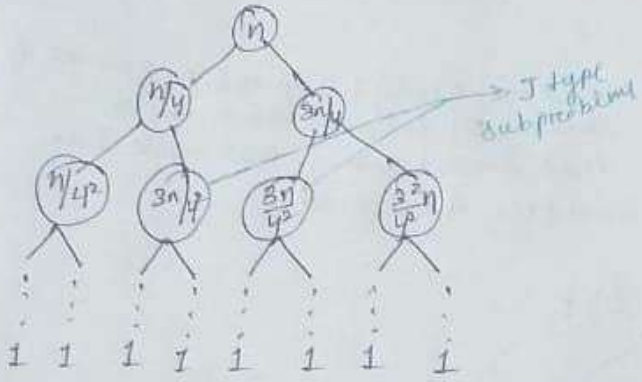
If $|S| \leq 3$ then
    sort S
    Output the sorted list
End if

Else
    while no central splitter has been found
        choose a splitter $a_i \in S$ uniformly at random
        For each element a_j of S
            put a_j in S^- if $a_j < a_i$
            put a_j in S^+ if $a_j > a_i$
        End for
        If $|S^-| \geq |S|/4$ and $|S^+| \geq |S|/4$ then
            a_i is a central splitter
        End if
    End while
    Recursively call quicksort(S^-) and QuickSort(S^+)
    Output the sorted set S^-, then a_i then the
    sorted set S^+

End if

* The expected running time for the algorithm on a set S, excluding the time spent on recursive call is $O(|S|)$.

* Define a subproblem type J: if the size of the subproblem lies between $\left(\frac{3}{4}\right)^{j+1} n < \text{size} < \left(\frac{3}{4}\right)^{j} n$



Expected time complexity of a type J subproblem is $O\left(\left(\frac{3}{4}\right)^j n\right)$

Let x = no of subproblem of type J

$$x \times \left(\frac{3}{4}\right)^{j+1} n = n \Rightarrow x = \left(\frac{4}{3}\right)^{j+1}$$

Expected time required by type J subproblem

$$= \left(\frac{4}{3}\right)^{j+1} \times \left(\frac{3}{4}\right)^{j} n \simeq O(n)$$

Total no. of different types of subproblem

$$= \log\left(\frac{3n}{4}\right) \simeq \log n$$

Thus Expected time complexity : $O(n \log n)$