CSC373 Week 9 Notes

Intro to Approx Algo:

- Many problems of practical significance are NP-complete but they are too important to abandon.

- There are at least 3 ways to get around NP-completeness:
  1. If the inputs are small, an algo with exp running time is fine.

  2. We may be able to isolate special cases that we can solve in poly time.

  3. We may be able to come up with approaches to find near-opt solns in poly time. In practice, near-opt is often good enough. We call an algo that returns near-opt solns an approx algo.

- We say than an algo has an approx ratio of $P(n)$ if for any input of size n, the cost C of the soln produced by the algo is within a factor $P(n)$ of the cost C^* of an opt soln.

  I.e. $\max\left(\underbrace{\frac{C}{C^*}}_{\text{Min}}, \underbrace{\frac{C^*}{C}}_{\text{Max}}\right) \leq P(n)$

  ↖ Applies to both max and min

- If an algo achieves an approx ratio of $P(n)$, we call it a $P(n)$-approx algo.

- The def of both approx ratio and $\rho(n)$-approx algo applies to both max and min problems.

  For a max problem, $0 \leq c \leq c^*$ and the ratio $\frac{c^*}{c}$ gives the factor by which the cost of an opt soln is larger than the cost of an approx soln.

  Similarly, for a min problem, $0 \leq c^* \leq c$ and the ratio $\frac{c}{c^*}$ gives the factor by which the cost of the approx soln is larger than the cost of the opt soln.

  These ratios are well-defined.

- The approx ratio of an approx algo is never less than 1. Hence, a 1-approx algo produces an opt soln.

- An approx scheme for an opt problem is an approx algo that takes both an instance of the problem and a value $\varepsilon > 0$ as input s.t. for any fixed ε, the scheme is a $(1+\varepsilon)$-approx algo.

- We say that an approx scheme is a poly-time approx scheme (PTAS) if for every $\varepsilon > 0$, there is a $(1+\varepsilon)$-approx algo that runs in poly time w.r.t n, the size of the input.
  Note: It could have exponential dependence on $\frac{1}{\varepsilon}$.
  E.g. $O(n^{2/\varepsilon})$

- We say that an approx scheme is a fully poly-time approx scheme (FPTAS) if it's an approx scheme and its running time is poly in both $\frac{1}{\varepsilon}$ and n.

  I.e. We say an approx scheme is FPTAS if for every $\varepsilon > 0$, there is a $(1+\varepsilon)$-approx algo that runs in poly time in both $\frac{1}{\varepsilon}$ and n.

  E.g. $O((\frac{1}{\varepsilon})^2 \cdot n^3)$

- Approx Landscape:
  1. FPTAS
     - knapsack problem
  2. PTAS but no FPTAS
     - Makespan problem
  3. C-approx for a constant $c > 1$, but no PTAS
     - Vertex cover and Jisp
  4. $\Theta(\log n)$-approx but no constant approx
     - Set cover
  5. No $n^{1-\varepsilon}$-approx for any $\varepsilon > 0$
     - Graph coloring and max indep set

- Approx Techniques:
  1. Greedy Algo:
     - Make decision on one element at a time in a greedy fashion without considering future decisions.

  2. LP Relaxation:
     - Formulate the problem as an ILP.
     - "Relax" it to an LP by allowing vars to take real values.
     - Find an opt soln of the LP, "round" it to a feasible soln of the original ILP and prove its approx opt

  3. Local Search:
     - Start with an arbitrary soln.
     - keep making "local" adjustments to improve the objective.

- Decision vs Opt Problems:
  - Decision Variant: Does there a soln with obj $\geq k$?
    E.g. Is there an assignment which satisfies at least k clauses of a given CNF formula F?

  - Opt Variant: Find a soln that max the obj.
    E.g. Find an assignment which satisfies the max possible number of clauses of a given CNF formula F.

  - If a decision problem is hard, then the opt variant is hard too.

Greedy Algo Examples:

1. Makespan Minimization:

- Problem: Given m identical machines and n jobs s.t. job j requires processing time t_j, assign the jobs to machines to minimize makespan.

Let $S[i]$ be the set of jobs assigned to machine i in a soln.

Constraints →
Each job must run contiguously on one machine.
Each machine can process at most one job at a time.

Load on machine i: $L_i = \sum_{j \in S[i]} t_j$

Goal: Minimize the max load.
I.e. $L = \max_i L_i$

- Even when m (the num of machines) is 2, this problem is NP hard.

Proof:
We will show that Partition $\leq_p$ Makespan with $m = 2$

Note: Partition states that given a set S containing numbers (ints) that sum up to n, is there a subset of S, S', s.t. the numbers in S' sum up to $\frac{n}{2}$?

n is also given.

for partition

Given S and n, we want to construct an instance of Makespan $(2, J_1, t_1, \ldots, J_n, t_n)$ s.t. Partition is true iff Makespan is true.

Since we have 2 machines, if we want to minimize the max load, then each machine should have a total processing time of

$$\frac{\sum_{i=1}^{n} t_i}{2}$$

I.e. If the total processing time of all n jobs is A, then each machine should take on $\frac{A}{2}$ processing time for best results.

Let each t_i be a number (not already taken) in S. Then, we know that the total processing time is n. If we are able to get the total processing time of both machines to be $\frac{n}{2}$, then there is a subset of S whose sum is $\frac{n}{2}$.

Proof (Makespan → Partition)

If each machine has a total processing time of $\frac{n}{2}$, then there is a subset of S that sums up to $\frac{n}{2}$.

Since there is a 1-1 mapping btwn t_i and S_i and we know that a subset of the jobs' processing time sum up to $\frac{n}{2}$, this means a subset of S sum up to $\frac{n}{2}$. Just use the t_i's in either machine.

Proof (Partition → Makespan)
If there is a subset of S that sums up to $\frac{A}{2}$, use the corresponding t_i's for one machine. This makes it so that both machines have a processing time of $\frac{A}{2}$.

- I will prove that partition is NP-hard by proving Subset Sum $\leq_p$ Partition.

Note: In subset sum, the input is a set of numbers S and a target t and we want to see if a subset of S sums up to t.

Given S and t of Subset sum, we want to create an instance (S', n) of partition.

Suppose the numbers in S sum up to A.
Let $S' = S \cup \{A - 2t\}$
Note that the sum of the elements in S' sum up to $2A - 2t$.
Furthermore, if there's a subset of S that sums up to t (Call this subset S^*), then we know the sum of the elements in $S \setminus S^*$ sum up to $A - t$.
Hence, if we can show a subset of S' that sums up to $\frac{n}{2}$ or $A - t$, then we know that there's a subset in S that sums up to t.

Let $X \subseteq S'$ and $Y \subseteq S'$ and say X sums up to $A - t$, and that $A - 2t$ is in X.
The sum of $X \setminus (A - 2t) = (A - t) - (A - 2t)$
$= A - t - A + 2t$
$= t$
Hence, a subset of S sums up to t.

- Going back to the original question, there are several ways we could order the jobs. It turns out, regardless of the order, greedy gives a 2-approx.

Note: This means that 2-approx is an upperbound on how bad the ratio is, and that it doesn't matter how we sort the jobs, the result will be no worse than 2-approx.

I.e. It can't be worse than 2-approx.

Proof:

Let L^* be the opt makespan result.

To show that the greedy makespan soln is not too much worse than L^*, we first need to show that L^* can't be too low.

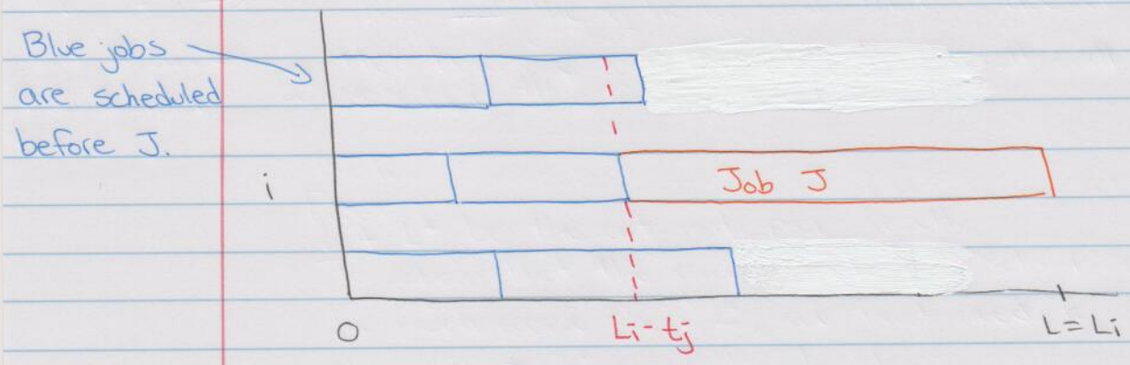
Lemma 1: $L^* \geq \max_j t_j$

Lemma 2: $L^* \geq \frac{\sum_j t_j}{m}$ ← The avg of the processing times.

By Lemma 1, we know that L^* is equal to or greater than the max processing time. Since the max processing time is no less than the avg processing time, it follows that L^* is greater than the avg processing time.

We will order the jobs in ascending order of processing time as well as schedule each job on the machine with the fewest/least load.

E.g.



Define bottleneck to be the machine with the most load. Suppose machine i is the bottleneck.

Let J be the last job scheduled on machine i. First, this means that $L = L_i$ as we've sorted the jobs in asc order of processing time.
Second, $L_i - t_j$ is less than or equal to L_k $\forall k$
$L_i - t_j$ denotes the total processing time of machine i before job j was added. We know that $L_i \leq L_k$ $\forall k$, because job J was added to it. If another machine had a smaller load, Job J would've been added to that machine.

$L_i - t_j \leq L_k \quad \forall k$

$\leq \frac{\sum_k L_k}{m}$ ← Since we know that $L_i - t_j$ is less than or equal to all L_k's, it must be less than the avg of the L_k's as the avg ≥ the smallest value.

$L_i \leq t_j + \frac{\sum_k L_k}{m}$

Moved to other side

$\leq L^* + L^*$ ← By Lemma 1 and 2 respectively.

$= 2L^*$

$t_j = \max_j t_j \rightarrow L^* \geq \max_j t_j$

$= t_j$

- Our analysis is essentially tight. This means that 2-approx is the worst possible result. We say that a C-approx is tight if there exists an approx algo that achieves the upper bound.

An approx is tight if we can find an example that shows that the ratio and the ratio is the best worst case.

- If we averaged $k \neq i$ above we can get a better $2 - \frac{1}{m}$ approx.

I'll show the example: Suppose we have $m(m-1)$ jobs of length 1 and 1 job of length m. The greedy algo would distribute all $m(m-1)$ jobs of unit length equally on the m machines. Then, it would put the job of length m on any machine. Hence, the makespan would be $m-1+m$ or $2m-1$.

From $m(m-1)$ unit jobs

From the 1 job with processing time m

The opt algo would stack the $m(m-1)$ jobs on $m-1$ machines and then put the job with processing time m on the last machine. Now, all machines have a load of m, making the makespan equal to m.

The algo is $\frac{2m-1}{m}$ or $2 - \frac{1}{m}$.

- Previously, when we put the heaviest jobs at the end, we got a $2 - \frac{1}{m}$ approx. Now, let's see what happens when you put it at the beginning.

- We first sort the jobs in dec order by their processing time.
  I.e. $t_1 \geq t_2 \geq \dots \geq t_n$

Lemma 3: If the bottleneck machine i only has one job, j, then the sdn is opt.

If machine i has the most load and only has 1 job, then we know that $L = L_i = t_j$.

Lemma 4: If there are more than m jobs, then $L^* \geq 2 \cdot t_{m+1}$.
We know that the first $m+1$ jobs each must have a processing time of at least t_{m+1}. Furthermore, if there are m machines and $m+1$ jobs, then by Pigeonhole Principle, we know that at least a machine must have 2 jobs. Hence, the processing time of that machine(s) is at least $2t_{m+1}$. Hence, $L^* \geq 2t_{m+1}$.

Claim: This greedy algo achieves a $\frac{3}{2}$-approx.

Proof:

Case 1: Machine i has only 1 job.

- By lemma 3, we know that this is the opt soln (1-approx)

This means → each machine has at most 1 job.

Case 2: Machine i has at least 2 jobs.

- We know that each machine must have at least 1 job.
- Suppose job J is the last job added to machine i. Then, $t_j \leq t_{m+1}$
- $L_i - t_j \leq L_k \quad \forall k$

$$\leq \frac{\sum_k L_k}{m}$$

$$L_i \leq \frac{\sum_k L_k}{m} + t_j$$

$$\leq L^* + \frac{1}{2} L^*$$

L^*: By lemma 2

$\frac{1}{2} L^*$: ← By Lemma 4. Earlier, we said that $L^* \geq 2 t_{m+1}$. $\frac{L^*}{2} \geq t_{m+1} \geq t_j$ (Since $t_j \leq t_{m+1}$)

$$= \frac{3}{2} L^*$$

- However, this isn't a tight bound.
- Graham's greedy algo achieves $(\frac{4}{3} - \frac{1}{3m})$ approx and is tight.

Example:
Suppose we have 2 jobs each of lengths $m, m+1, \ldots, 2m-1$.
Then, we'll one more job of length m.
Our greedy algo has a makespan of $4m-1$.

Proof:
Since we have 2 jobs each of lengths $m, m+1, \ldots, 2m-1$ and we have m machines, each machine will have a total processing time of $m+2m-1$ or $3m-1$.
Then, we still have the one remaining job that takes m time. Hence, the total processing time or makespan is $4m-1$.

The opt soln has a makespan of $3m$.
You pair $(2m-1)$ with $(m+1)$, $(2m-2)$ with $(m+2)$, and so on leaving the 3 jobs of m time alone.
You pair those 3 jobs together. Hence, the makespan is $3m$.

$$\frac{4m-1}{3m} = \frac{4}{3} - \frac{1}{3m} \leftarrow \text{Tight}$$

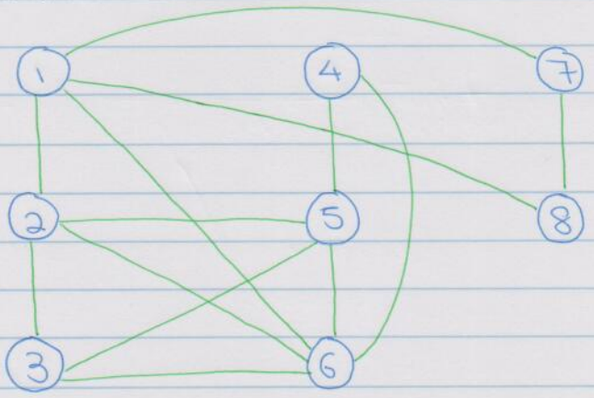
2. Weighted Set Packing:

- Problem: Given a universe of m elements, sets $S_1, \ldots, S_n$ with values $V_1, \ldots, V_n \geq 0$, we want to pick disjoint sets with the max total value.
  I.e. We want to pick $\omega \subseteq \{1, \ldots, n\}$ to max $\sum_{i \in \omega} w_i$ s.t. $S_i \cap S_j = \emptyset \ \forall i, j \in \omega$.

- This problem is NP-hard.
  We can prove this by proving Clique $\leq_p$ WSP

Note: In the Clique problem, the input is an undirected graph G and an int k. We want to know if there exists a clique of size k.
A clique is a subset of nodes, V', s.t. there exists an edge btwn any 2 pairs of nodes in V'.

E.g.



Some cliques are: $\{1,2\}$, $\{2,3\}$, $\{1,7,8\}$, $\{2,3,5,6\}$.

Given an undirected graph G and an int k, the instance of Clique, we want to construct in poly-time a universe U, sets $S_1, \ldots, S_n$ with values $V_1, \ldots, V_n \geq 0$, s.t. Clique is True iff WSP is True. and int C

Note: The decision variant of WSP takes in $U, S_1, \ldots, S_n$ and an int C and returns if there are C pairwise disjoint sets.

Let $U = V$.
Ie. Let U be the nodes.

Let S_i be the set of edges connected to node i in $\overline{G}$.

Let $C = k$.

Then, G has a Clique of size k iff there are k disjoint sets.

Another reduction → We can also do Indep Set $\leq_p$ WSP.
In IS, were given an undirected graph $G = (V, E)$ and an int k and we want to output if there are k vertices in G s.t. no 2 vertices are connected.

Given (G, k) for IS, we want to construct $(S_1, \ldots, S_n, C)$ for WSP in poly time s.t. IS is true iff WSP is true.

Let $S_i = \{$edges connected to vertex $i\}$
Let $C = k$
Now, if we have k disjoint sets $S_1, \ldots, S_k$, then we know there are k vertices in G that aren't connected.

Greedy Template:

1. Sort the sets in some specific order and relabel them as $1, 2, \ldots, n$ in this order.

2. $\omega = \emptyset$

3. For $i = 1, \ldots, n$:
   If $S_i \cap S_j = \emptyset \ \forall j \in \omega$, then $\omega = \omega \cup \{i\}$

4. Return ω

Suppose we sort the sets with non-inc order of values.
I.e $V_1 \geq V_2 \geq \ldots \geq V_n$
We only get an m-approx.
Let opt WSP = L^*.
Let greedy WSP = L.
WTS: $\frac{L}{L^*} = m$

Lemma 1: $L^* \geq V_1$. This is bc any opt algo will choose S_1 as it conflicts with nothing and has the highest value.

Suppose that set S_m is the last set added to ω.
We know that $\omega = \{S_1, \ldots, S_m\}$, so $L = V_1 + \ldots + V_m$.

$$\begin{aligned} L &\leq V_1 + \ldots + V_1 \\ &= m \cdot V_1 \\ &\leq m \cdot L^* \text{ (By Lemma 1)} \end{aligned}$$

Now, suppose that we order the sets in the below method:

$$\frac{v_1}{|S_1|} \geq \frac{v_2}{|S_2|} \geq \cdots \geq \frac{v_n}{|S_n|}$$

We still get an m-approximation. (Very similar proof to the prev one)

$$L^* \geq \frac{v_1}{|S_1|}$$

$$\omega = \left\{ \frac{v_1}{|S_1|}, \ldots, \frac{v_m}{|S_m|} \right\}$$

← Last set added to ω

$$L \leq m \cdot \frac{v_1}{|S_1|}$$
$$\leq m \cdot L^*$$

If we sort the sets in the following way, we get $\sqrt{m}$ - approx: $\frac{v_1}{\sqrt{|S_1|}} \geq \frac{v_2}{\sqrt{|S_2|}} \geq \cdots \geq \frac{v_n}{\sqrt{|S_n|}}$.

Let OPT be some optimal soln.
Let ω be our greedy soln.
For $i \in \omega$, $OPT_i = \{j \in OPT : j \geq i, S_i \cap S_j = \emptyset\}$

This means for each set in ω, we look at sets with a greater or equal index in OPT and get all compatible sets.

Claim 1: $OPT \subseteq \bigcup_{i \in \omega} OPT_i$
Claim 2: It is enough to show that $\forall i \in \omega$, $\sqrt{m} \cdot v_i \geq \sum_{j \in OPT_i} v_j$

We know that $\frac{v_i}{\sqrt{|S_i|}} \geq \frac{v_j}{\sqrt{|S_j|}}$ if $j \geq i$.

Rearranging the eqn, we get $v_j \leq \frac{v_i}{\sqrt{|S_i|}} \sqrt{|S_j|}$

$$\sum_{j \in OPT_i} v_j \leq \frac{v_i}{\sqrt{|S_i|}} \sum_{j \in OPT_i} \sqrt{|S_j|}$$

Now, using the Cauchy-Schwarz Inequality which states $\sum_i x_i y_i \leq \sqrt{\sum_i x_i^2} + \sqrt{\sum_i y_i^2}$, we can show that

$$\begin{aligned}
\sum_{j \in OPT_i} \sqrt{1 \cdot |S_j|} &\leq \sqrt{\sum_{j \in OPT_i} 1^2} + \sqrt{\sum_{j \in OPT_i} |S_j|} \\
&\leq \sqrt{|OPT_i|} \cdot \sqrt{\sum_{j \in OPT_i} |S_j|} \\
&\leq \sqrt{m} \cdot \sqrt{|S_i|}
\end{aligned}$$

$$\begin{aligned}
\sum_{j \in OPT_i} \sqrt{1 \cdot |S_j|} &\leq \sum_{j \in OPT_i} (\sqrt{1})(\sqrt{|S_j|}) \\
&\leq \sqrt{\sum_{j \in OPT_i} (\sqrt{1})^2} \cdot \sqrt{\sum_{j \in OPT_i} (\sqrt{|S_j|})^2} \\
&= \sqrt{\sum_{j \in OPT_i} 1} \cdot \sqrt{\sum_{j \in OPT_i} |S_j|} \\
&\leq \sqrt{|OPT_i|} \cdot \sqrt{m} \\
&\leq \sqrt{|S_i|} \cdot \sqrt{m}
\end{aligned}$$

3. Unweighted Vertex Cover:

- **Problem:** Given an undirected graph $G = (V, E)$, output a vertex cover S of min cardinality.

A vertex cover is a subset of nodes s.t. every edge has at least 1 of its 2 endpoints in S.

Greedy Algo:

1. Start with $S = \emptyset$
2. While there exists an edge whose both endpoints aren't in S, add both its endpoints in S.

Thm: This greedy algo achieves 2-approx.

Proof:

Let S be the soln generated by the greedy algo.
Let S^* be the opt soln.
WTS: $\frac{S}{S^*} = 2$

Lemma 1: Let M be a matching. A matching is a set of edges s.t. no 2 edges share an endpoint. $|S^*| \geq |M|$
This is because we know S^* must contain at least 1 endpoint for each edge in M.

Lemma 2: $|S| = 2 \cdot |M|$ → This is bc with a matching, each endpoint is used at most once. So, when you choose an edge, you're locking 2 vertices.

$\therefore \frac{|S|}{|S^*|} = 2$

LP Relaxation Example:

1. Weighted Vertex Cover:

- Problem: Given an undirected graph $G = (V, E)$ and weights for each vertex, we want to output a vertex cover S of min total weight.

- For each vertex v, create a binary var $x_v \in \{0,1\}$ indicating whether vertex v is chosen in the vertex cover.

Then, computing the min weight VC is equivalent to solving the following ILP:

ILP with binary vars

$$\min \sum_v w_v \cdot x_v$$

w_v: The weight of vertex v

$$\text{s.t.}$$

$$x_u + x_v \geq 1, \quad \forall (u,v) \in E$$

$$x_v \in \{0,1\}, \quad \forall v \in V$$

Now, we'll "relax" the problem by allowing x_v's to hold real values instead of just 0/1.

LP with real vars

$$\min \sum_v w_v \cdot x_v$$

$$\text{s.t.}$$

$$x_u + x_v \geq 1, \quad \forall (u,v) \in E$$

$$x_v \geq 0, \quad \forall v \in V$$

Consider LP opt soln x^*.
Let $\hat{x}_v = 1$ if $x_v^* \geq 0.5$ and $\hat{x}_v = 0$ otherwise.

Claim 1: $\hat{x}$ is a feasible soln of the ILP (I.e. A vc)
This is bc, for every edge $(u,v) \in E$, at least one of $\{x_u^*, x_v^*\}$ must be at least 0.5.
This is bc by the def of VC, at least 1 node of each edge is in the VC.
Hence, at least 1 of $\{\hat{x_u}, \hat{x_v}\}$ is 1.

Claim 2: $\sum_v w_v \cdot \hat{x}_v \leq 2 \cdot \sum_v w_v \cdot x_v^*$
Consider the chart below:

x_v^*	$\hat{x_v}$
≥ 0.5	1
< 0.5	0

Suppose each of the x_v^*'s are 0.5.
Then, we know that each of the $\hat{x_v}$'s are 1.
Hence, $\sum_v w_v \cdot \hat{x}_v = 2 \cdot \sum_v w_v \cdot x_v^*$.
If any of the x_v^*'s are greater than 0.5,
then $\sum_v w_v \cdot \hat{x}_v < 2 \cdot \sum_v w_v \cdot x_v^*$.
Furthermore, if any of the x_v^*'s are less than 0.5,
then its corresponding $\hat{x}_v$ value is 0 and so
$\sum_v \hat{x}_v \cdot w_v \leq 2 \cdot \sum_v w_v \cdot x_v^*$.

By claims 1 and 2, $\hat{x}$ is a VC with weight at most 2* LP opt value and so, it's less than or equal to 2* ILP OPT value.

I.e.
$\hat{x} \leq 2 \cdot$ LP OPT value
$\leq 2 \cdot$ ILP OPT value

- Suppose we've created the LP relaxation version of the original ILP and we're trying to minimize the obj func $c^T x$.

Since the LP min this over a larger feasible space than the ILP, OPT LP obj val $\leq$ OPT ILP obj val

Let X^*_{LP} be an opt LP soln, which we can compute efficiently.

Let X^*_{ILP} be an opt ILP soln, which we can't compute efficiently.

Then, we know $c^T X^*_{LP} \leq c^T X^*_{ILP}$.

However, X^*_{LP} may have non-int values. To get around this, we can efficiently round X^*_{LP} to an ILP feasible soln $\hat{x}$ without inc the obj too much.

If we can prove this, then we can prove our algo achieves p-approx.

$$c^T \hat{x} \leq p \cdot c^T X^*_{LP}$$
$$\leq p \cdot c^T X^*_{ILP} \leftarrow \text{Since we know } c^T X^*_{LP} \leq c^T X^*_{ILP}$$

Thus, our algo will achieve p-approx.

- Suppose now we're max the obj func instead of min.

In this case, $c^T x^*_{LP} \geq c^T x^*_{ILP}$.

Now, we want to efficiently round x^*_{LP} to an ILP feasible soln $\hat{x}$ without decreasing the obj too much.

$$c^T \hat{x} \geq \frac{1}{p}(c^T x^*_{LP}) \geq \frac{1}{p}(c^T x^*_{ILP})$$

} Proving this will show/prove that our algo will achieve p-approx.

- General LP Relaxation Strategy:

1. Your NP-Complete problem amounts to solving
Max $c^T x$ s.t. $Ax \leq b$, $x \in N$ (x doesn't have to be binary)

2. Solve:
   a) Max $c^T x$ s.t. $Ax \leq b$, $x \in R_{\geq 0}$ (LP Relaxation)
   Bc we're max the obj func, we know that LP opt value $\geq$ ILP opt value

   b) Let x^* = LP opt soln

   c) Round x^* to $\hat{x}$ s.t. $c^T \hat{x} \geq \frac{c^T x^*}{p}$

   $$\geq \frac{c^T x_{ILP-OPT}}{p}$$

   Note: The best p you can hope for via this approach for a particular LP-ILP comb is called the integrality gap.

Local Search:

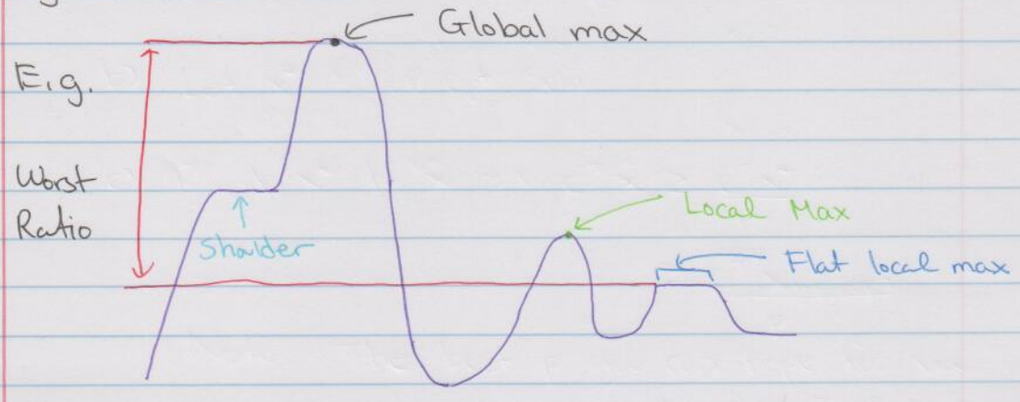
- We start at some point and look at its neighbours/ adj points. If there's a better one, go to the best. Repeat.

- A more formal pseudo-code is:
  Start at/with some feasible soln s
  While there's a better soln s' in the local neighbourhood of s:
    Switch to s'

- However, we need to define:
1. What the initial feasible soln should be.
2. What is "better"
3. What is the "local neighbourhood"

- We've already seen some problems that use this approach, Network Flow and LP via simplex.

- Sometimes, local search doesn't return an opt value and either gets stuck in a local max or gets fooled by a local max.



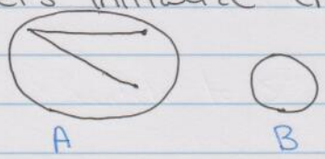
We could get stuck at the Flat local max or think that the local max is the global max.

Local Search Examples:

1. Max-Cut:

- Problem: Given an undirected graph $G = (V, E)$, we want to output a partition (A, B) of V that max the number of edges going across the cut.
I.e. We want to max $|E'|$ where $E' = \{(u,v) \in E \mid u \in A, v \in B\}$

- Let's initialize (A, B) s.t. $A = G$ and $B = \emptyset$.
I.e.

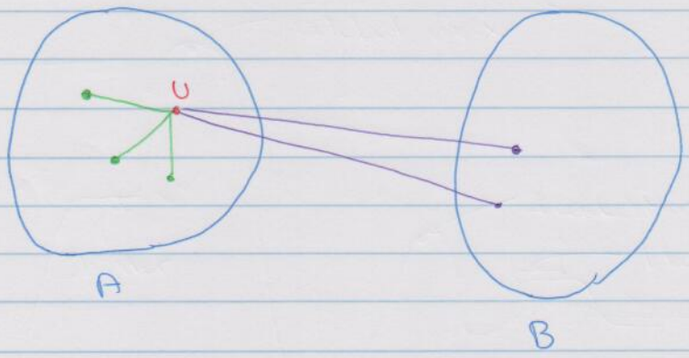


While there's a vertex u s.t. moving u to the other side improves the obj value:

Move u to the other side.

The only time when moving vertex u from A to B improves the obj value is if the number of edges connected to u that's in A is greater than the number of edges connected to u that's going to B.

I.e.



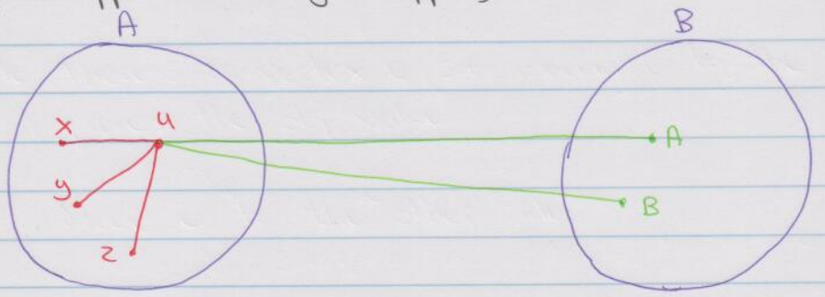
If we move u from A to B, we gain 3 edges but lose 2. Hence, we still improve our obj value.

Note: We <u>can't</u> use the number of vertices as a bound bc it might loop and look at a vertex multiple times.

- The number of iterations is at most $|E|$ since moving a vertex from A to B will improve the obj value by at least 1.

- After the algo stops, at least half of the edges connected to each vertex is part of the cut. This is because, if more than half of the edges are still part of A, then moving that vertex to B would improve the objective function.

E.g. Suppose the algo stopped, and here are our A and B.



If vertex u has more than half of its edges in A, then we seeing moving u from A to B improves the obj value. Hence, bc the algo terminated, we know that the obj value is maxed and so, at least half of the edges of each vertex is part of the cut.

∴ The approx ratio is 2.

2. Exact Max-k Sat:

- Problem: Given an exact k-SAT formula $F = C_1 \wedge C_2 \wedge \dots \wedge C_m$ where each clause C_i has exactly k literals and a weight $w_i \geq 0$, output a truth assignment τ max the total weight of clauses satisfied under τ.

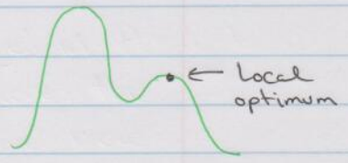
- Let $w(\tau)$ be the total weight of clauses satisfied under τ.

- Let $N_d(\tau)$ be the set of all truth assignments τ' which differ from τ in the values of at least d vars.

- Thm: The local search algo with $d=1$ and $k=2$ gives a $\frac{2}{3}$ - approx ratio.

Proof:
Let τ be a local optimum.
Let S_0 be the set of clauses not satisfied under τ.
Let S_1 be the set of clauses s.t exactly 1 literal is true under τ.
Let S_2 be the set of clauses s.t. both literals are true under τ.
Let $w(S_0)$, $w(S_1)$ and $w(S_2)$ be the corresponding weights.
Our goal is to prove that $w(S_1) + w(S_2) \geq \frac{2}{3}(w(S_0) + w(S_1) + w(S_2))$
or equivalently $w(S_0) \leq \frac{1}{3}(w(S_0) + w(S_1) + w(S_2))$.

Eg.



We say that a clause C involves var j if either x_j or $\bar{x}_j$ is used in C.

Let A_j be the set of clauses in S_0 involving var j.
Let $w(A_j)$ be A_j's weight.

Let B_j be the set of clauses in S_1 involving var j s.t. var j is true under τ.
Let $w(B_j)$ be B_j's weight.

Lemma 1: $2w(S_0) = \sum_j w(A_j)$
Recall that S_0 is the set of all clauses s.t. the clauses are False under τ. This means that both literals in each of those clauses are false. This means that every clause in S_0 is counted twice.

E.g. Suppose $C_1 = (\bar{x}_1 \vee \bar{x}_2) \in S_0$, and the only clause in S_0.
Then, we have $w(S_0) = w(C_1)$.
Now, $w(A_{x_1}) = w(C_1)$ and $w(A_{x_2}) = w(C_1)$.
Hence, $2w(S_0) = \sum_j w(A_j)$

Lemma 2: $w(S_1) = \sum_j w(B_j)$
Recall that S_1 is the set of clauses that has exactly 1 True literal under τ. Hence, every clause in S_1 is listed exactly once on the RHS.

Lemma 3: For each j, $w(A_j) \leq w(B_j)$
Recall that we assumed τ is a local optimum.
This means that making any changes dec the value.
Suppose $w(A_j) > w(B_j)$. Then, we can flip all the x_i's to $\bar{x}_i$ and get a better result. But this contradicts our assumption that τ is a local optimum.

From the above 3 lemmas, we know that
$2w(S_0) \leq w(S_1)$.

Proof:

$2(w(S_0)) \leq \sum_j w(A_j)$ ← By Lemma 1
$\leq \sum_j w(B_j)$ ← By Lemma 3
$= w(S_1)$ ← By Lemma 2

$2w(S_0) \leq w(S_1)$
$2w(S_0) + w(S_0) - w(S_0) \leq w(S_1)$ ← Adding LHS with 0
$3w(S_0) \leq w(S_0) + w(S_1)$ ← Moved one $w(S_0)$ to RHS
$\leq w(S_0) + w(S_1) + w(S_2)$ ← Bc $w(S_2) \geq 0$
$2w(S_0) \leq w(S_1) + w(S_2)$
↑ what we wanted to show.
This implies that $w(S_1) + w(S_2) \geq \frac{2}{3}(w(S_0) + w(S_1) + w(S_2))$