Approximation algorithms

We are in the business of designing efficient algorithms.

Over the last few weeks we are Struggling or going through a bad Phase in our business.

We can try to give our algorithm a little freedom

Instead of demanding that the algorithm finds the best possible (optimal) solution, we will allow the algorithm to find a solution that is within a certain range of the optimal solution.

As usual we demand: algorithm should run in Polynomial time.

Approximation Algorithms

optimizations

Many Problems of Practical Significance are NP-hard

we don't know how to find an optimal solution in Polynomial time.

However we Still want a Soln that is as good as possible even if it is not the optimal answer.

In Practice, near-optimal solutions are often good enough.

We call an algorithm that returns near-optimal solutions an approximation algorithm

We study Polynomial time approximation algorithms for several NP-hard optimization problems.

Key Question:

How to argue/prove that an algorithm finds near-optimal solution to a Problem?

Approximation Quality

- Constant factor approximation
- Polynomial time approximation schemes

Terminology

Assume that all solns have positive cost

Let Π be an optimization problem (say minimization)

we say that an algorithm solving Π has an approximation ration $\rho(n)$ if for any input of size n, the cost C of the soln produced by the algorithm is within a factor of $\rho(n)$ of the cost C^* of an optimal solution.

ie, $$\frac{C}{C^*} \leq \rho(n)$$

[called $\rho(n)$ - approximating algorithm]

ie, $$0 < C^* \leq C$$

Similarly we can define, for maximization problems

$$\frac{C^*}{C} \leq \rho(n)$$

ie., $$0 < C \leq C^*$$

If $\rho(n)$ is constant then we call the algorithm as constant factor approximation algorithm.

1-approximation algorithm produces an optimal soln and an approximation algorithm with a large approximation ratio may return a soln that is much worse than optimal.

* When designing an approximation algorithm for an NP-hard optimization problem, we need to establish the approximation guarantee. That is the cost of the solution produced by the algorithm needs to be compared with the cost of an optimal solution.

* But we know that computing the cost of an optimal solution is a difficult task.

Q: How do we establish the approximation guarantee?

we will demonstrate with an example.

VERTEX COVER PROBLEM

Suppose A is a factor 2-approximation algorithm for Vertex Cover Problem and A returns Vertex Cover with 100 vertices on input a graph G.

Q1: What is the maximum number of vertices in Optimal Solution ?

Q2: What is the minimum number of vertices in Optimal Solution ?

Problem 2 :

Same as above except replace VertexCover Problem with Clique Problem.

Polynomial time approximation schemes [PTAS]

Time vs Quality (trade-off)

ie, The running time depends on the quality of solution.

The better guarantee you demand, w.r.to quality the more running time you will have to put into that algorithm.

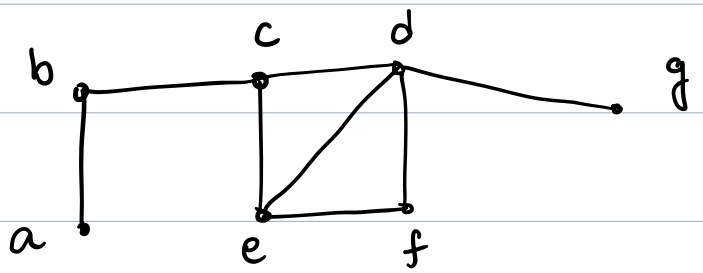
Vertex cover Problem

Recap: A **vertex cover** of an **undirected graph** G, is a **subset** $X \subseteq V(G)$ such that if uv is an edge in G, then either $u \in X$ or $v \in X$ (or both).

The size of a **vertex cover** is the **number of vertices** in it.

The vertex cover problem is to **find a vertex cover** of **minimum size** in a given **undirected graph**.

Example:



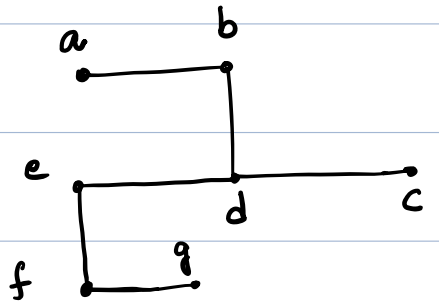
$X = \{b, d, e\}$ is a vertex cover of size 3.

Algorithm

```
APPROX-VC(G)
1   X = ϕ
2   E' = E(G)
3   while E' ≠ ϕ
4       let uv be an arbitrary edge of E'
5       X = X ∪ {u, v}
6       remove from E' every edge incident on either u or v
7   return X
```

Running time : $O(V+E)$ using adjacency lists to represent E'.

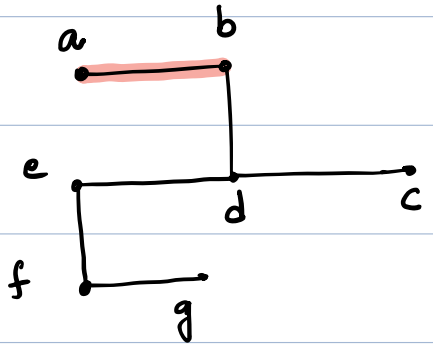
Example:



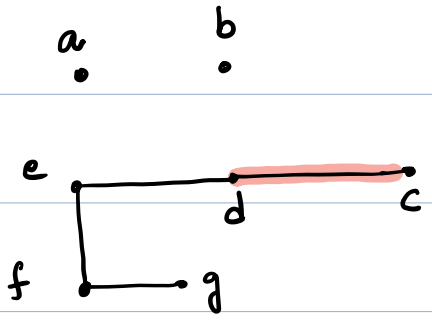
Graph G

Let us run APPROX-VC on the above graph

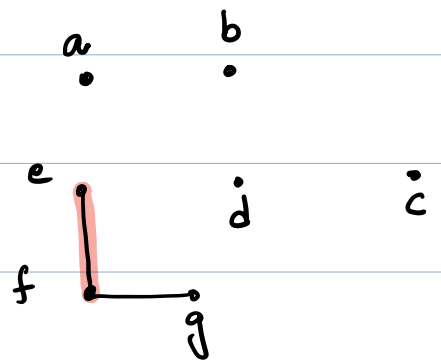
$X = \phi$



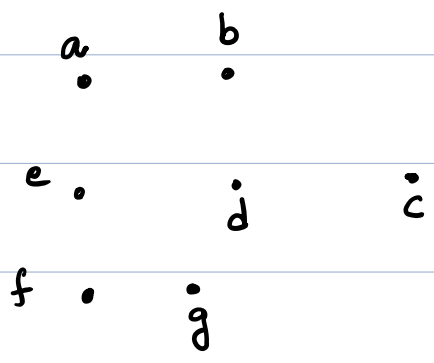
$X = \{a, b\}$



$X = \{a, b, c, d\}$



$X = \{a, b, c, d, e, f\}$



Soln returned by algo is $X = \{a, b, c, d, e, f\}$

Claim: APPROX-VC is a Polynomial-time 2-approximation algorithm.

Proof: Clearly the set of vertices returned by the algorithm is a vertexcover.

Let F denote the set of edges that line 4 of algorithm Picked.

In order to cover the edges in F, any optimal vertex cover X^* must include at least one endPoint of each edge in F.

No two edges in F share an endpoint, since once an edge is Picked in line 4, all other edges that are incident on its endpoints are deleted from E' in line 6.

∴ No two edges in F are covered by the same vertex from X^*, and we get

$$|X^*| \geq |F| \quad \text{— (1)}$$

Each execution of line 4 pick an edge for which neither endpoints is already in X, giving an upper bound on X.

$$|X| = 2|F| \quad — (2)$$

From (1) & (2)

$$|X| = 2|F| \leq 2|X^*|$$

ie, algorithm return a vertex cover X whose size is guaranteed to be no more than twice the size of an optimal vertex cover.

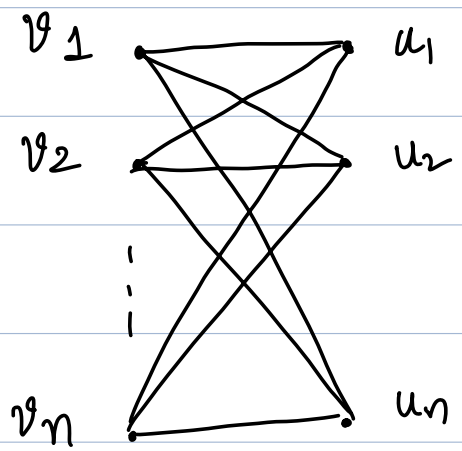
Note: under Unique games Conjecture, this factor of 2 for the VertexCover Problem is the best possible one.

Q: Can the approximation guarantee of above algorithm be improved by a better analysis?

Ans: No,

We can show that 2-approximation is a tight bound by a tight example.

Consider the graph $K_{n,n}$ (Complete bipartite graph)



$K_{n,n}$

The 2-approximation algorithm selects n edges hence returns a Vertex cover of size $2n$.

But clearly the optimal solution $= n$

Load Balancing Problem

Given a set of m identical machines $M_1, \dots M_m$ and a set of n jobs, each job j has a processing time t_j. A machine can process at most one job at a time.

We seek to assign each job to one of the machines so that loads placed on all machines are as "balanced" as possible.

More precisely, in any assignment of jobs to machines, we can let $A(i)$ denote the set of jobs assigned to machine M_i. Then the total time machine M_i needs to work for a total time

$$T_i = \sum_{j \in A(i)} t_j$$

We call T_i as the load on machine M_i.

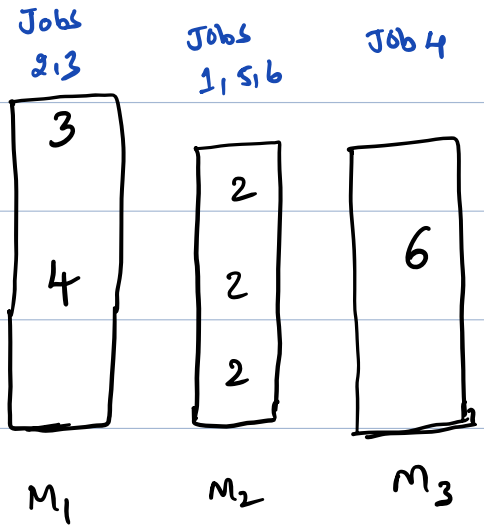
We seek to minimize a quantity known as "makespan" (T). which is the maximum load on any machine. $T = \max_i T_i$.

Remark: Finding an assignment of minimum makespan is NP-hard.

Example:

	M_1	M_2	M_3

Jobs:	1	2	3	4	5	6
Processing time:	2	3	4	6	2	2



$T_1 = 7$, $T_2 = 6$ $T_3 = 6$

Makespan $T = 7$

[Warmup]

If we have fewer than m jobs then can we solve load balancing efficiently?

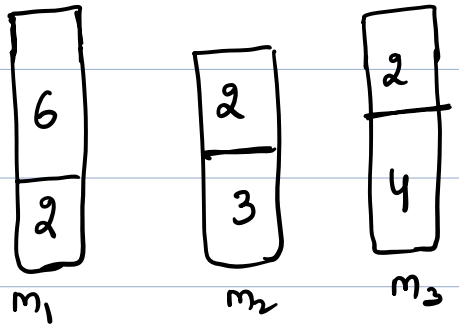
Algorithm

Idea: [Greedy Algorithm]

The Algorithm makes one pass through the jobs in any order, when it comes to job j, it assigns j to the machine whose load is smallest so far.

```
Greedy-Balance:
Start with no jobs assigned
Set T_i = 0 and A(i) = ∅ for all machines M_i
For j = 1, ..., n
  Let M_i be a machine that achieves the minimum min_k T_k
  Assign job j to machine M_i
  Set A(i) ← A(i) ∪ {j}
  Set T_i ← T_i + t_j
EndFor
```

Let's run the above algorithm one our previous example with the sequence 2, 3, 4, 6, 2, 2



Makespan = 8

↓

Clearly not optimal.

let T denote the makespan obtained by the algorithm.

T^* is the minimum possible makespan (ie, optimal makespan)

Observations:

① $$T^* \geq \frac{1}{m} \sum_j t_j$$

ie, one of the m machines must to do at least a $\frac{1}{m}$ fraction of the total work.

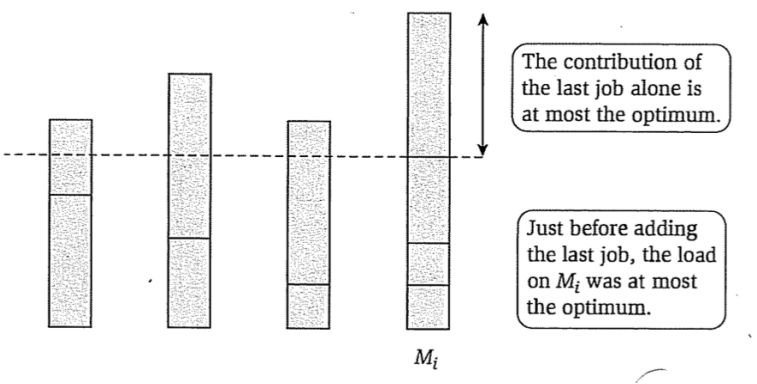
② $$T^* \geq \max_j t_j$$ [Trivial]

Claim: Algorithm Greedy-Balance Produces an assignment of jobs to machines with makespan $T \leq 2T^*$

Proof Idea:

Let M_i be the machine that attains the maximum load T in our assignment.

Let j be the last job assigned to M_i



When we assigned job j to M_i, the machine M_i had the smallest load of any machine.

$\therefore$ Load of machine M_i before assignment of job j was $T_i - t_j$

Also, at that moment every machine had load at least $T_i - t_j$.

$$\therefore \quad \sum_k T_k \geq m(T_i - t_j)$$

$$T_i - t_j \leq \frac{1}{m} \sum_k T_k$$

$$= \frac{1}{m} \sum_j t_j$$

$$= T^* \quad \text{(from observation 1)}$$

ie, $T_i - t_j \leq T^*$ — (A)

Now we bound the remaining part of the load on M_i (ie, t_j).

From observation (2), we have $t_j \leq T^*$. — (B)

From (A) & (B)

$$T_i = (T_i - t_j) + t_j$$

$$\leq T^* + T^* = 2T^*.$$

Since our makespan $T = T_i \leq 2T^*$.

Q: Is our analysis tight.

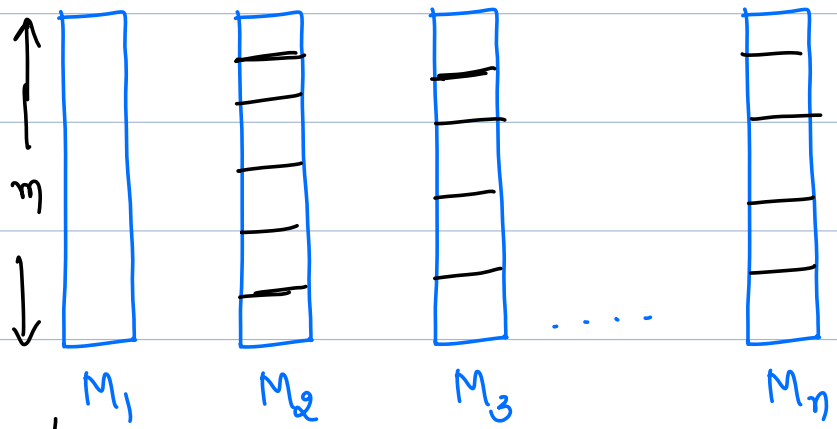
YES, see the below example.

Eg: m – machines

$m(m-1)$ jobs of length 1

one job of length m

Optimal soln:

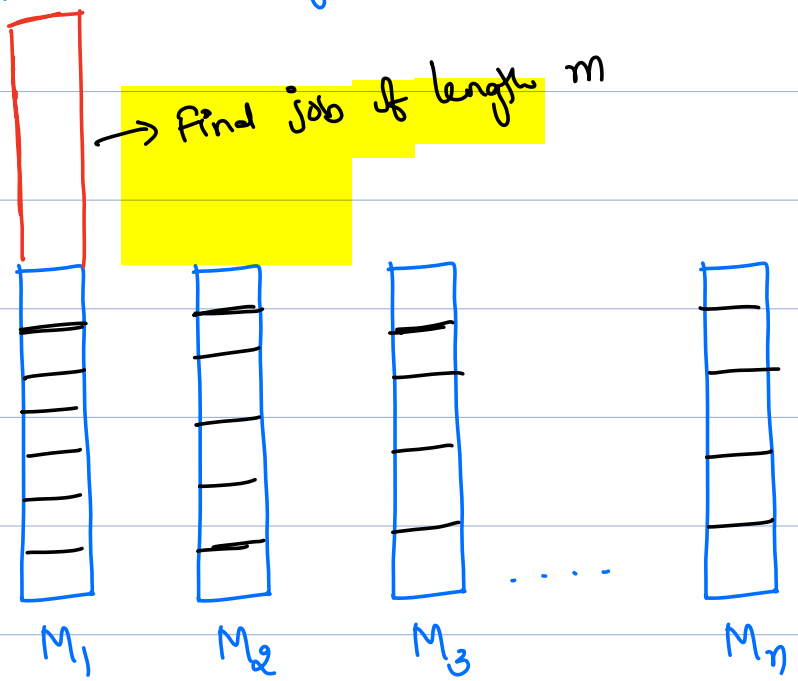


Assigned one job of length m

$(m-1)$ machines each assigned m jobs of length 1

Makespan = m

Output of the Algorithm:



Makespan = $(m-1) + m = 2m-1$

Improved Approximation Algorithm

In the previous algorithm, we may encounter the following bad case.

We spread everything out very evenly across the machines and then one last, giant, unfortunate job arrived.

Intuitively, it looks like it would help to get the largest jobs arranged nicely first, with the idea that later, small jobs can only do so much damage.

Algorithm

Sorted-Balance:
Start with no jobs assigned
Set $T_i = 0$ and $A(i) = \emptyset$ for all machines M_i
Sort jobs in decreasing order of processing times t_j
Assume that $t_1 \geq t_2 \geq \ldots \geq t_n$
For $j = 1, \ldots, n$
  Let M_i be the machine that achieves the minimum $\min_k T_k$
  Assign job j to machine M_i
  Set $A(i) \leftarrow A(i) \cup \{j\}$
  Set $T_i \leftarrow T_i + t_j$
EndFor

Analysis: Recap: We have m machines

if we have at most m jobs then the above algorithm solution will clearly be optimal.

Observation_3: If there are more than m jobs then

$$T^* \geq 2\, t_{m+1}$$

Proof: look at the first $m+1$ jobs in the sorted order.

they each take $\geq t_{m+1}$ time.

There are $m+1$ jobs & m machines. So there must be a machine that gets assigned two of these jobs. This machine will have Processing time $\geq 2\, t_{m+1}$.

Claim: Algorithm Sorted-Balance produces an assignment of jobs to machines with makespan $T \leq \frac{3}{2} T^*$

Proof:- Analysis is similar to the previous algorithm.

Let M_i be the machine that has the maximum load. If M_i has assigned a single job then the schedule is optimal.

So assume M_i has at least two jobs.

Let j be the last job assigned to the machine M_i.

Clearly $j \geq m+1$.

Since the algorithm will assign the first m jobs to m distinct machines.

Thus $t_j \leq t_{m+1} \leq \frac{1}{2} T^*$ (from observation 3)

From here onward Proof is similar to previous algo analysis Proof

We had

$$T_i - t_j \leq T^* \quad \& \quad t_j \leq T^*$$

— (A)

But now we have $t_j \leq \frac{1}{2} T^*$ — (B)

From (A) & (B)

$$T_i \leq \frac{3}{2} T^*$$

hence makespan $T \leq \frac{3}{2} T^*$.