CSC373: Algorithm Design, Analysis and Complexity
Winter/Spring 2020

Allan Borodin and Sara Rahmati

Week of March 16-20, 2020

Week 10 : Announcements

- As we all know, the University has cancelled classes until the end of term and the Faculty has cancelled final exams. In consultation with the Department of Computer Science, we have proposed two possible revised grading schemes which we will put to the class (in Quercus) for a majority vote of approval.
- We will provide information during this period on the web page, on piazza and on these lecture slides. We will mainly respond to general questions as above and not by individual emails. However, we will try to make ourselves as available as possible by email and even skype for questions that are more individual specific.
- Assignment 3 is now complete and still due Thursday, April 2 at 4:59. There will be an **individual, not team** Assignment 4 due April 15. We will soon post the start of assignment A4.
- We will continue to try to answer questions on Piazza as promptly as we can. In many cases, students are answering question correctly and we appreciate that being done.

This weeks agenda

- We will be discussing approximation algorithms.
- We'll first consider a couple of online and greedy algorithms.
- Then, as already suggested, we consider one general method to obtain an approximation algorithm (often used for *NP* hard problems); namley, we 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 exact and approximate algorithms.

The approximation algorithms agenda

- Why approximation algorithms?
- Strict and asymptotic approximation ratios.
- Simple online algorithms for the bin packing and makespan (for identical machines) problems.
- A simple online 2-approximation algorithm for the unweighted vertex cover problem. A greedy algorithm that does *not* work for the unweighted and weighted vertex cover problem.
- An IP formulation of the weighted vertex cover problem and its rounding to produce a 2-approximation. The integrality gap. (This has already been suggested.)
- 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.

Approximation algorithms agenda continued

- A greedy algorithm for axis-aligned weighted rectangles in 2-space. Another charging argument.
- Abstraction of the previous problem to $k + 1$ claw free graphs.
- A greedy 2-approximation algorithm for the unweighted JISP problem.
- Abstraction 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. (This has also already been suggested.
- The local search paradigm; oblivious and non-oblivious local search
- Local search for approximating max-cut and max-sat.

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.

But *NP*-hardness does not necessarily 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.

But *NP*-hardness does not necessarily 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.

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 an algorithm 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 $\lim_{Cost[OPT(\mathcal{I})] \to \infty} \frac{Cost[ALG(\mathcal{I})]}{Cost[OPT(\mathcal{I}]}$.
This is equivalent to
$Cost[ALG(\mathcal{I}))] \leq c \cdot Cost[OPT(\mathcal{I})] + o[Cost[OPT(\mathcal{I})]$

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.

For minimization problems, approximation ratios are always such that $c \geq 1$. For maximization problems, in the above definition, approximation ratios are a fraction of the optimal profit (i.e., as $\frac{1}{c}$ where $c \geq 1$. However, if you are reading about approximation ratios for a maximization problem, you will often also see claims that an algorithm ALG achieves (for example) has an approximation ratio $\frac{4}{3}$ in the sense that an optimalm solution cannot be better than a fraction $\frac{4}{3}$ of what is achieved by ALG.

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

The approximation landscape

- In terms 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 different 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 $H_n \approx \ln n$ is 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).

The 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 items $\{a_i\}$ 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 dating back to a seminal paper by Johnston et al [1974]. Why *NP* hard?

The 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 items $\{a_i\}$ 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 dating back to a seminal paper by Johnston et al [1974]. Why *NP* hard?
THe PARTITION problem is defined as follows: Given $A = \{a_1, a_2, \ldots, a_n\}$, is there a partition of A into $A_1 \cup A_2$ such that $\sum_{a_i \in A_1} = \sum_{a_i \in A_2}$. It can be shown that SUBSET-SUM $\leq_p$ PARTITION and hence the PARTITION problem is *NP* complete. Therefore, 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* "next-fit" opens a new bin whenever the last used bin does not have enough room for an arriving item. This is a 2-approximation algorithm.

Bin packing continued

The online "first fit" and "best fit" algorithms achieve strict approximation ratio $\frac{17}{10}$. First fit will try placing each item in the first bin in which it fits before opening a new bin if necessary. Best fit will try to place each item in the bin (if any) so as to minimize the remaining space before opening a new bin.

By *online algorithm* we mean that when each input item arrives, an irrevocable decision is made without seeing the remaining items. Most algorithms are not online and they can be called *offline algorithms*.

There are offline algorithms that for any input instance $\mathcal{I}$ pack the items in $OPT(\mathcal{I}) + o(OPT(\mathcal{I}))$*bins*. Hence the *asymptotic* approximation ratio of such an algorithms is 1.

Online and greedy algorithms

We recall that greedy algorithms consider the input items in some order determined by the algorithm, and for each input item makes 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).

In class, a natural question was asked. What is the approximation if we first sorted the bin packing items and then say using the First Fit algorithm. It has been shown that when sorting items so that $a_1 \geq a_2 \ldots \geq a_n$, the (asymptotic) approximation ratio is $\frac{11}{9}$ and after a sequence results the precise bound for FFD *first fit decreasing* is $FFD(\mathcal{I}) \leq \frac{11}{9} OPT(\mathcal{I}) + \frac{6}{9}$. In contrast, when sorting items so that $a_1 \leq a_2 \ldots \leq a_n$, the ratio is no better than $\frac{5}{3}$.

The makespan problem for identical machines

For our next approximation algorithm we consider the makespan problem on m identical machines. 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 also an NP hard optimization problem even for just $m = 2$ since an optimal algorithm for 2 machines would solve the NP cxomplete PARTITION decision problem.

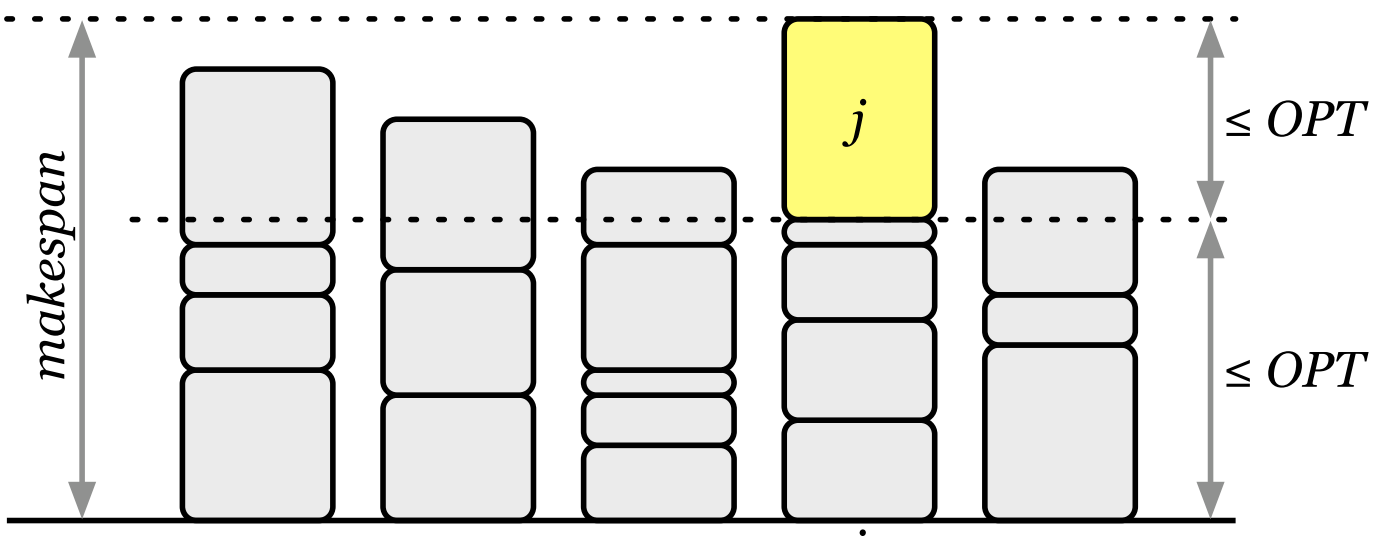
This problem was first studied by Graham [1966] where he analyzed the following "natural greedy algorith". This result precedes the introduction of NP compleness in 1971, and is considred to be the first analsysi of an approximaton ratio.

The makespan problem continued

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?



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 distribution.

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 maximal matching and a vertex cover must contain at least one vertex for each edge in a maximal matching. The algorithm is taking both vertices.

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 adds to the vertex cover by choosing the vertex adjacent to the most uncovered edges.

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 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.

The "natural" extension of the previous greedy algorithm for the weighted vertex cover problem

Although the previous algiorithm did not result in a constant approximation, it is still of interest to see if we can extend it to the weighted case in some natural way.

Greedy algorithm for weighted 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
    $\frac{w(v)}{current-deg(v)}$ is a minimum
  $V' = V' \cup \{v\}$;
  Remove all (v, u_i) from E'
End While

When $w(v) = 1$ for all v, this becomes the greedy algorithm for the unweighted cass . Thus the approximation cannot be better than $H_{d_{max}}$. As previously stated, vertex cover is a special case of set cover and the analogous greedy algorithm achieves approximation H_m ratio as we will see.

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 $w : V \to \Re^{\geq 0}$. 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. $\sum_j w_j \bar{x}_j \leq 2 \cdot \sum_j w_j x_j^* = 2 \cdot \mathit{LP\text{-}OPT} \leq 2 \cdot \mathit{IP\text{-}OPT}$; 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.

Lets consider a problem for which there is an optimal (DP) algorithm. Namely, lets consider a restricted version of the weighted interval scheduling problem (on one machine) where the weight (or valule) of any $I' = [s, f)$ interval is equal to its its length $f - s$.

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 Greedy that sorts by non-increasing weight (equal to or at most 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 I' 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 intervalis I in the greedy solution will be at most 3 times the weight of I'. (Remainder of proof on next slide.)
Note: In graph theoretic terms, we are approximating the maximum weighted independent set problem in an interval graph.

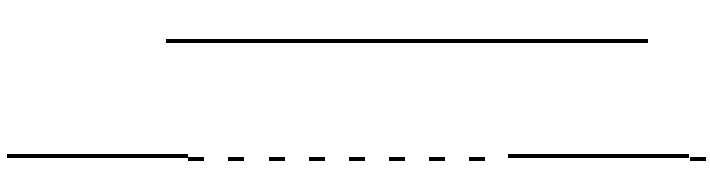
Completing the charging argument proof for proportional weight interval selection

Let I' be an interval in OPT. If I' is also in Greedy, then we charge I' to iteself. I' cannot intersect any other interval in Greedy since OPT has no conflicts. So now consider an I' in OPT that is not in Greedy. We charge I' to the leftmost interval I in Greedy that intersects I'. There must be at least one interval intersecting I' or Greedy would have taken it. Consider the intervals I in Greedy to which $I' = [s, f)$ has charged its weight $w(I')$. These intervals intervas I are such that:

- I' intersects I at s. There can be at most one such I' and $w(I') \leq w(I)$ or else Greedy would have taken I' before seeing I.
- I' intersects I at f. There can be at most one such I' and $w(I') \leq w(I)$ or else Greedy would have taken I' before seeing I.
- I' is contained in I. But the total weight of all such intervals is at most $w(I)$ since we have weights equal to length $f - s$.

This guarantees that the total weight of intervals I' in OPT is at most three times the weight of intervals in Greedy.

Visualizing the charging argument



It can be shown that for a reasonably general model of what we mean by a greedy algorithm, it can be shown that for every $\epsilon > 0$, for every greedy algorithm G, there is a set of weighted intervals $\mathcal{I}$ such that $OPT(\mathcal{I}) \geq (3 - \epsilon) \cdot G(\mathcal{I})$.

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 parallel 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 rectangles, 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 parallel 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 rectangles, 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 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 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 $O(n^3/\epsilon)$.
  - After a sequence of improvements the best time bound is $\tilde{O}(n + (\frac{1}{\epsilon})^{\frac{9)}{4}})$.