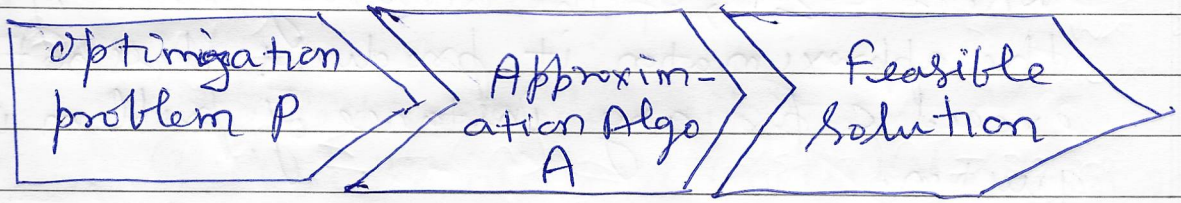
Approximation Algo. for NP-Hard Problems

Definition:- Given an optimization problem P, an algorithm A is said to be an approximation algo for P, if for any given instance I, it returns an approximate solution, that is a feasible solution.

- It takes an optimization problems because most of the optimization problems are often associated with NP-hard problems.



Properties of Approximation Algorithms:

(1) Guaranteed to run in polynomial time.

(2) Guaranteed to get solution which is close to the optimal solution (near optimal)

Some important terms:

(1) Accuracy Ratio : $r(S_a)$

$S_a \rightarrow$ Approximate solution

$f(S_a) \rightarrow$ value of objective function for solution given by approximation algo

$f(S^*) \rightarrow$ Exact solution of the problem

$$r(s_a) = \frac{f(s_a)}{f(s^*)}$$ for minimizing the objective function.

$$r(s_a) = \frac{f(s^*)}{f(s_a)}$$ for maximizing the objective function

(ii) performance ratio:

- A polynomial-time approximation algo. is said to be a c-approximation algo, where $c \geq 1$, if the accuracy ratio of the approximation it produces does not exceed c for any instance of problem in question.

$$r(s_a) \leq c$$

Approximation Algorithm for TSP:

Idea - The idea behind the TSP is to find to optimal tour when travelling between many cities. TSP is the well known optimization problem.

We can get solution for small instances in reasonable time but for large instances, we cannot get the solution in reasonable time.

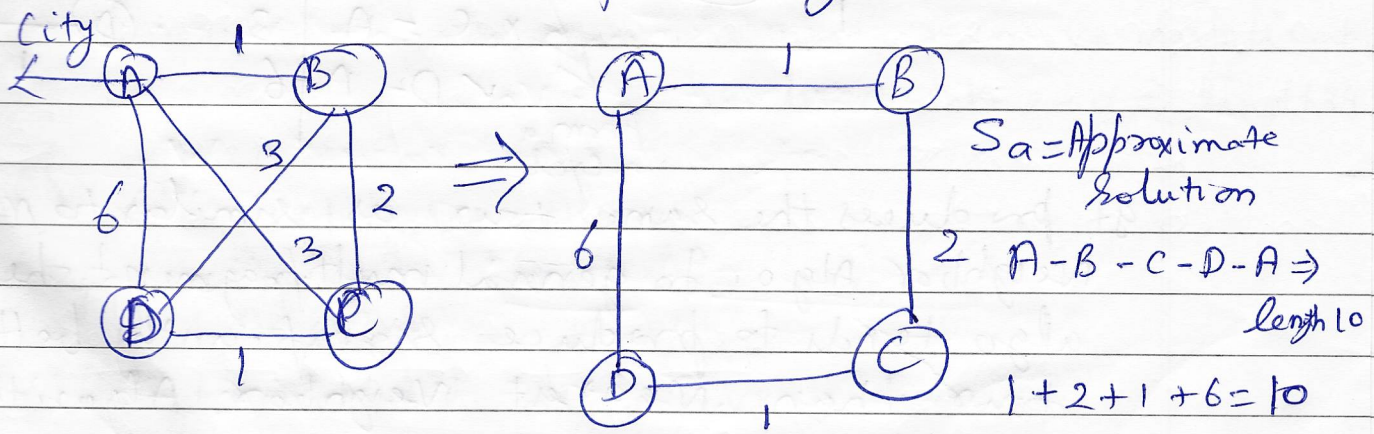
Approximation Algorithms

- Greedy Approach
  - Nearest-neighbor Algo
  - Multifragment-heuristic Algo
- Minimum Spanning-Tree-Based Algo.
  - Twice-around-the-tree Algo
  - Christofides Algo

① Nearest neighbor Algo:

Steps:

1. Select a random city
2. Find the nearest unvisited city & go there.
3. Are there any unvisited cities left?
   If yes, repeat step 2.
4. Return to the first city.



S_a = Approximate Solution

A-B-C-D-A ⇒ length 10

1+2+1+6 = 10

→ Conclusion:

Tour S_a is 25% longer than optimal tour S^*.

S^*: Optimal Solution

A-B-D-C-A ⇒ length 8

1+3+1+3 ⇒ 8

$$r(S_a) = \frac{f(S_a)}{f(S^*)} = \frac{10}{8} = 1.25$$

So Nearest neighbor Alg is simple to use but it has drawbacks even it ensures the small distance to the neighbor but return back to the starting point that may not be the shortest path.

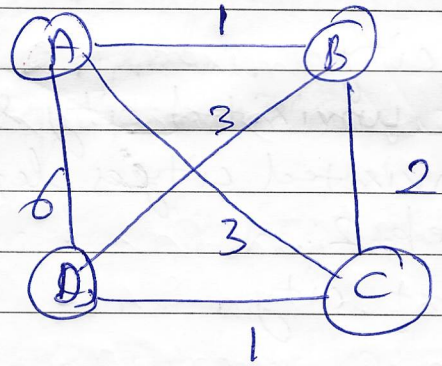
(2) Multifragment - heuristic Algo:

Steps:

(1) Sort the edges in increasing order of their weights.

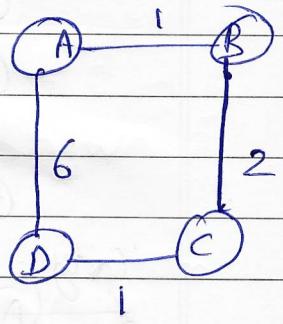
(2) Repeat this step n times, where n is the number of cities in the instance being solved: add the next edge on the sorted edge list to the set of tour edges, provided this addition does not create a vertex of degree 3 or a cycle of length less than n; otherwise, skip the edge.

(3) Return the set of tour edges.



Node	Edge wt
✓ A→B	1
✓ C-D	1
✓ B-C	2 ⇒
✗ B-D	3
✗ C-A	3
✓ D-A	6

Form a cycle



It produces the same tour as similar to nearest neighbor Algo. In general multifragment-heuristic algo tends to produce significantly better tour than Nearest Neighbor Algorithms.

Minimum-Spanning-Tree-Based Algo:

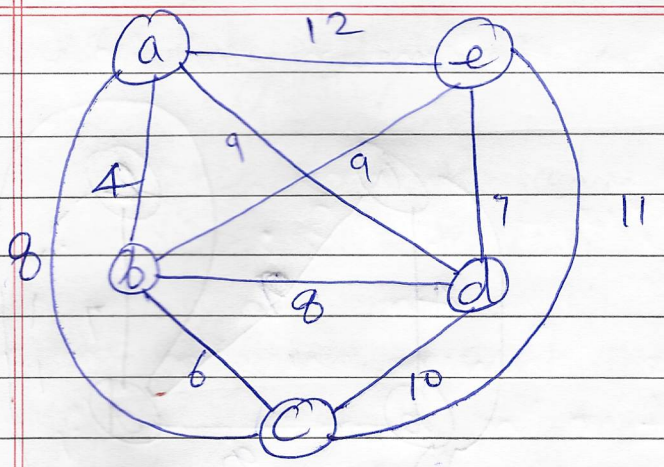
(i) Twice-around-the-tree Algo.

Steps: (1) Construct a MST

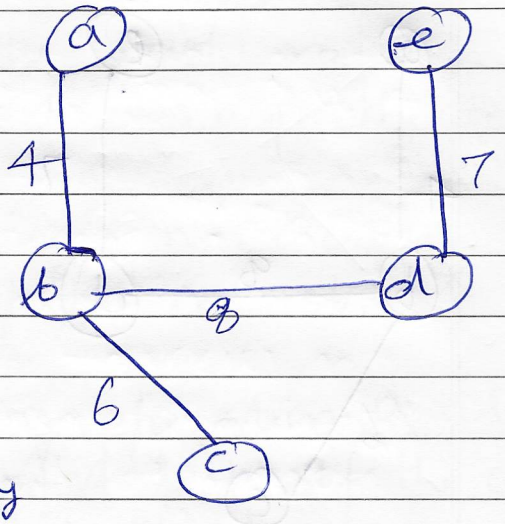
(2) Let the root be an arbitrary vertex.

(3) Traverse all the vertices by depth-first search, record the sequence of vertices (Both visited and unvisited)

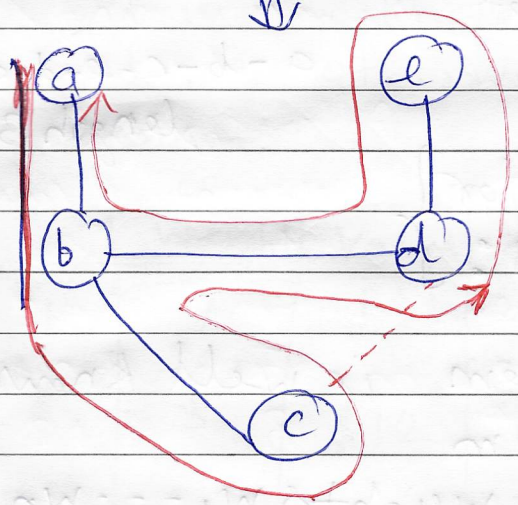
(4) Use shortcut strategy to generate a feasible tour



MST



Traverse all the vertices by depth-first search



Use shortcut strategy to generate a feasible tour.

Record the visited nodes.

a-b-c-~~b~~-d-e-~~d~~-~~b~~-a

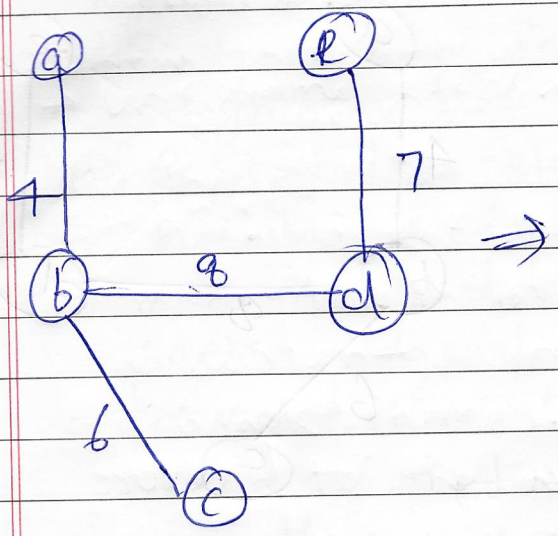
remove duplicates

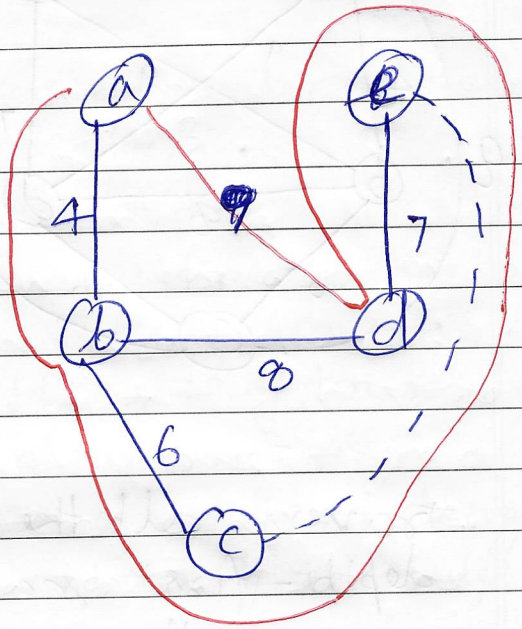
a-b-c-d-e-a ⇒ length 39

(ii) Christofides Algo:

1. Find MST
2. Find odd degree vertices
3. minimum weight matching
4. Find Euler cycle path
5. Find TSP cycle path

MST





Odd degree vertices - a, b, c, e

a-b-c-e-d-a→
length 37

Approximation algo for Knapsack problems:

- Knapsack problem is well known NP-hard problem
- n items of known weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$.
- Knapsack of weight capicity W.
- Problem objective: find the most valuable subset of the items that fits into the knapsack.

Two versions of Knapsack:

(1) 0-1 Knapsack problem: (Discrete)
↳ items are indivisible (either take an item or not)

(2) Fractional Knapsack problem: (Continuous)
↳ items are divisible (can take any fraction of item)

Greedy algo for the discreate Knapsack problem

Steps: (1) Compute value to the weight ratio
$r_i = v_i / w_i$, $i = 1, \dots, n$ for the items given.

(2) Sort the items in non increasing order of the ratios computed in Step 1.

(3) Repeat the following operation until no item is left in the sorted list: if the current item on the list fits into the knapsack, place it in the knapsack and proceed to the next item; otherwise just proceed to the next item.

Item	Weight	Value	v_i/w_i
1	7	$42	6
2	3	$12	4
3	4	$40	10
4	5	$25	5

Step 2 - Arrange in non-increasing order

Item	Weight	Value	v_i/w_i
3	4	$40	10
1	7	$42	6
4	5	$25	5
2	3	$12	4

If max capicity of Bag = 10
Load Item 3 into Bag = 10 - 4 = 6
Load next item 4 = 6 - 5 = 1
So total weight = 9 / 10
Profit = $40 + $25 ⇒ $65

Greedy Algo. for the continuous knapsack problem.

⇒ use same steps except step 3.

Step 3: Repeat the following operation until no item is left in the sorted list: if the current item on the list fits into the knapsack in its entirely, take it and proceed to the next item; otherwise, take its largest fraction to fill the knapsack to its full capacity and stop.

$W = 10$

Load first item in Bag.
⇒ $10 - 4 = 6$

Load second item in Bag.
⇒ Select 6/7 of the item as available capacity is 6.

Then value $= 42 * \frac{6}{7} = 36$

After sorting.

item	weight	value	v/w
1	4	$40	10
2	7	$42	6
3	5	$25	5
4	3	$12	4

Total weight = 10
Profit = $40 + $36 = $76

Conclusion:

- Knapsack problems can be solved using Greedy as well as Approximation schemes. If we use greedy then it gives optimal solution for continuous knapsack but not for discrete.

- So we move to the approximation scheme to solve the Discrete knapsack problem.

Approximation scheme for knapsack problem: (Discrete)

- This scheme / Algo generates all subsets of k items or less, and for each one that fits into the knapsack it adds the remaining items as the greedy algorithm would do.

- The subset of the highest value obtained in this fashion is returned as the algorithm's o/p.

item	weight	value	vi/wi
1	4	$40	10
2	7	$42	6
3	5	$25	5
4	1	$4	4

K=2, W=10 (start with nullset to max. of 2 elements.)

W	Subset	Possible added items	Profit
4+5+1	{0}	1, 3, 4	$69
4+5+1	{1}	3, 4	$69
7+1	{2}	4	$46
5+4+1	{3}	1, 4	$69
1+4+5	{4}	1, 3	$69
11 > W	{1,2}	Not Feasible	
9+1	{1,3}	4	$69
5+5	{1,4}	3	$69
12 > W	{2,3}	Not Feasible	
8	{2,4}		$46
6	{3,4}	1	$69

Profit with $69 are the feasible solutions.
From these feasible solutions, we can easily select the optimal solutions, that is the beauty of approximation scheme.

Final conclusion:

Algo. efficiency is in $O(K n^{K+1})$

optimal Solution is $\{1, 3, 4\}$
$W = 10$, Profit = $69

Conclusion:

Brute force Algo:
1. Guaranteed to find optimal Solution
2. No guarantees on running time.

Heuristics:
1. Guaranteed to run in polynomial time.
2. No guarantees on quality of Solution

Approximation algorithms:
1. Guaranteed to run in polynomial time
2. Guaranteed to get a solution which is close to the optimal solution (a.k.a near optimal)