CSC373: Algorithm Design, Analysis and Complexity
Fall 2017

Allan Borodin

November 14, 2017

Week 9 : Annoucements

- Assignment 2 was due earlier today at 10:00 AM.
  We will go over solutions today.
- Term test 2 is tomorrow 5-6 in the exam center, room 200. The test will cover flow networks, polynomial time transformations and reductions, and NP completeness. The usual one sheet hand-written notes are allowed.
- **Next assignment:** The questions for Assignment 3 have been posted. Note that the due date has now been postponed to Monday, Dec 4 at 10:00 AM. Solutions will be posted by 10:10 AM on December 4.
- I will continue to try to answer questions on Piazza as promptly as I can. In most cases, students are answering question correctly and we appreciate that being done. In some cases, questions are being asked that were already discussed in lectures. If you miss a lecture, you should find out what was discussed which can go beyond what the lecture slides.

More announcements

- As previously discussed, we are merging Greg's and Sasa's tutorial sections into one tutorial so as to save TA hours (and given the low attendance). Both Greg's section in BA2139 and Sasa's section in BA2145 will now meet in BA2145.
- There is no tutorial this week as that is when we are holding the term test for all students (excpet those in the morning section who have legitimate conflicts).

Todays agenda

- Discuss solutions to Assignment 2.
- We will be beginning approximation algorithms.
- We'll first consider a couple of online and greedy algorithms.
- One general method to obtain an approximation algorithm (often used for *NP* hard problems) is to use an LP relaxation of an IP formulation of an optimization problem. We then need to "round" the optimal LP solution to an integral solution and analyze the approximation ratio.
- We will also look at other algorithmic paradigms (e.g. greedy, dynamic programming, local search) that are used to derive approximations to optimality.
- We will see that *NP* hard optimization problems will have different possible approximation ratios.
- In our next topic, randomized algorithms, we will see how randomization is sometimes used to fascilitate approximations.

Todays approximation algorithms agenda

- Strict and asymptotic approximation ratios.
- The makespan problem for identical machines.
- A simple online 2-approximation algorithm for the vertex cover problem. A greedy algorithm that does not work.
- An IP formulation of the weighted vertex cover problem and its rounding to produce a 2-approximation. The integrality gap.
- A greedy approximation for the weighted interval selection problem when the value of an interval is equal to the interval length. A simple charging algorithm.
- A greedy algorithm for axis-aligned weighted rectangles in 2-space.
- Abstraction of previous problem to $k + 1$ claw free graphs.
- A greedy 2-approximation algorithm for the unweighted JISP problem.
- Abstract of JISP to "inductive independence graphs".
- A dynamic programming algorithm for the knapsack problem that leads to a *fully polynomial time approximation algorithm* (FPTAS) algorithm for the knapsack problem.
- Brief discussion of *local search* for approximation.

The why of approximation algorithms

We have already seen that many optimization problems are *NP* hard in the sense that computing an optimal solution for the optimization problem immediately solves a corresponding *NP* complete decision problem.

For example, solving the graph colouring optimzation problem immediately lets us determine if a graph has a 3-colouring. Similarly solving the Max-Sat optimization problem (i.e. finding a truth assignment that maximizes the number of satisfied clauses in a CNF formula) immediately lets us decide if a graph is satifiable.
I hope this is obvious

But *NP*-hardness does not preclude being able to find solutions which yield a "good" approximation to the value (for a maximization problem) or cost (for a minimization problem) of an optimal solution.

Approximation algorithms are also often used for optimization problems for which we do know optimal algorithms. But why?

The why of approximation algorithms

We have already seen that many optimization problems are *NP* hard in the sense that computing an optimal solution for the optimization problem immediately solves a corresponding *NP* complete decision problem.

For example, solving the graph colouring optimzation problem immediately lets us determine if a graph has a 3-colouring. Similarly solving the Max-Sat optimization problem (i.e. finding a truth assignment that maximizes the number of satisfied clauses in a CNF formula) immediately lets us decide if a graph is satifiable.
I hope this is obvious

But *NP*-hardness does not preclude being able to find solutions which yield a "good" approximation to the value (for a maximization problem) or cost (for a minimization problem) of an optimal solution.

Approximation algorithms are also often used for optimization problems for which we do know optimal algorithms. But why? Simply stated, the approximation algorithm may be more efficient and/or easier to implement and/or other considerations prevent using an optimal algorithm.

Approximation ratios for minimization problems

In order to measure how well we are approximating an optimization problem (with respect to the worst case perspective), we define the *approximation ratio* of an algorithm ALG.

For a minimization problem, we say that ALG has an approximation $c \geq$ (which can be a constant or a function of the input "size" n) if for all input instances $\mathcal{I}$ we have $Cost[ALG(\mathcal{I})] \leq c \cdot Cost[OPT(\mathcal{I}]$ where OPT is an optimal solution.

This is called a strict approximation ratio (which is what we will mainly consider today). There is an asymptotic approximation ratio defined as $\lim_{Cost[OPT(\mathcal{I})] \to \infty} \frac{Cost[ALG(\mathcal{I})]}{Cost[OPT(\mathcal{I}]}$.

Approximation ratios for maximization problems

We have the analogous concept for maximization problems. One possible way to state a strict approximation ratio for a maximizatio problem is as follows:
ALG has an approximation $c \geq 1$ (which can be a constant or a function of the input "size" n) if for all input instances $\mathcal{I}$ we have $Profit[ALG(\mathcal{I})] \geq \frac{1}{c} Profit[OPT(\mathcal{I}]$ where OPT is an optimal solution.

Whereas for minimization problems, approximation ratios are always such that $c \geq 1$, for maximization problems, one often states approximation ratios as a fraction of the optimal profit (i.e., as $\frac{1}{c}$ where $c \geq 1$. So if you are reading about approximation ratios for a maximization problem, you will often see claims like an algorithm achieves (for example) approximation ratio $\frac{3}{4}$.

There is an analogous concept of asymptotic approximation ratios for maximization problems.

The approximation landscape

We repeat some comments from Week 3.

- In term of computing optimal solutions, all "NP complete optimization problems" (i.e. optimization problems corresponding to NP complete decision problems) can be viewed (up to polynomial time) as a single class of problems.
- But in the world of approximation algorithms, this single class splits into many classes of approximation guarantees. Up to our believed complexity assumptions, we next discuss these possibilities.

Definition

1. An FPTAS (Fully Polynomial Time Approximation Scheme) is a $(1+\epsilon)$ approximation algorithm using poly time in the input encoding and $\frac{1}{\epsilon}$.
2. A PTAS (Polynomial Time Approximation Scheme) is a $(1+\epsilon)$ approximation algorithm using poly time in the input encoding but can have any complexity in terms of $\frac{1}{\epsilon}$.

Different approximation possibilities for NP complete optimization

Given widely believed complexity claims

1. An FPTAS
   - e.g. the knapsack problem
2. A PTAS but no FPTAS
   - e.g. makespan (when the number of machines m is not fixed but rather is a parameter of the problem.
3. Having a constant $c > 1$ approximation but no PTAS
   - e.g. Vertex cover and JISP to be discussed today
4. An $\Theta(\log n)$ approximation and no constant approximation
   - e.g. set cover H_n essentially tight.
5. No $n^{1-\epsilon}$ approximation for any $\epsilon > 0$
   - e.g. graph colouring and MIS for arbitrary graphs

Here n stands for some input size parameter (e.g. size of the universe for set cover and number of nodes in the graph for colouring and MIS).

An aside: The bin packing problem

Before we discuss the makespan problem, lets take a short diversion to mention the following related *bin packing problem*:
Given: B and $\{a_1, a_2, \ldots, a_n\}$ where each $a_i \leq B$, the bin size.
Goal: Pack the elements $\{a_i\}$ into the fewest number of bins such that sum of the elements in a bin does not exceed the bin size B.

I am mentioning the bin packing problem as it is one of the most studied *NP* hard approximation problems. Why *NP* hard?

An aside: The bin packing problem

Before we discuss the makespan problem, lets take a short diversion to mention the following related *bin packing problem*:
Given: B and $\{a_1, a_2, \ldots, a_n\}$ where each $a_i \leq B$, the bin size.
Goal: Pack the elements $\{a_i\}$ into the fewest number of bins such that sum of the elements in a bin does not exceed the bin size B.

I am mentioning the bin packing problem as it is one of the most studied *NP* hard approximation problems. Why *NP* hard?
Since the PARTITION problem is *NP* hard, we cannot distinguish between needing 2 or 3 bins by choosing $B = (\sum_i a_1)/2$. Hence the *strict* approximation ratio is at least $\frac{3}{2}$ **if we assume** $P \neq NP$.

There are online algorithms that produce constant approximation ratios. For example, the online algorithm "best fit greedy" achieves approximation $\frac{17}{10}$

There are other algorithms that for any input instance $\mathcal{I}$ pack the items in $OPT(\mathcal{I}) + o(OPT(\mathcal{I}))$ Hence the *asymptotic* approxination ratio is 1,

Online and greedy algorithms

We recall that greedy algorithms consider the input items in some order and for each input item make an irrevocable "greedy" decision.

An online algorithm processes the input items in the order given (i.e. the algorithm has no control over the order of input arrivals).

For our first approximation algorithm we consider the makespan problem on m identical machines. Similar to the question on the Assignment, the goal here is to schedule n jobs (each job J_i having a processing time or load p_i)on m identical machines so as to minimize the latest completion time.

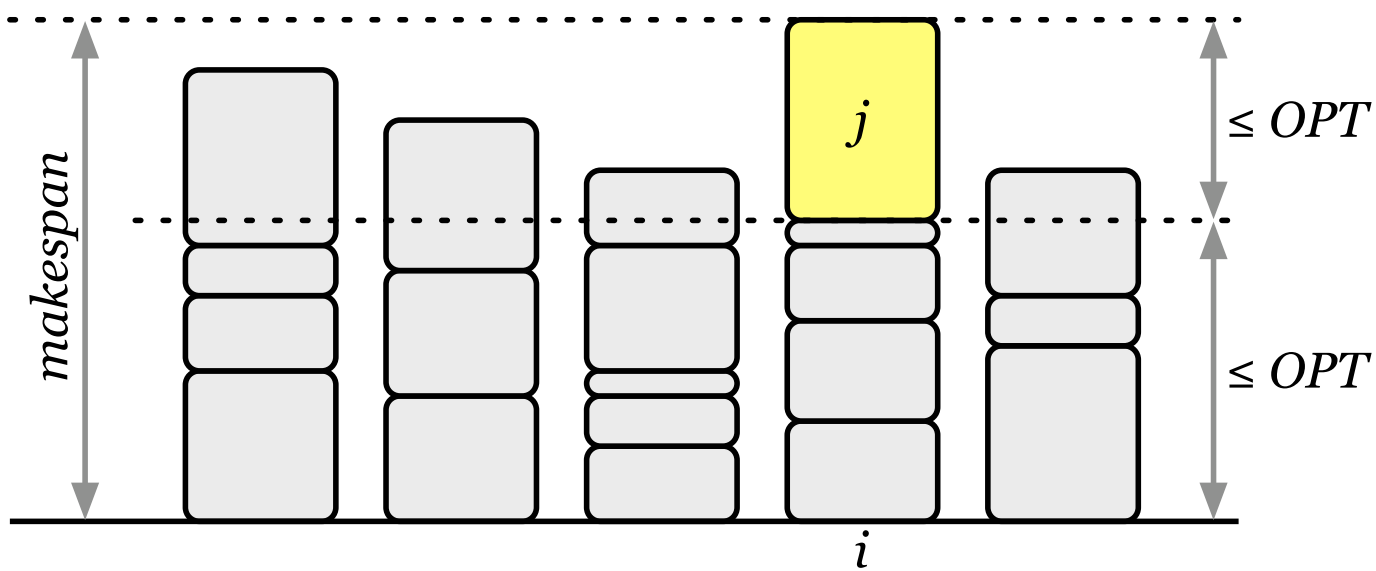
This is an NP hard optimization problem even for just $m = 2$ since an optimal algorithm for 2 machines solves the PARTITION decision problem.

Since we showed the PARTITION problem is NP complete, we cannot hope to have a polynomial time algorithm that produces an optimal solution for all input instances **again if we assume that** $P \neq NP$.

Graham's online greedy algorithm

Consider input jobs in any order (e.g. as they arrive in an *online* setting) and schedule each job J_j on any machine having the least load thus far.

- The **approximation ratio** for this algorithm is $2 - \frac{1}{m}$; that is, for any set of jobs $\mathcal{J}$, $C_{Greedy}(\mathcal{J}) \leq (2 - \frac{1}{m}) C_{OPT}(\mathcal{J})$.
  - C_A denotes the cost (or makespan) of a schedule A.
  - OPT stands for any optimum schedule.
- **Basic proof idea:** $OPT \geq (\sum_j p_j)/m$; $OPT \geq max_j p_j$
  What is C_{Greedy} in terms of these requirements for any schedule?



[picture taken from Jeff Erickson's lecture notes]

Comments on Graham's online greedy algorithm

- Graham's algorithm is considered to be the first paper formally studying worst case approximation algorithms.
- In the online "competitive analysis" literature the ratio $\frac{C_A}{C_{OPT}}$ is called the **competitive ratio** and it allows for this ratio to just hold in the limit as C_{OPT} increases. This is then just the *asymptotic approximation ratio*.
- The approximation ratio for the online greedy is "tight" in that there is a sequence of jobs forcing this ratio. The nemesis input sequence is a sequence of $m(m-1)$ jobs each with processing time $p_i = 1$ and a final job with processing time $p_{m(m-1)+1} = m$. The greedy algorithm has makespan $m + (m-1) = 2m - 1$ while an optimal schedule has makespan m. Why?
- This bad input sequence suggests a better algorithm, namely the LPT (offline) greedy algorithm.

The LPT algorithm for the identical machines makespan problem

Graham's LPT algorithm

Sort the jobs so that $p_1 \geq p_2 \ldots \geq p_n$ and then greedily schedule jobs on the least loaded machine.

- The (tight) approximation ratio of LPT is $\left(\frac{4}{3} - \frac{1}{3m}\right)$.
- It is believed that this is the **best** "greedy" algorithm but how would one prove such a result? This of course raises the question as to what is a greedy algorithm.
- Assuming we maintain a priority queue for the least loaded machine,
  - the online greedy algorithm would have time complexity $O(n \log m)$ which is $(n \log n)$ since we can assume $n \geq m$.
  - the LPT algorithm would have time complexity $O(n \log n)$.

The weighted and unweighted vertex cover problem

The vertex cover optimization problem:
Given a graph $G = (V, E)$
Goal: To compute a minimum size cover V'; that is, a subset $V' \subseteq V$ such that for all $e = (u, v) \in E$, either u or v (or both) are in V'.

In the weighted case, there is a weight function $w : V \rightarrow \mathbb{R}^{\geq 0}$ and the goal is to minimize the weight of a cover.

NOTE: We are maintaining a worst case perspective and one can always ask about computing an optimal or approximate solution when inputs are restricted or coming from some distrubution.

For this problem (and, in general, for graph problems) there are two obvious choices for what is an *input item*; namely,
(1) The edges are the input items and each edge is repreesented by its endpoints
(2) The vertices are the input items and each vertex is represented by its vertex or edge adjacency list.

An online greedy algorithm in the edge input model

We first consider the edge input model

Online greedy algorithm for unweighted vertx cover

$V' = \varnothing; E' = E$ % E' is the set of currently uncovered edges
$M = \varnothing$ % The algorithm is also creating a maximal mathching
While $E' \neq \varnothing$
  Let $e = (u, v)$ be the next uncovered edge
  $E' = E' \setminus \{$ all edges adjacent to u or $v\}$
  $V' = V' \cup \{u, v\}$
  $M = M \cup \{e\}$
End While

Claim: $|V'| \leq 2 \cdot |V^*|$ for any vertex cover V^*.
Proof: The algorithm is creating a maximmal matching and a vertex cover must contain at least one vertex for each edge in a maximal matching. The algorithm is taking both edges.

A greedy approximation algorithm in the vertex adjacency model

We will next consider the vertex adjacency input model. The "natural" greedy algorithm in this model chooses its next vertex to process and add to the vertex cover by choosing the vertex adjacent to the most uncovered edges.

Online greedy algorithm for unweighted vertex cover

$V' = \varnothing; E' = E$ % E' is current uncovered edges
While there are any uncovered edges
  Let $v = (u_1, \ldots, u_{d_v})$ be the input vertex such that the number of edges $(v, u_i) \in E'$ is maximum
  $V' = V' \cup \{v\}$;
  Remove all (v, u_i) from E'
End While

Surprisingly, although it is not obvious, this algorithm is does not result in a constant approximation. Rather the approximation ratio is $H_{d_{max}} \approx \ln d_{max}$ where d_{max} is the maximum vertex degree. .

LP relaxation and rounding

- One standard way to use IP/LP formulations is to start with an IP representation of the problem and then relax the integer constraints on the x_j variables to be real (but again rational suffice) variables.
- We start with the well known simple example for the weighted vertex cover problem. Let the input be a graph $G = (V, E)$ with a weight function $w : V \rightarrow \Re^{\geq 0}$. To simplify notation let the vertices be $\{1, 2, \ldots . n\}$. Then we want to solve the following "natural IP representation" of the problem:
  - Minimize $\mathbf{w} \cdot \mathbf{x}$
  - subject to $x_i + x_j \geq 1$ for every edge $(v_i, v_j) \in E$
  - $x_j \in \{0, 1\}$ for all j.
- The *intended meaning* is that $x_j = 1$ iff vertex v_j is in the chosen cover. The constraint forces every edge to be covered by at least one vertex.
- Note that we could have equivalently said that the x_j just have to be non negative integers since it is clear that any optimal solution would not set any variable to have a value greater than 1.

LP rounding for the natural weighted vertex cover IP

- The "natural LP relaxation" then is to replace $x_j \in \{0,1\}$ by $x_j \in [0,1]$ or more simply $x_j \geq 0$ for all j.
- It is clear that by allowing the variables to be arbitrary reals in [0,1], we are admitting more solutions than an IP optimal with variables in $\{0,1\}$. Hence the LP optimal has to be at least as good as any IP solution and usually it is better.
- The goal then is to convert an optimal LP solution into an IP solution in such a way that the IP solution is not much worse than the LP optimal (and hence not much worse than an IP optimum)
- Consider an LP optimum $\mathbf{x}^*$ and create an integral solution $\bar{\mathbf{x}}$ as follows: $\bar{x}_j = 1$ iff $x_j^* \geq 1/2$ and 0 otherwise. We need to show two things:
  1. $\bar{\mathbf{x}}$ is a valid solution to the IP (i.e. a valid vertex cover).
  2. $\sum_j w_j \bar{x}_j \leq 2 \cdot \sum_j w_j x_j^* \leq 2 \cdot \mathit{IP\text{-}OPT}$; that is, the LP relaxation results in a 2-approximation.

The integrality gap

- Analogous to the locality gap (that we will discuss for local search), for LP relaxations of an IP we can define the integrality gap (for a minimization problem) as $\max_{\mathcal{I}} \frac{IP-OPT}{LP-OPT}$; that is, we take the worst case ratio over all input instances $\mathcal{I}$ of the IP optimum to the LP optimum. (For maximization problems we take the inverse ratio.)
- Note that the integrality gap refers to a particular IP/LP relaxation of the problem just as the locality gap refers to a particular neighbourhood.
- The same concept of the integrality gap can be applied to other relaxations such as in semi definite programming (SDP).
- It should be clear that the simple IP/LP rounding we just used for the vertex cover problem shows that the integrality gap for the previously given IP/LP formulation is at most 2.
- By considering the complete graph K_n on n nodes, it is also easy to see that this integrality gap is at least $\frac{n-1}{n/2} = 2 - \frac{1}{n}$.

Integrality gaps and approximation ratios

- When one proves a positive (i.e upper) bound (say c) on the integrality gap for a particular IP/LP then usually this is a constructive result in that some proposed rounding establishes that the resulting integral solution is within a factor c of the LP optimum and hence this is a c-approximation algorithm.
- When one proves a negative bound (say c') on the integrality gap then this is only a result about the given IP/LP. In practice we tend to see an integrality gap as strong evidence that this particular formulation will not result in a better than c' approximation. Indeed I know of no natural example where we have a lower bound on an integrality gap and yet nevertheless the IP/LP formulation leads "directly" into a better approximation ratio.
- In theory some conditions are needed to have a provable statement. For the VC example, the rounding was "oblivious" (to the input graph). In contrast to the K_n input, the LP-OPT and IP-OPT coincide for an even length cycle. Hence this integrality gap is a tight bound on the formulation using an oblivious rounding.

Some *charging arguments* for approximation guarantees

In a somewhat different approach to proving approximation bounds, we want to show how to use a charging argument; that is, in the next couple of examples, we will *charge* the weight of any algorithm (and in particular an optimal algorithm) to items in a greedy solution. The same idea can be used in local search algorithms.

Note that we considered a charging argument in proving the optimality of the greedy EFT algorithm for the unweighted interval scheduling problem.

Let first consider a problem for which there is an optimal algorithm. Namely, lets consider a restricted version of the weighted interval scheduling problem (on one machine) where the weight (or valule) of any interval is equal to its length.

We know there is an optimal dynamic programming algorithm for the weighted interval scheduling problem so this is just an exercise to show the charging method.

A simple greedy algorithm for proportional profit interval scheduling

Consider the greedy algorithm that sorts by non-increasing weight (= processing time) and accepts greedily (i.e. if an interval doesn't conflict with previously selected intervals, then select it).

Claim: This is a 3-approximation (or $\frac{1}{3}$ approximation for those who prefer fractional approximations for maximization problems). That is the weight of the greedy solution is at least $\frac{1}{3}$ of an optimal solution.
Proof: For every interval in say an OPT solution, we will charge its weight to a unique item in the Greedy solution. This charging will be done so as to guarantee that the charge to say interval I in the greedy solution will be at most 3 times its own weight. **Remainder of proof on board**
Note: In graph theoretic terms, we are approximating the maximum weighted independent set problem in an interval graph.

The weighted independent set problem (WISP) for the intersection graph of axis parallel translates of a rectangle

Consider an axis parallel rectangle R in 2-space. An axis paralalel translate of R is a copy of R shifted left, right, down, up. We consider the intersection graph of n translates of a fixed rectangle R where like interval graphs, the intersection graph has an edge whenever the rectangles intersect. Each translate R_j has a weight w_j and the goal in to choose a non intersecting subset S of these n translates so as to maximize the weight of the rectangles in S.

Claim: Consider any one of the n rectanngles, call it R^* and let $R_1, \ldots R_m$ be the translates that intersect R^*. Then there can be at most 4 of these R_i that do not intersect each other. Why?

The weighted independent set problem (WISP) for the intersection graph of axis parallel translates of a rectangle

Consider an axis parallel rectangle R in 2-space. An axis paralalel translate of R is a copy of R shifted left, right, down, up. We consider the intersection graph of n translates of a fixed rectangle R where like interval graphs, the intersection graph has an edge whenever the rectangles intersect. Each translate R_j has a weight w_j and the goal in to choose a non intersecting subset S of these n translates so as to maximize the weight of the rectangles in S.

Claim: Consider any one of the n rectanngles, call it R^* and let $R_1, \ldots R_m$ be the translates that intersect R^*. Then there can be at most 4 of these R_i that do not intersect each other. Why?

As all translates have the same dimension, all the intersecting translates must intersect at some corner of R^*.

A greedy algorithm for the WISP for the intersection graph of axis parallel translates of a rectangle

Following the discussion for the weighted interval selection problem with proportional weights, we again consider a greedy algorithm that sorts by non-increasing weight; i.e. $w_1 \geq w_2 \ldots \geq w_n$ and accepts greedily. What is the approximation ratio of this algorithm and how would you prove it?.

A greedy algorithm for the WISP for the intersection graph of axis parallel translates of a rectangle

Following the discussion for the weighted interval selection problem with proportional weights, we again consider a greedy algorithm that sorts by non-increasing weight; i.e. $w_1 \geq w_2 \ldots \geq w_n$ and accepts greedily. What is the approximation ratio of this algorithm and how would you prove it?. We use a charging argument and the property that for any rectangle R in the greedy solution, there are at most 4 non-intersecting (i.e., independent) rectangles in an OPT solution intersecting R.

There is a nice graph theoretic way to abstract this independence property. **Definition** A graph $G = (V, E)$ is $k+1$ *claw free* if for every $v \in V$, there are at most k independent vertices in the neighbourhood $Nbhd(v) = \{u \in V : (v, u) \in E\}$.

Often, but not always, "intersection graphs" are $k+1$ claw free for some k. In particular, the previous example was a 5 claw free graph (given the assumption of having one fixed dimension). The interval selection problem does not result in a $k+1$ claw free graph for any fixed k. Why?

The WISP for $k+1$ claw free graphs

The suggested greedy algorithm for the intersection graph of axis parallel translates of a rectangle will always provide a solution that has value at least $\frac{1}{4}$ of an optimal solution.

The charging argument shows us how to charge the weight of rectangles in an OPT solution to a rectangle R in the greedy solution. By the greedy ordering any rectangles intersecting R have weight no more than the weight of R. There can be at most 4 disjoint rectangles intersecting R, and this completes the charging argument.

The same argument can be used to show that the WISP for $k+1$ claw free graphs can be approximated by a greedy algorithm resulting in a solution that obtains at least a fraction $\frac{1}{k}$ of an optimal solution.

The Job Interval Selection Problem (JISP)

We consider the following extension of interval scheduling. In the JISP problem, we are given intervals $I_j = (s, f_j, C_j)$ where s_j and t_j are as before the start and finish time of the interval. In addition, C_j is the class or job of which I_j is member. A set of intervals S is a feasible solution if (as before) the intervals in S do not intersect and for each class C there is at most one interval in S having class C.

In the unweighted version, our goal is to compute a maximum size feasible set of intervals. In the weighted case (WJISP), the goal is to maximize the weight of a feasible set.

Although the interval selection problem is solvable (very efficiently), the JISP is *NP* hard. It is known that it cannot have a PTAS but the best approximation ratio remains an open problem.

The greedy algorithm for JISP

We can obtain a 2-approximation for JISP by the same optimal greedy algorithm that we used for the unweighted interval selection problem. Namely, the greedy algorithm sorts intervals so that $f_1 \leq f_2 \ldots \leq f_n$ and then accepts greedily.

We can prove the stated approximation by a charging argument that charges intervals in an optimal solution OPT to intervals in the greedy solution so that at most two intervals in OPT are charged to any interval in the greedy solution.

The graph theoretic abstraction of interval selection is the class of *chordal graphs*. One characterization is that a graph $G = (V, E)$ is a chordal graph is there is an ordering of the vertices $v_1, v_2, \ldots v_n$ such that for all i, $Nbhd(v_i) \cap \{v_{i+1}, \ldots, v_n\}$ is a clique. Equivalently, there is at most one independent vertex in $Nbhd(v_i) \cap \{v_{i+1}, \ldots, v_n\}$.

Such a vertex ordering is called a *perfect elimination ordering* (PEO).

Interval graphs are chordal graphs

It can be observed that interval graphs ,intersection graphs of intervals $[s_j, f_j)$, are chordal graphs where the ordering is given by $f_1 \leq f_2 \ldots \leq f_n$.

Thinking back to our discusion of the greedy algorithm for interval selection (i.e., the max independent set problem for interval graphs), we see that the algorithm used the PEO given by non-decreasing finish times $\{f_j\}$.

Similarly, we solved interval colouring (i.e., the colouring problem for interval graphs) used the reverse of the PEO. That is, schedule by sorting intervals so that $s_1 \leq s_2 \ldots s_n$ and then coloured greedily. Equivalently, we can sort so that $f_1 \geq f_2 \ldots \geq f_n$ (i.e., the reverse of the PEO) and colour greedily.

Extending chordal graphs

In order to model the JISP problem in graph theoretic terms, we generalize the idea of a PEO. We csan say that a graph $G = (V, E)$ is an *inductive k-independence graph* if the vertices can be ordered $v_1, v_2, \ldots, v_n$ so that there are at most k independent vertices in $Nbhd(v_i) \cap \{v_{i+1}, \ldots, v_n\}$.

We can call such an ordering a k-PEO.

Just as interval graphs are chordal, the intersection graphs induced by the JISP problem are *inductive 2-indepedence graphs* where the same ordering $f_1 \leq f_2 \ldots f_n$ provides the appropriate 2-PEO.

Note: $k + 1$ claw free graphs are a special case of inductive k-independence graphs. That is, any ordering of the vertices is a k-PEO.

For example, the intersection graph of translates of a unit radius disk in 2-space is both a a 6-claw free graph and an inductive 4 independence graph (ordering by non-increasing radius).
How would you approximate the (unweighted) maximumum independent set and colouring problems for inductive k independence graphs?

An FPTAS for the knapsack problem

We recall our discussion of the knapsack problem from Week 3.

The Knapsack problem

- In the knapsack problem we are given a set of n items $I_1, \ldots, I_n$ and a size bound B where where each item $I_j = (s_j, v_j)$ with s_j being the size of the item and v_j the value.
- A feasible set is now a subset of items S such that the sum of the sizes of items in S is at most the bound B.
- Goal: Find a feasible set S that maximizes the sum of the values of items in S.

- In general we can allow real valued parameters but in some algorithms need to restrict attention to integral parameters. But by scaling inputs this is not a significant restriction.
- This is known to be an NP hard problem but as we now recall from Week 3, it is only "weakly NP hard" and there is an FPTAS for this problem.

A DP algorithm for the knapsack problem leading to an FPTAS

- In the first algorithm, if the sizes (or the bound B) are small (i.e. $B = poly(n)$) then the algorithm runs in polynomial time.
- What if the values $\{v_i\}$ are integral and small?
- Consider the following semantic array

$$W[i, v] = \begin{cases} \text{minimum size required to obtain at least profit } v \text{ using} \\ \qquad\qquad \text{a subset of the items } \{I_1, \ldots, I_i\} \text{ if possible} \\ \infty \text{ otherwise} \end{cases}$$

- The desired optimum value is $\max\{v : W[n, v] \text{ is at most } B\}$.

An FPTAS for the knapsack problem

- This algorithm can be used as the basis for an efficient approximation algorithm for all input instances.

- The basic idea is relatively simple:
  - The high order bits/digits of the values can determine an approximate solution (disregarding low order bits after rounding up).
  - The fewer high order bits we use, the faster the algorithm but the worse the approximation.
  - The goal is to scale the values in terms of a parameter ϵ so that a $(1+\epsilon)$ approximation is obtained with time complexity polynomial in n and $(1/\epsilon)$.
  - The details are given in the DPV text (section 9.2.4) or the KT text (section 11.8).
  - Namely, KT sets $\hat{v}_i = \lceil \frac{v_i n}{\epsilon v_{max}} \rceil$ where $v_{\max} = \max_j\{v_j\}$. DPV use the floor $\lfloor\ \rfloor$.
  - The running time is $O(n^3/\epsilon)$.

Local Search: another conceptually simple approach

We now begin a discussion of *local search* which for me, along with greedy algorithms, is one of the two conceptually simplest search/optimization paradigms. (We briefly mentioned local search when discussing flows.)

The vanilla local search paradigm

"Initialize" S
While there is a "better" solution S' in the "local neighbourhood" $Nbhd(S)$
$S := S'$
End While

If and when the algorithm terminates, the algorithm has computed a *local optimum*. To make this a precise algorithmic model, we have to say:

1. How are we allowed to choose an initial solution?
2. What constitutes a reasonable definition of a local neighbourhood?
3. What do we mean by "better"?

Answering these questions (especially as to defining a local neighbourhood) will often be quite problem specific.

Towards a precise definition for local search

- We clearly want the initial solution to be efficiently computed and to be precise we can (for example) say that the initial solution is a random solution, or a greedy solution or adversarially chosen.
  Of course, in practice we can use any efficiently computed solution.
- We want the local neighbourhood $Nbhd(S)$ to be such that we can efficiently search for a "better" solution (if one exists).
  1. In many problems, a solution S is a subset of the input items or equivalently a $\{0,1\}$ vector, and in this case we often define the $Nbhd(S) = \{S' | d_H(S, S') \leq k\}$ for some "small" k where $d_H(S, S')$ is the Hamming distance.
  2. More generally whenever a solution is a vector over a small domain D, we can use Hamming distance to define a local neighbourhood. Hamming distance k implies that $Nbhd(S)$ can be searched in at most time $|D|^k$.
  3. As we previously discussed, we can view Ford Fulkerson flow algorithms as local search algorithms where the (possibly exponential size but efficiently search-able) neighbourhood of a flow solution S are flows obtained by adding an augmenting path flow.

What does "better" solution mean? Oblivious and non-oblivious local search

- For a search problem, we would generally have a non-feasible initial solution and "better" can then mean "closer" to being feasible.
- For an optimization problem it usually means being an improved solution which respect to the given objective. For reasons I cannot understand, this has been termed *oblivious* local search. I think it should be called greedy local search.
- For some applications, it turns out that rather than searching to improve the given objective function, we search for a solution in the local neighbourhood that improves a related potential function and this has been termed non-oblivious local search.
- In searching for an improved solution, we may want an arbitrary improved solution, a random improved solution, or the best improved solution in the local neighbourhood.
- For efficiency we sometimes insist that there is a "sufficiently better" improvement rather than just better.

The weighted max cut problem

- Our first local search algorithm will be for the (weighted) max cut problem defined as follows:

The (weighted) max-cut problem

- Given a (undirected) graph $G = (V, E)$ and in the weighted case the edges have non negative weights.
- **Goal:** Find a partition (A, B) of V so as to maximize the size (or weight) of the cut $E' = \{(u, v) | u \in A, v \in B, (u, v) \in E\}$.

- We can think of the partition as a characteristic vector χ in $\{0, 1\}^n$ where $n = |V|$. Namely, say $\chi_i = 1$ iff $v_i \in A$.
- Let $N_d(A, B) = \{(A', B') |$ the characteristic vector of (A') is Hamming distance at most d from $(A)\}$
- So what is a natural local search algorithm for (weighted) max cut?

A natural oblivious local search for weighted max cut

Single move local search for weighted max cut

Initialize (A, B) arbitrarily
WHILE there is a better partition $(A', B') \in N_1(A, B)$
    $(A, B) := (A', B')$
END WHILE

- This single move local search algorithm is a $\frac{1}{2}$ approximation; that is, when the algorithm terminates, the value of the computed local optimum will be at least half of the (global) optimum value.
- In fact, if W is the sum of all edge weights, then $w(A, B) \geq \frac{1}{2}W$.
- This kind of ratio is sometimes called the absolute ratio or totality ratio and the approximation ratio must be at least this good.
- The worst case (over all instances and all local optima) of a local optimum to a global optimum is called the locality gap.
- It may be possible to obtain a better approximation ratio than the locality gap (e.g. by a judicious choice of the initial solution) but the approximation ratio is at least as good as the locality gap.

Proof of totality gap for the max cut single move local search

- The proof is based on the following property of any local optimum:

$$\sum_{v \in A} w(u, v) \leq \sum_{v \in B} w(u, v) \text{ for every } u \in A$$

- Summing over all $u \in A$, we have:

$$2 \sum_{u,v \in A} w(u, v) \leq \sum_{u \in A, v \in B} w(u, v) = w(A, B)$$

- Repeating the argument for B we have:

$$2 \sum_{u,v \in B} w(u, v) \leq \sum_{u \in A, v \in B} w(u, v) = w(A, B)$$

- Adding these two inequalities and dividing by 2, we get:

$$\sum_{u,v \in A} w(u, v) + \sum_{u,v \in B} w(u, v) \leq w(A, B)$$

- Adding $w(A, B)$ to both sides we get the desired $W \leq 2w(A, B)$.

The complexity of the single move local search

- **Claim:** The local search algorithm terminates on every input instance.
  - Why?

The complexity of the single move local search

- **Claim:** The local search algorithm terminates on every input instance.
  - Why?
- Although it terminates, the algorithm could run for exponentially many steps.
- It seems to be an open problem if one can find a local optimum in polynomial time.
- However, we can achieve a ratio as close to the state $\frac{1}{2}$ totality ratio by only continuing when we find a solution (A', B') in the local neighborhood which is "sufficiently better". Namely, we want

$$w(A', B') \geq (1 + \epsilon) w(A, B) \text{ for any } \epsilon > 0$$

- This results in a totality ratio $\frac{1}{2(1+\epsilon)}$ with the number of iterations bounded by $\frac{n}{\epsilon} \log W$.

Final comment on this local search algorithm

- It is not hard to find an instance where the single move local search approximation ratio is $\frac{1}{2}$.
- Furthermore, for any constant d, using the local Hamming neighbourhood $N_d(A, B)$
  still results in an approximation ratio that is essentially $\frac{1}{2}$.
  And this remains the case even for $d = o(n)$.
- It is an open problem as to what is the best "combinatorial algorithm" that one can achieve for max cut.
- There is a vector program relaxation of a quadratic program that leads to a .878 approximation ratio.

Exact Max-k-Sat

- **Given:** An exact k-CNF formula

$$F = C_1 \wedge C_2 \wedge \ldots \wedge C_m,$$

where $C_i = (\ell_i^1 \vee \ell_i^2 \ldots \vee \ell_i^k)$ and $\ell_i^j \in \{x_k, \bar{x}_k \mid 1 \leq k \leq n\}$.
In the weighted version, each C_i has a weight w_i.

- **Goal:** Find a truth assignment τ so as to maximize

$$W(\tau) = w(F \mid \tau),$$

the weighted sum of satisfied clauses w.r.t the truth assignment τ.

- It is NP hard to achieve an approximation better than $\frac{7}{8}$ for (exact) Max-3-Sat and hence for the non exact versions of Max-k-Sat for $k \geq 3$.

The natural oblivious local search

- A natural oblivious local search algorithm uses a Hamming distance d neighbourhood:
  $N_d(\tau) = \{\tau' : \tau \text{ and } \tau' \text{ differ on at most } d \text{ variables } \}$

Oblivious local search for Exact Max-k-Sat

Choose any initial truth assignment τ
WHILE there exists $\hat{\tau} \in N_d(\tau)$ such that $W(\hat{\tau}) > W(\tau)$
    $\tau := \hat{\tau}$
END WHILE

How good is this algorithm?

- Note: Following the standard convention for Max-Sat, I am using approximation ratios < 1.
- It can be shown that for $d = 1$, the approximation ratio for Exact-Max-2-Sat is $\frac{2}{3}$.
- In fact, for every exact 2-Sat formula, the algorithm finds an assignment τ such that $W(\tau) \geq \frac{2}{3} \sum_{i=1}^{m} w_i$, the weight of all clauses, and we say that the "totality ratio" is at least $\frac{2}{3}$.
- (More generally for Exact Max-k-Sat the ratio is $\frac{k}{k+1}$). This ratio is essentially a tight ratio for any $d = o(n)$.
- This is in contrast to a naive greedy algorithm derived from a randomized algorithm that achieves totality ratio $(2^k - 1)/2^k$.
- "In practice", the local search algorithm often performs better than the naive greedy and one could always start with (for example) a greedy algorithm and then apply local search.

Analysis of the oblivious local search for Exact Max-2-Sat

- Let τ be a local optimum and let
  - S_0 be those clauses that are not satisfied by τ
  - S_1 be those clauses that are satisfied by exactly one literal by τ
  - S_2 be those clauses that are satisfied by two literals by τ

  Let $W(S_i)$ be the corresponding weight.

- We will say that a clause involves a variable x_j if either $x_j$or $\bar{x}_j$ occurs in the clause. Then for each j, let
  - A_j be those clauses in S_0 involving the variable x_j.
  - B_j be those clauses C in S_1 involving the variable x_j such that it is the literal x_j or $\bar{x}_j$ that is satisfied in C by τ.
  - C_j be those clauses in S_2 involving the variable x_j.

  Let $W(A_j), W(B_j), W(C_j)$ be the corresponding weights.

Analysis of the oblivious local search (continued)

- Summing over all variables x_j, we get
  - $2W(S_0) = \sum_j W(A_j)$ noting that each clause in S_0 gets counted twice.
  - $W(S_1) = \sum_j W(B_j)$

- Given that τ is a local optimum, for every j, we have

$$W(A_j) \leq W(B_j)$$

  or else flipping the truth value of x_j would improve the weight of the clauses being satisfied.

- Hence (by summing over all j),

$$2W_0 \leq W_1.$$

Finishing the analysis

- It follows then that the ratio of clause weights not satisfied to the sum of all clause weights is

$$\frac{W(S_0)}{W(S_0)+W(S_1)+W(S_2)} \leq \frac{W(S_0)}{3W(S_0)+W(S_2)} \leq \frac{W(S_0)}{3W(S_0)}$$

- It is not easy to verify but there are examples showing that this $\frac{2}{3}$ bound is essentially tight for any N_d neighbourhood for $d = o(n)$.

- It is also claimed that the bound is at best $\frac{4}{5}$ whenever $d < n/2$. For $d = n/2$, the algorithm would be optimal.

- In the weighted case, as in the max-cut problem, we have to worry about the number of iterations. And here again we can speed up the termination by insisting that any improvement has to be sufficiently better.

Using the proof to improve the algorithm

- We can learn something from this proof to improve the performance.
- Note that we are not using anything about $W(S_2)$.
- If we could guarantee that $W(S_0)$ was at most $W(S_2)$ then the ratio of clause weights not satisfied to all clause weights would be $\frac{1}{4}$.
- Claim: We can do this by enlarging the neighbourhood to include $\tau' =$ the complement of τ.

The non-oblivious local search

- We consider the idea that satisfied clauses in S_2 are more valuable than satisfied clauses in S_1 (because they are able to withstand any single variable change).

- The idea then is to weight S_2 clauses more heavily.

- Specifically, in each iteration we attempt to find a $\tau' \in N_1(\tau)$ that improves the potential function

$$\frac{3}{2}W(S_1) + 2W(S_2)$$

instead of the oblivious $W(S_1) + W(S_2)$.

- More generally, for all k, there is a setting of scaling coefficients $c_1, \ldots, c_k$, such that the non-oblivious local search using the potential function $c_1 W(S_1) + c_2 W(S_2 + \ldots + c_k W(S_k)$ results in approximation ratio $\frac{2^k-1}{2^k}$ for exact Max-k-Sat.

Sketch of $\frac{3}{4}$ totality bound for the non oblivious local search for Exact Max-2-Sat

- Let $P_{i,j}$ be the weight of all clauses in S_i containing x_j.
- Let $N_{i,j}$ be the weight of all clauses in S_i containing $\bar{x}_j$.
- Here is the key observation for a local optimum τ wrt the stated potential:

$$-\frac{1}{2}P_{2,j} - \frac{3}{2}P_{1,j} + \frac{1}{2}N_{1,j} + \frac{3}{2}N_{0,j} \leq 0$$

- Summing over variables $P_1 = N_1 = W(S_1)$, $P_2 = 2W(S_2)$ and $N_0 = 2W(S_0)$ and using the above inequality we obtain

$$3W(S_0) \leq W(S_1) + W(S_2)$$