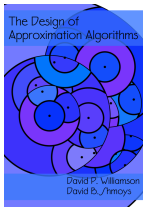


These lecture notes are for the course Advanced Algorithms (2016-2017) and are supplementary to the book 'The design of approxiation algorithms' by David Shmoys and David Williamson. See the book for missing definitions. These notes only cover a part of the sections of the book. If you find an error/typo please let me know (r.a.sitters@vu.nl) and I will update de file.

Contents

1 An introduction to approximation algorithms

1.1 Approximation Algorithms

Definition 1. *An α-approximation algorithm for an optimization problem is a polynomial-time algorithm that for all instances of the problem produces a solution for which the value is within a factor α of the value of an optimal solution.*

To show that an algorithm ALG is an α-approximation algorithm we need to show three things:

[1] The algorithm runs in polynomial time.

[2] The algorithm always produces a feasible solution.

[3] The value is within a factor of α of the value of an optimal solution.

1.2 Introduction to techniques of LP

The first chapter gives five different approximation algorithms for the weighted set cover problem. The so called vertex cover problem is the special case of the set cover problem in which each item appears in exactly two sets. The vertex cover problem is not explained separately in the book but it is helpful to consider this easier problem first. In this problem, we need to find for a given graph $G = (V, E)$ a subset of vertices such that each edge has an endpoint in the set. The goal is to minimize the number of vertices in the subset.

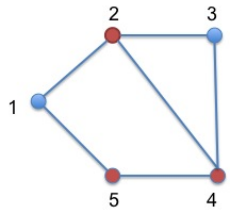


Figure 1: *The red vertices form a minimum vertex cover:* $I = \{2, 4, 5\}$.

VERTEX COVER:

Instance:	Graph $G = (V, E)$.
Solution:	$I \subseteq V$ such that each edge has at least one endpoint in I.
Cost:	$\|I\|$ (the number of vertices in I)
Goal:	Find a solution of minimum cost.

The Vertex Cover problem is $\mathcal{NP}$-hard. Thus, there is no polynomial time algorithm that solves the problem, unless $\mathcal{P} = \mathcal{NP}$. The next algorithm is a very simple 2-approximation. (Not in the book.)

Algorithm $\mathcal{A}$:
Find a maximal matching M and add all endpoints of the edges in M to I.

Theorem 1.1. *Algorithm $\mathcal{A}$ is a 2-approximation algorithm.*

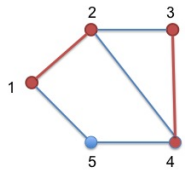


Figure 2: *The red edges form a maximal matching: $M = \{(1,2),(3,4)\}$. The solution given by the algorithm $\mathcal{A}$ is $I = \{1,2,3,4\}$.*

Proof. We need to prove three things: [1] the running time is polynomial, [2] the solution is feasible, [3] the value of the solution is at most 2 times the optimal value.
[1] A maximal matching can be found by choosing the edges one by one until no edges can be added anymore. [2] Assume edge e is not covered. Then we can add e to M and get a bigger matching. This is not possible since we assumed that M is maximal. [3] Since the edges in M have no endpoints in common, we have $|I| = 2|M|$. Also, any solution must use at least one point from each of the M edges, but since these edges are independent (have no endpoints in common), we must have $\textsc{Opt} \geqslant |M|$. We conclude that $|I| = 2|M| \leqslant 2\textsc{Opt}$. □

An algorithm by linear programming The vertex cover problem can easily be formulated as an integer linear programming problem (ILP). Let $m = |V|$ be the number of vertices[1].

$$\begin{aligned} \text{(ILP)} \quad \min \quad & Z = \sum_{j=1}^{m} x_j & \\ s.t. \quad & x_i + x_j \geqslant 1 & \text{for all } (i,j) \in E \\ & x_j \in \{0,1\} & \text{for all } j \in V. \end{aligned}$$

[1] Usually, we write $n = |V|$ for the number of vertices. Here we use m in stead of n in order to be consistent with the notation for Set Cover used in Chapter 1.

The vertex cover problem is $\mathcal{NP}$-hard which implies that the ILP above can not be solved in polynomial time, unless $\mathcal{P}$=$\mathcal{NP}$. However, the following LP-relaxation (in which $x_j \in \{0,1\}$ is replaced by $x_j \geq 0$) can be solved efficiently.

$$\begin{aligned} \text{(LP)} \ \min \quad & Z = \sum_{j=1}^{m} x_j & \\ s.t. \quad & x_i + x_j \geqslant 1 & \text{for all } (i,j) \in E \\ & x_j \geqslant 0 & \text{for all } j \in V. \end{aligned}$$

The idea of the algorithm is to solve (LP) and then *round* that solution in a feasible solution for (IP). This technique is called *LP-rounding*.

Algorithm $\mathcal{B}$:
Step 1: Solve the LP. $\rightarrow$ Optimal solution $x_1^*, x_2^*, \ldots, x_m^*$ with value Z_{LP}^*.
Step 2: Add j to solution I if $x_j^* \geqslant 1/2$.

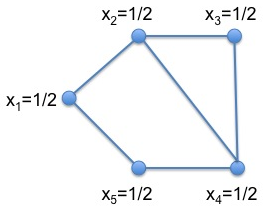


Figure 3: *In this example, an optimal LP-solution is $x_j^* = 1/2$ for all j. The value of the LP-solution is 2.5. Which is strictly smaller than the optimal ILP-solution which is 3. Algorithm $\mathcal{B}$ takes all five vertices: $I = \{1,2,3,4,5\}$ and has therefore value 5. The approximation ratio for this particular instance is therefore 5/3. In general, the ratio is never more than 2, as stated in Theorem 1.2.*

Theorem 1.2. *Algorithm $\mathcal{B}$ is a 2-approximation algorithm.*

Proof. We need to prove three things: [1] the running time is polynomial, [2] the solution is feasible, [3] the value of the solution is at most 2 times the optimal value.
[1]: True, since LP's can be solved in polynomial time. [2]: We need to show that every edge has an endpoint in I. Consider an arbitrary edge (i,j). We have $x_i^* + x_j^* \geqslant 1$. But then, either $x_i^* \geqslant 1/2$ or $x_j^* \geqslant 1/2$ (or both). That means that either i or j or both is added to I. [3] Denote the rounded solution by $\hat{x}$.

That means: $\hat{x}_j = 1$ if $x_j^* \geqslant 1/2$ and $\hat{x}_j = 0$ otherwise. In either case, $\hat{x}_j \leqslant 2x_j^*$. The value of the solution found is

$$|I| = \sum_{j=1}^{m} \hat{x}_j \leqslant 2 \sum_{j=1}^{m} x_j^* = 2Z_{LP}^* \leqslant 2Z_{ILP}^* = 2\textsc{Opt}.$$

In the equation above, Z_{ILP}^* is the optimal value of the integer linear program ILP. (It is common to add a * to indicate the optimal solution and optimal value, as we do here.) □

Weighted case Now consider the *weighted* vertex cover problem. In this case, each vertex j has a given weight $w_j > 0$ and the goal is to minimize the total weight of the cover. The analysis of algorithm $\mathcal{A}$ does not go through (Check this. Can you find an example where algorithm $\mathcal{A}$ gives a solution that is far from optimal?). However, the second algorithm, $\mathcal{B}$, does apply with only some minor changes: In the LP, there is only an extra term w_j and in the analysis there is only a small change in [3]

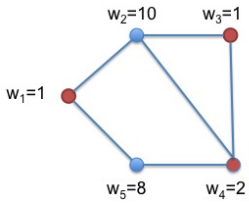


Figure 4: *Graph G with weights on the vertices is an instance of the weighted Vertex Cover problem. The optimal solution has total weight $1+1+2=4$.*

$$\begin{aligned} \text{(LP)} \quad \min \quad & Z = \sum_{j=1}^{m} w_j x_j & \\ s.t. \quad & x_i + x_j \geqslant 1 & \text{for all } (i,j) \in E \\ & x_j \geqslant 0 & \text{for all } j \in V. \end{aligned}$$

[3] The value of the solution found is

$$\sum_{j \in I} w_j = \sum_{j=1}^{m} w_j \hat{x}_j \leqslant 2 \sum_{j=1}^{m} w_j x_j^* = 2Z_{LP}^* \leqslant 2Z_{ILP}^* = 2\textsc{Opt}.$$

1.3 A deterministic rounding algorithm

The Set Cover problem is a generalization of the Vertex Cover problem which we discussed above.

SET COVER:

Instance:	Set of items (elements) $E = \{e_1, \ldots, e_n\}$, subsets $S_1, \ldots, S_m \subseteq E$, and weights $w_1, \ldots, w_m \geqslant 0$.
Solution:	$I \subseteq \{1, 2, \ldots, m\}$ such that each item is covered: $\bigcup_{j \in I} S_j = E$.
Cost:	Weight of the cover: $\sum_{j \in I} w_j$.
Goal:	Find a solution of minimum cost.

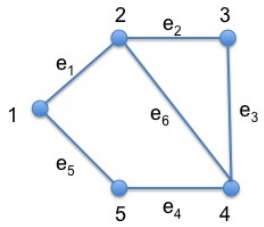


Figure 5: *Graph G is an instance of the Vertex Cover problem. Equivalently, we can write it as a Set Cover problem. For each vertex j there is a set S_j containing the adjacent edges: $S_1 = \{e_1, e_5\}$, $S_2 = \{e_1, e_2, e_6\}$, $S_3 = \{e_2, e_3\}$, $S_4 = \{e_3, e_4, e_6\}$, and $S_5 = \{e_4, e_5\}$.*

The vertex cover problem is the special case of the set cover problem in which each element e_i appears in at most 2 sets. Now consider the Set Cover problem and assume that each item appears in at most f sets, for some constant f. The LP-rounding algorithm for vertex cover problem applies here in the same way. This is the first algorithm given in the book.

$$\begin{aligned} \text{(ILP)} \quad \min \quad & Z = \sum_{j=1}^{m} w_j x_j \\ s.t. \quad & \sum_{j : e_i \in S_j} x_j \geqslant 1 \qquad \text{for all } i = 1, \ldots, n \\ & x_j \in \{0, 1\} \qquad \text{for all } j = 1, \ldots, m. \end{aligned}$$

The LP-relaxation is obtained by replacing $x_j \in \{0, 1\}$ by $x_j \geq 0$.

$$\begin{aligned} \text{(LP)} \quad \min \quad & Z = \sum_{j=1}^{m} w_j x_j \\ s.t. \quad & \sum_{j : e_i \in S_j} x_j \geqslant 1 \qquad \text{for all } i = 1, \ldots, n \\ & x_j \geqslant 0 \qquad \text{for all } j = 1, \ldots, m. \end{aligned}$$

Algorithm 1:
Step 1: Solve the LP. $\rightarrow$ Optimal values $x_1^*, x_2^*, \ldots, x_m^*, Z_{LP}^*$
Step 2: Add j to solution I if $x_j^* \geqslant 1/f$.

Theorem 1.3. *Algorithm 1 is an f-approximation algorithm for Set Cover.*

Proof. [1] Clearly, the running time is polynomial since LP's can be solved in polynomial time and the second step can be done in linear time. (We only need to check each x_j^* once.)
[2] Any solution produced by this algorithm is feasible since each item appears in at most f sets. That means, there are at most f variables in the constraint for e_i and at least one of the variables must has value $\geqslant 1/f$.
[3] Denote the rounded solution by $\hat{x}$. That means, $\hat{x}_j = 1$ if $x_j^* \geqslant 1/f$ and $\hat{x}_j = 0$ otherwise. In either case, $\hat{x}_j \leqslant f x_j^*$. The value of the solution found is

$$\sum_{j \in I} w_j = \sum_{j=1}^{m} \hat{x}_j w_j \leqslant f \sum_{j=1}^{m} w_j x_j^* = f Z_{LP}^* \leqslant f Z_{ILP}^* = f \textsc{Opt}.$$

□

1.4 Rounding a dual solution

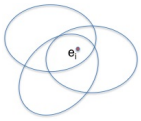
An alternative approach with the same approximation guarantee is obtained by using the dual of the LP for Set Cover. In the dual LP-formulation there is a variable y_i for each item e_i. The objective is to *maximize* the sum of the $y_i's$. The dual becomes:

$$\begin{aligned} \text{(D)} \ \max \quad & Z = \sum_{i=1}^{n} y_i & \\ s.t. \quad & \sum_{i:e_i \in S_j} y_i \leqslant w_j & \text{for all } j = 1, \ldots, m \\ & y_i \geqslant 0 & \text{for all } i = 1, \ldots, n. \end{aligned}$$

Algorithm 2:
Step 1: Solve the dual (D) $\rightarrow y_1^*, y_2^*, \ldots, y_n^*, Z_D^*$
Step 2: Add j to solution I if the dual constraint for S_j is *tight*: $\sum_{i:e_i \in S_j} y_i^* = w_j$.

Theorem 1.4. *Algorithm 2 is an f-approximation algorithm for Set Cover.*

Proof. [1] Clearly, the running time is polynomial.
[2] We need to show that any solution produced by this algorithm is feasible. Assume it is not, that means, some item e_i is not covered. But then, none of the dual constraints j for which $i \in S_j$ is tight. Hence, we can increase the value y_i^* by some small value and still maintain a feasible dual solution. This would increase the dual objective value, which contradicts the fact the dual solution is optimal.



[3] The value of the solution found is

$$\sum_{j \in I} w_j = \sum_{j \in I} \sum_{i: e_i \in S_j} y_i^* \leqslant f \sum_{i=1}^{n} y_i^* = f Z_D^* = f Z_{LP}^* \leqslant f \textsc{Opt}.$$

The first *equality* above follows since only tight sets S_j were picked by the algorithm: $\sum_{i: e_i \in S_j} y_i^* = w_j$. The first *inequality* follows from the fact that each of the y_i^*'s appears at most f times in the summation. The second inequality follows from strong duality. □

1.5 The primal-dual method

In the analyis of the previous section, we don't really need that y^* is an optimal dual solution. To prove property [3], optimality was used in the last inequality but it is enough to use only *weak* duality: $Z_D^* \leqslant Z_{LP}^*$. In [2], we only used that every element e_i is in at least one tight set S_j. We showed that this is true for the optimal solution y^* but this property can be obtained a lot easier than by solving (D). We simply start with $y_i = 0$ for all i and then increase the dual values one by one.

Algorithm 3:
Step 1: $I = \emptyset$, $y_i = 0$ for all i.
Step 2: As long as some e_i is uncovered do: increase y_i until some dual constraint l becomes tight and add **all** j to the solution I for which the corresponding dual constraint becomes tight.

First, note that the algorithm is well-defined. If element e_i is uncovered then none of the sets that contain e_i were picked sofar, which means that none of the corresponding constraints were tight. Then, y_i can be increased until some set becomes tight and if we then take all tight sets in the solution, e_i will be covered. Hence, the algorithm will always return a feasible solution.

NB. In Algorithm 1.1 in the book, only one set S_l is added in each iteration. For example, if two sets become tight at the same moment, then Algorithm 1.1 adds these to solution I in two consecutive iterations. In Algorithm 3 above we just add these two in one step. Further, the algorithms are the same. Consider

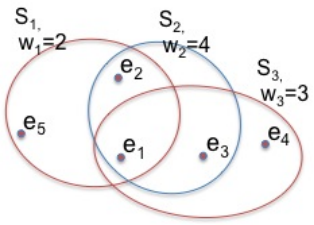


Figure 6: *The algorithm increases the y_i's in arbitrary order. Assume the order is $y_1, \ldots, y_5$. Then y_1 gets value 2. Next, y_2 cannot be increased (since it is in tight set S_2). Next y_3 can be increased to value 1. Next, y_4 and y_5 cannot be increased. The solution obtained by the algorithm is $(y_1, y_2, y_3, y_4, y_5) = (2, 0, 1, 0, 0)$. The tight sets are S_1 and S_3 and the value is $w_1 + w_3 = 5$. Choosing a different order to increase the y_i's may give another solution.*

the example $S_1 = \{e_1\}, S_2 = \{e_1, e_2\}$, and $w_1 = w_2 = 1$. The dual becomes

$$\begin{array}{lll} \text{(D) max} & Z = y_1 + y_2 & \\ s.t. & y_1 \leqslant 1 & \text{(for } S_1\text{)} \\ & y_1 + y_2 \leqslant 1 & \text{(for } S_2\text{)} \\ & y_1, y_2 \geqslant 0. & \end{array}$$

If the algorithms start with increasing y_1 then the algorithm of the book first takes S_1 in the solution and adds S_2 in the next iteration. Algorithm 3 takes both sets S_1, S_2 in the solution at once.

Theorem 1.5. *Algorithm 3 is an f-approximation algorithm.*

[1] The running time is even better than that of Algorithm 1 and 2 since we do not need to solve a linear program. It runs on $O(fn)$ time. [2] As we noted above, the algorithm always returns a feasible solution. [3] Let y_i denote the dual values obtained at the end of the algorithm and let Z_D be the corresponding objective value of the dual. Then, the value of the solution I is

$$\sum_{j \in I} w_j = \sum_{j \in I} \sum_{i : e_i \in S_j} y_i \leqslant f \sum_{i=1}^{n} y_i = f Z_D \leqslant f Z_D^* = f Z_{LP}^* \leqslant f \textsc{Opt}.$$

Note that the only difference with Algorithm 2 is that we do not necessarily have the optimal dual value. But that is no problem since $Z_D \leqslant Z_D^*$.

1.6 A greedy algorithm

The 4th. algorithm is a simple greedy algorithm. A *greedy* algorithm builds a solution step by step and at each step makes a greedy decission, that means, it extends the partial solution in a way that is, in some sense, the cheapest option. You will see more greedy algorithms in Chapter 2. Here, the greedy set cover algorithm picks in each step the set for which the cost per newly covered item is minimum.

The analysis given in the book is not so easy. The presentation of the proof here is slightly different and, hopefully, a bit easier than in the book. Instead of Fact 1.10 we use the following trivial observation.

Lemma 1. *Let $A_1, A_2, \ldots, A_q$ be finite sets. Then $\sum_j |A_j| \geqslant |\bigcup_j A_j|$.*

The greedy algorithm chooses sets in the solution one by one (in some order) until all items are covered. The next claim holds for any algorithm that chooses the sets one by one. Say that k_i is the number of uncovered items just before item e_i gets covered.

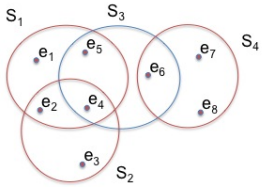


Figure 7: *In the unweighted case (all weights are 1), the algorithm picks the sets S_1, S_4, S_2 in this order. Then, $k_1 = k_2 = k_4 = k_5 = 8$, $k_6 = k_7 = k_8 = 4$, and $k_3 = 1$.*

Claim 1. $\sum_{i=1}^{n} 1/k_i \leqslant H_n$. *($H_n = \frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{1} \approx \ln n$ is the n-th harminic number.)*

Proof. Assume (without loss of generality) that the items are covered in the order $e_n, e_{n-1}, \ldots, e_1$. (Possibly, multiple items are covered at the same time but then we pick any order between these items.) Just before e_i gets covered there are at least i uncovered items. Hence, $k_i \geqslant i$.

$$\sum_{i=1}^{n} \frac{1}{k_i} \leqslant \sum_{i=1}^{n} \frac{1}{i} = H_n.$$

□

Unweighted case. The book only gives the algorithm for the weighted problem but it may be helpful to understand the unweighted version first. That algorithm always chooses the set with the largest number of uncovered elements. Let $\hat{S}_j$ be the set of uncoverd items in S_j during some state of the algorithm. Intially, $\hat{S}_j = S_j$ for all j and the set $\hat{S}_j$ is adjusted during the algorithm. At the end $\hat{S}_j = \emptyset$ for all j.

Algorithm 4: (unweighted)

Step 1: $I = \emptyset$, $\hat{S}_j = S_j$ for all j.

Step 2: As long as some e_i is uncovered, add that l to I for which $\hat{S}_l$ is maximal. Update $\hat{S}_j$ for all j.

Theorem 1.6. *Algorithm 4 is an H_n-approximation algorithm.*

Proof. The idea is to divide the cost for the solution over all items. When a new set S_l is added to the solution, then the cost for this set (which is 1) is divided over all newly covered items. Denote this cost per new item by y_i. Then,

$$y_i = \frac{1}{|\hat{S}_l|}.$$

The cost of the solution is exatly the sum of the y_i's.

$$|I| = \sum_{i=1}^{n} y_i.$$

Let Opt be the number of sets in the optimal solution.

Claim 2.

$$y_i \leqslant \frac{\text{Opt}}{k_i}.$$

Proof. Consider an arbitrary item e_i and assume it gets covered in Step 2 of the algorithm by set S_l. For any j, let $\hat{S}_j$ denote the uncovered items of S_j just before e_i gets covered. By defintion of the algorithm we have $|\hat{S}_l| \geqslant |\hat{S}_j|$ for all j (since the algorithm picks the set with the maxmimum number of uncovered items). Let I' be an optimal solution.

$$\text{Opt} = |I'| = \sum_{j \in I'} 1 \geqslant \sum_{j \in I'} \frac{|\hat{S}_j|}{|\hat{S}_l|} = \frac{1}{|\hat{S}_l|} \sum_{j \in I'} |\hat{S}_j| \geqslant \frac{1}{|\hat{S}_l|} k_i = y_i k_i.$$

The last inequality follows from Lemma 1: $\sum\limits_{j \in I'} |\hat{S}_j| \geqslant |\bigcup\limits_{j \in I'} \hat{S}_j| = k_i$. □

Combining Claims 1 and 2 we see that the total cost of the solution is

$$|I| = \sum_{i=1}^{n} y_i \leqslant \textsc{Opt} \sum_{i=1}^{n} \frac{1}{k_i} \leqslant \textsc{Opt} \cdot H_n.$$

□

Weighted case. Now, the weighted version. The changes are minimal.

Algorithm 4: (weighted)
Step 1: $I = \emptyset$, $\hat{S}_j = S_j$ for all j.
Step 2: As long as some e_i is uncovered, add that l to I for which $w_l/\hat{S}_l$ is minimal.

Theorem 1.7. *Algorithm 4 gives an H_n-approximation.*

Proof. Again, the cost for each added set is divided over the newly covered items. Denote this cost by y_i. If e_i gets covered by S_l then,

$$y_i := \frac{w_l}{|\hat{S}_l|}. \tag{1}$$

The cost of the solution is exatly the sum of the y_i's.

$$\sum_{j \in I} w_j = \sum_{i=1}^{n} y_i.$$

Claim 3.

$$y_i \leqslant \frac{\textsc{Opt}}{k_i}.$$

Proof. Consider an arbitrary item e_i and assume it gets covered in Step 2 of the algorithm by set S_l. For any j, let $\hat{S}_j$ denote the uncovered items of S_j just before e_i gets covered. By defintion of the algorithm we have $\frac{w_l}{|\hat{S}_l|} \leqslant \frac{w_j}{|\hat{S}_j|}$, for all j with $|\hat{S}_j| > 0$. This implies that

$$w_j \geqslant \frac{w_l}{|\hat{S}_l|} |\hat{S}_j|, \text{ for all } j.$$

Let I' be an optimal solution. We get that

$$\textsc{Opt} = \sum_{j \in I'} w_j \geqslant \frac{w_l}{|\hat{S}_l|} \sum_{j \in I'} |\hat{S}_j| \geqslant \frac{w_l}{|\hat{S}_l|} k_i = y_i k_i.$$

Again, the last inequality follows from Lemma 1: $\sum\limits_{j \in I'} |\hat{S}_j| \geqslant |\bigcup\limits_{j \in I'} \hat{S}_j| = k_i$. □

Combining Claims 1 and 3 we see that the total cost of the solution is

$$\sum_{i=1}^{n} y_i \leqslant \textsc{Opt} \sum_{i=1}^{n} \frac{1}{k_i} \leqslant H_n \textsc{Opt}.$$

□

Alternative analysis. (Dual fitting) By using the dual LP formulation of the problem one can prove an even better bound. Consider an arbitrary set S_j. To simplify the notation assume that $S_j = \{e_1, e_2, \ldots, e_{|S_j|}\}$. Assume that they are covered in opposite order: $e_{|S_j|}, \ldots, e_1$.

Now consider an arbitrary item e_i, $i \leqslant |S_j|$. Just before item e_i gets covered, there are at least i items uncovered in S_j, i.e., $|\hat{S}_j| \geqslant i$. Assume that e_i gets covered by set S_l. The algorithm also has the option to choose set S_j. That means

$$y_i := \frac{w_l}{|\hat{S}_l|} \leqslant \frac{w_j}{|\hat{S}_j|} \leqslant \frac{w_j}{i}.$$

Let $g = \max_j |S_j|$. Then,

$$\sum_{i:e_i \in S_j} y_i \leqslant \sum_{i=1}^{|S_j|} \frac{w_j}{i} = w_j H_{|S_j|} \leqslant w_j H_g.$$

We see from the inequality above that y satisfies the constraint of the dual-LP (D) up to a factor H_g. If we let $y'_i = y_i / H_g$, then the solution y'_i is feasible for the dual. The cost of the solution of the greedy algorithm is

$$\sum_{i=1}^{n} y_i = H_g \sum_{i=1}^{n} y'_i \leqslant H_g Z^*_D = H_g Z^*_{LP} \leqslant H_g \textsc{Opt}.$$

Note that this is better (or at least not worse) than the bound $H_n \textsc{Opt}$ since $H_g \leqslant H_n$.

1.7 A randomized rounding algorithm

Algorithm 1 uses LP-rounding to get an f-approximation. The rounding is deterministic in the sense that a variable with value $\geqslant 1/f$ is *always* rounded to 1. An natural variant is to round a variable x_j to 1 with a probability that depends on its value x^*_j. For example, let X_j be a stochastic variable with $\Pr(X_j = 1) = x^*_j$ and $\Pr(X_j = 0) = 1 - x^*_j$. Then (by linearity of expectations) the expected value of the solution is exactly the optimal LP-value (See book). However, the probabality that the solution is feasible is almost 0. In order to

increase this probability, the book suggests the following algorithm:

First, solve the LP-relaxation. Then, repeat the rounding K times. If the variable X_j is set to one at least once, then the set S_j is taken in the solution. It turns out that taking $K = c \ln n$ is large enough to get a feasible solution with high probability. The analysis can be found in the book.

Let us consider a slightly different approach here. The analysis is simpler.

Algorithm 5:
Step 1: Solve the LP $\rightarrow x_1^*, x_2^*, \ldots, x_n^*, Z_{LP}^*$
Step 2: Add j to solution I with probability $\min\{1, Kx_j^*\}$.

Theorem 1.8. *For $K = c \ln n$ and $c \geqslant 2$, Algorithm 5 finds a $2c \ln n$-aproximation with high probability.*

Proof. Consider an arbitrary element e_i. If $Kx_j^* \geqslant 1$ for some j with $e_i \in S_j$, then e_i will be covered by S_j. Otherwise,

$$\Pr(e_i \text{ not covered}) = \prod_{j:e_i \in S_j} (1 - Kx_j^*) \leq \prod_{j:e_i \in S_j} e^{-Kx_j^*} = e^{-\sum_{j:e_i \in S_j} Kx_j^*} \leqslant e^{-K}.$$

The first inequality follows from $1 - x \leqslant e^{-x}$ for all x. The second inequality follows from the LP-constraint $\sum_{j:e_i \in S_j} x_j^* \geqslant 1$. If we take $K = c \ln n$, then $e^{-K} = n^{-c}$. By the union bound, the probaility that *some* element e_i is not covered is at most n times as large, i.e., at most n^{-c+1}. If we take $c \geqslant 2$, then the probaility that the solution is feasible is at least

$$1 - \frac{1}{n^{c-1}} \geqslant 1 - \frac{1}{n}.$$

This is called *with high probability*. Sofar, the analysis has been the same as for the algorithm in the book. The computation for the expected value is a lot easier though. The expected value of the solution is

$$\sum_{j=1}^{m} \Pr(j \in I) w_j \leqslant \sum_{j=1}^{m} K x_j^* w_j = K Z_{LP}^*.$$

Note, however, that this expectation is also over the events that the solution is infeasible. We would like to know the expected value, given that the solution is feasible. Let F denote the event that the solution is feasible and let *Cost* denote

the cost of the solution by the algorithm. Then,

$$\begin{aligned}\mathbb{E}[\mathit{Cost}] &= \mathbb{E}[\mathit{Cost} \mid F]\Pr(F) + \mathbb{E}[\mathit{Cost} \mid \text{not } F]\Pr(\text{not } F) \\ &\geq \mathbb{E}[\mathit{Cost} \mid F]\Pr(F).\end{aligned}$$

For $n \geqslant 2$, we have $\Pr(F) \geqslant (1 - 1/n) \geqslant 1/2$. In that case, we get

$$\mathbb{E}[\mathit{Cost} \mid F] \leqslant 2\mathbb{E}[\mathit{Cost}] \leqslant 2KZ^*_{LP} \leqslant 2K\textsc{Opt} = 2c\ln n\textsc{Opt}.$$

□

2 Greedy and Local search

See the book for an introduction of this Chapter.

2.1 Single machine scheduling

See the book for a formal description of the optimization problem of this section.

Say that at job j is *available* at time t if $r_j \geq t$ and job j has not been scheduled yet. Say that the machine is *idle* at time t if no job is processed at time t.

Earliest Due Date (EDD) algorithm: At any moment that the machine is idle, start the job that has the earliest due date among the jobs that are available at that moment.

Theorem 2.1. *EDD is a 2-apprixmation algorithm for the scheduling problem in this section.*

The proof of this theorem given in te book can be simplified and the result made more general. Here, we only present this simplified version. Compare it with the proof in the book and pick your favorite version. EDD is a greedy type of algorithm since it always makes the choice that seems best at the moment. Let's consider a wider class of algorithms, which we simply call *Greedy* too. Clearly, EDD is greedy by this definition.

Greedy algorithm: At any moment that the machine is idle, start *some* job among the jobs that are available at that moment.

Theorem 2.2. *Any greedy algorithm gives a 2-approximation for the scheduling problem in this section.*

Proof. Consider an optimal schedule and let $L^*_{\max}$ and $C^*_{\max} = \max_j C^*_j$ denote, respectively, its maximum lateness and length. Since due dates d_j are negative and completion times C_j are positive we have

$$\begin{aligned} L^*_{\max} &= \max_j (C^*_j - d_j) \geq \max_j C^*_j = C^*_{\max} \text{ and} \\ L^*_{\max} &= \max_j (C^*_j - d_j) \geq \max_j (-d_j). \end{aligned} \tag{2}$$

Consider a schedule produced by a greedy algorithm and let $L_{\max}$ and $C_{\max} = \max_j C_j$ denote its maximum lateness and its length. Clearly, a greedy

algorithm always produces a schedule of minimal length. Hence, $C_{\max} \leqslant C^*_{\max}$. Together with (2) this implies

$$L_{\max} = \max_j (C_j - d_j) \leqslant C_{\max} + \max_j (-d_j) \leqslant C^*_{\max} + \max_j (-d_j) \leqslant 2L^*_{\max}.$$

□

2.2 The k-center problem.

The proof in the book for the k-center problem is short and clear. Nevertheless, here is a summary that may be helpful.

Algorithm Greedy
Pick the first center arbitrarily. Next, choose the centers one by one until k centers are chosen, and always choose the point that is furthest away from the set of centers already chosen. Let S be the chosen centers.

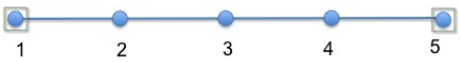


Figure 8: *It is easy to see that Greedy may be factor 2 away from the optimal value. Take this instance with $k = 2$ and 5 equidistant points on the line. The algorithm might choose points* 1 *and 5 while it is optimal to pick 2 and 4.*

Theorem 2.3. *Algorithm Greedy is a 2-approximation algorithm.*

Proof. Clearly, the algorithm runs is polynomial time and always returns a feasible solution. It remains to prove that the value is always at most twice the optimal value. Let S^* be an optimal set of centers and let r^* be its value. Each center in S has a center in S^* at distance $\leqslant r^*$. Now, connect each center from S with its nearest center from S^*.

There are two possible cases. (i) Each center from S^* is connected to exactly one center from S. (ii) There is a center $j_i \in S^*$ which is connected to at least two centers, say j, j', from S.

Case (i). Consider an arbitrary point $v \in V$. Then v has a center in S^* at distance at most r^*. This center is connected to some center in S at distance is at most r^*. By the triangle inequality, v has a center in S at distance $\leqslant 2r^*$.

Case (ii) First note that by the triangle inequality $d(j, j') \leqslant d(j, j_i) + d(j_i, j') \leqslant 2r^*$. Assume that the algorithm chose j' before point j. When

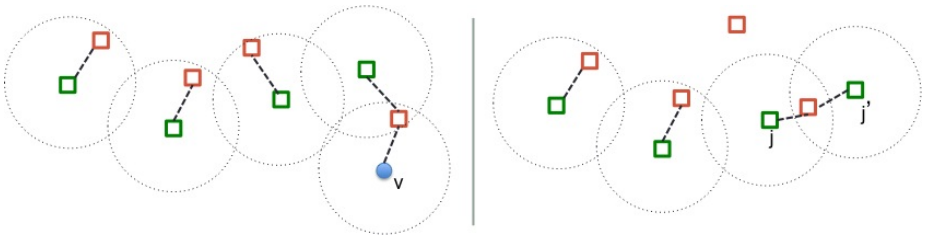


Figure 9: *Greedy centers (S) are green and the optimal centers (S^*) are red. Left: Each center from S^* is connected to exactly one center from S. Right: There is a center in S^* which is connected to more than one center from S.*

the algorithm picked j, this was the point with maximum distance to the chosen centers. This distance was at most $d(j, j') \leqslant 2r^*$. Therefore, any point has a distance to S of at most $2r^*$. □

By a reduction from the Dominating Set problem, it follows that there is no α-approximation algorithm for any $\alpha < 2$ possible, unless $\mathcal{P} = \mathcal{NP}$. See book.

2.3 Parallel machine scheduling.

In this scheduling problem, we are given n jobs and m machines and the goal is to assign each job to a machine such that the maximum machine load is minimized. (The *load* of a machine is the sum of the processing times of all jobs on the machine.) The problem is also known as *load balancing*.

LOAD BALANCING:

Instance:	n jobs with processing times $p_1, \ldots, p_n$ and m machines.
Solution:	An assignment of jobs to machines
Cost:	The maximum load over the machines
Goal:	Minimize cost.

The section gives two 2-approximation algorithms for the load balancing problem and one 4/3-approximation.

Algorithm 1 Local search
Start with an arbitrary schedule. If the completion time of some job j can be reduced by putting it at the end of some other machine i, then add j at the end of machine i. (To ensure polynomial running time, we let i be the machine with smallest load at that moment.) Continue this until no job can improve its completion time this way.

Algorithm 2 List scheduling
Place the jobs in arbitrary order. Following this order, place the jobs one by one on the machines, always scheduling the jobs as early as possible.

Algorithm 3 Longest Processing Time first rule (LPT)
Order jobs by their processing time ($p_1 \geqslant p_2 \geqslant \cdots \geqslant p_n$) and apply the list scheduling algorithm with this order.

Analysis of the algorithms There are two easy lower bounds on the length C^*_{max} of the optimal schedule.

$$C^*_{\max} \geqslant \max_j p_j \text{ and } C^*_{\max} \geqslant \frac{1}{m}\sum_j p_j. \tag{3}$$

The first says that the length of the schedule is at least the length of the longest job. The second bound says that the length of the schedule is at least the average machine load.

Clearly, Algorithms 2 and 3 run in polynomial time. However, this is not so clear for the local search algorithm. For example, if the completion time of the moving job is reduced by 1 unit in every step, then the running time could in principle be as large as $\Omega(\sum_j p_j)$, which is only pseudo polynomial time. If we always move the job to the machine with smallest load then polynomial running time is guaranteed. Oberve that the following is true.

> At any step of the local search algorithm, the smallest machine load does not decrease.

(See Figure 2.3 in the book.) This implies that any job can move at most once. (For a job to move a second time, the destination machine should have a smaller load than that of the destination machine in the first move.) Now, since each job moves at most once, the number of iterations is at most n.

Theorem 2.4. *Algorithms 1 and 2 are 2-approximation algorithms.*

Proof. It remains to prove the approximation factor. Let $C_{\max}$ be the length (makespan) of the schedule of either of the two algorithms. Let l be the job that completes last. See Figure 10. That means $C_l = C_{\max}$. Let S_l be the start time of job l in the schedule. Then, no machine has a load that is less than S_l since otherwise the list scheduling algorithm could have started l earlier and the local search algorithm would have moved l to another machine. Using the two

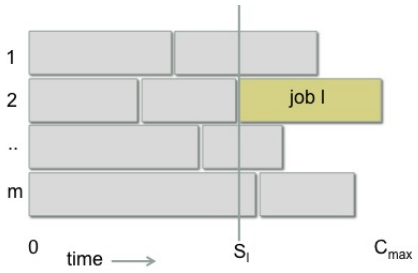


Figure 10: *Proof of the approximation factor. Job l is the last job to complete.*

lower bounds from (3) we obtain:

$$\begin{aligned} C_{\max} &= S_l + p_l \leqslant \frac{1}{m}\sum_{j\neq l} p_j + p_l = \frac{1}{m}\sum_j p_j + (1-\frac{1}{m})p_l \\ &\leqslant C^*_{\max} + (1-\frac{1}{m})p_l \leqslant (2-\frac{1}{m})C^*_{\max}. \end{aligned}$$

□

Theorem 2.5. *LPT is a 4/3 approximation algorithm.*

Proof. Polynomial running time and feasibility are obvious. It remains to show that for any instance, the length of the LPT schedule is at most 4/3 times the optimal length. Clearly this is true if the instance has only one job. Now assume that it holds for any instance with at most $n' < n$ jobs. Consider an instance I of n jobs and let σ be the LPT schedule and denote its length by $C_{\max}$. Let l be a job that completes last. Consider two cases: (i) $l < n$ and (ii) $l = n$.

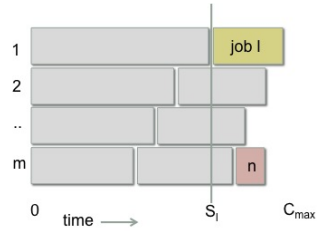


Figure 11: *LPT gives a 4/3-approxiamtion.*

Case (i). Remove job n from schedule σ. Denote the remaining schedule by σ'. Then, the length of σ' is equal to the length of σ, since job n was added last but is not the (only) job that completes last. Let I' be the instance consisting

of the jobs $1, \ldots, n-1$. Note that σ' is exactly the schedule that you get by applying LPT to I'. By induction, $C_{\max}$ is at most $4/3$ times the optimal length of I', which is at most $4/3$ times the optimal length of I.

Case (ii) Let S_n be the start time of job n. From the list scheduling analysis we know that

$$C_{\max} \quad = \quad S_n + p_n \leq \frac{1}{m}(\sum_j p_j) + p_n \leqslant C^*_{\max} + p_n.$$

If $p_n \leqslant C^*_{\max}/3$ then we are done. Now assume that $p_n > C^*_{\max}/3$. Then, $C^*_{\max} < 3p_n$. Since p_n is the smallest processing time this implies that the optimal schedule has at most 2 jobs per machine. It follows from Exercise 2.2 that the LPT schedule is optimal in that case, i.e., $C_{\max} = C^*_{\max}$. □

2.4 The traveling salesman problem.

TSP (SYMMETRIC):

Instance:	Complete graph with a cost c_{ij} for every pair i, j.
Solution:	A cycle that goes through each point exactly once
Cost:	The length (sum of the edge costs) of the cycle.
Goal:	Find a solution of minimum cost.

In the *symmetric* TSP, the costs c_{ij} and c_{ji} are the same. In the *asymmetric* TSP, the cost c_{ij} may be different from c_{ji}. In that case, the cost depends on the direction in which the edge is traversed. For both versions, one usually considers the *metric* version, which means that the triangle inequality holds: $c_{ik} \leq c_{ij} + c_{jk}$ for every triple i, j, k.

Theorem 2.6. *The Hamiltonian Cycle problem is reducable to the TSP problem.*

Proof. Given an instance $G = (V, E)$ of HC, form an instance of TSP by defining $c_{ij} = 1$ for all edges $(i, j) \in E$ and $c_{ij} = 2$ for all $(i, j) \notin E$. If there is no HC in G, then any TSP should use at least one of the edges of length 2. The other edges on the tour have length at least 1. The total length of the optimal TSP tour is at least $n + 1$. Hence,

$$\begin{array}{lcl} G \text{ has a HC} & \Rightarrow & \textsc{Opt}_{TSP} = n. \\ G \text{ has no HC} & \Rightarrow & \textsc{Opt}_{TSP} \geq n + 1. \end{array}$$

We conclude that G has a HC if and only if $\textsc{Opt}_{TSP} = n$. □

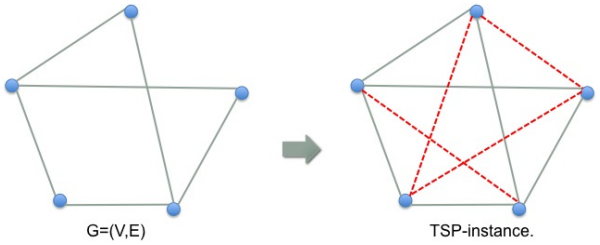


Figure 12: *An instance $G = (V, E)$ of the Hamiltonian Cycle problem and the corresponding TSP instance. Graph G has no Hamiltonian Cycle. Thus, any Hamiltonian Cycle in the graph to the right uses at least one of the dotted (red) edges.*

The Hamiltonian Cycle problem was one of the first problems shown to be NP-complete (Karp, 1972). It followed immediately (from the reduction above) that TSP is NP-complete (NP-hard) too. If we do not assume the triangle inequality, the same reduction shows a stronger hardness result.

Theorem 2.7. *For TSP without the triangle inequality assumption, there does not exist an α-approximation algorithm for any $\alpha \geqslant 1$, provided $\mathcal{P} \neq \mathcal{NP}$.*

Proof. We follow the same proof as for Theorem 2.6 but instead of taking a cost 2 for the missing edges we take a much larger cost: αn. Assume there exists an α-approximation algorithm for some $\alpha \geq 1$. We show that such an algorithm can be used to solve the Hamiltonian Cycle (HC) problem in polynomial time. If there is no HC in G, then any TSP should use at least one of the edges of length αn. The other edges on the tour have length at least 1. The total length of the optimal TSP tour is at least $\alpha n + (n-1) \geq \alpha n + 1$ (for $n \geq 2$). Hence,

$$\begin{array}{lcll} G \text{ has a HC} & \Rightarrow & \textsc{Opt}_{TSP} = n & \Rightarrow \quad \textsc{Alg} \leq \alpha n. \\ G \text{ has no HC} & \Rightarrow & \textsc{Opt}_{TSP} \geq \alpha n + 1 & \Rightarrow \quad \textsc{Alg} \geq \alpha n + 1. \end{array}$$

We conlude that G has a HC if and only if the value Alg of the solution given by the algorithm is at most αn. □

Algorithm 1 (Nearest addition).
Pick an arbitrary point i_1 and let i_2 be the point nearest to i_1. Make a directed tour from i_1 to i_2 and back to i_1. Let $S = \{i_1, i_2\}$.
Repeat the following until a feasible tour is found:
Find a pair $i \in S, j \notin S$ with minimum cost c_{ij}. (In other words, find the point

j that is nearest to the already chosen set S.) Insert j in the tour after i. Add j to S.

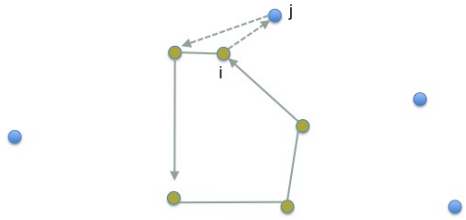


Figure 13: *Iteration of the nearest addition algorithm.*

Algorithm 2 (Double tree). Find a minimum spanning tree T. Double all the edge of the tree. Find an Euler tour in the double tree. Apply shortcutting in order to turn the Euler tour into a Hamiltonian cycle.

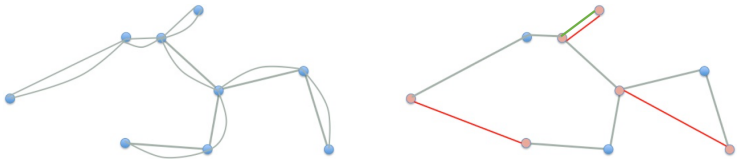


Figure 14: *The double tree (left) and the MST plus a matching of the odd-degree nodes (right).*

Algorithm 3 (Christofides' algorithm (1976)). Find a minimum spanning tree T. Let O be the vertices of odd degree in T. Find a minimum cost perfect matching of the vertices in O. Denote the edges in this matching by M. Find an Euler tour in the graph $T + M$. Apply shortcutting in order to turn the Euler tour into a Hamiltonian cycle.

Note that a perfect matching on O exists since $|O|$ is even. (Any graph contains an even number of odd-degree points.) Also note that the cheapest perfect matching can be found in polynomial time. (See for example book [1].)

Lemma 2. *Let T be a minimum spanning tree and* Opt *be the length of the shortest TSP tour. Then* $\text{Cost}(T) \leq \text{Opt}$.

Proof. Removing an arbitrary edge from the optimal TSP tour gives a spanning tree. □

Theorem 2.8. *Nearest addition is a 2-approximation algorithm.*

Proof. Consider edge (i_1, i_2) and all the pairs (edges) (i, j) chosen by the algorithm. Together they form a minimum spanning tree since the algorithm behaves exactly like Prim's algorithm to find a minimum spanning tree.

Consider an arbitrary iteration of the algorithm. Assume that pair (i, j) was picked in that step. That means j was the point nearest to S and this distance is c_{ij} with $i \in S$. Let k be the point that follows i in the tour constructed so far. By the triangle inequality $c_{jk} \leq c_{ji} + c_{ik} \Rightarrow c_{jk} - c_{ik} \leq c_{ji}$. Inserting j in the tour increases its length (cost) by at most

$$c_{ij} + c_{jk} - c_{ik} \leq 2c_{ij}.$$

We conclude that the length of the tour is at most twice the length of the minimum spanning tree. Now the proof follows from Lemma 2 □

Theorem 2.9. *Double tree is a 2-approximation algorithm.*

Proof. The length of the tour before shortcutting is exactly twice the length of the MST, which is at most twice Opt. The shortcutting step at the end does not increase the length of the tour (since the triangle inequality holds). □

Theorem 2.10. *Christofides' algorithm is a* 3/2*-approximation algorithm.*

Proof. Consider an optimal tour. Now shortcut this tour to get a tour on O. The length of this tour is at most Opt (by the triangle inequality). This tour is composed of exactly two perfect matchings on O. Since the algorithm computes the cheapest perfect matching, its cost is no more than Opt/2.

$$\mathit{Cost}(T) + \mathit{Cost}(M) \leq \frac{3}{2}\textsc{Opt}.$$

The shortcutting step at the end of the algorithm does not increase the length of the tour (since the triangle inequality holds). □

Christofides' algorithm is the best approximation algorithm known so far (in terms of approximation ratio) for the metric TSP. Note that the gap with the lower bound (1.0045, see book) is quite large. (Compare this for example with the k-center problem for which the upper and lower bound are both 2.)

3 Rounding data and dynamic programming

The title of this chapter might as well be 'Designing polynomial time approximation schemes'. The two main techniques for such approximation schemes are *rounding data* and *dynamic programming*.

Definition 2. *A polynomial-time approximation scheme (PTAS) is a family of algorithms A_ϵ, where there is an algorithm for each $\epsilon > 0$, such that A_ϵ is a $(1+\epsilon)$-approximation algorithm (for minimization problems) or a $(1-\epsilon)$-approximation algorithm (for maximization problems).*

It is important to note that the running time depends on ϵ. For example, a PTAS might have a running time of $O(n^{1/\epsilon})$. Then, for $\epsilon = 0.01$, its running time is $O(n^{100})$, which is not very practical but still polynomial. Polynomial time approximation schemes are interesting in theory since it learns us something about the complexity of the problem: If there is PTAS then there is no lower bound on the approximation ratio. Conversely, for the k-center problem we showed that no algorithm can have a smaller approximation ratio than 2. Thus, for this problem there does not exist a PTAS. (Assuming $\mathcal{P} \neq \mathcal{NP}$).

Overview of the sections The three sections give PTAS's for the optimization problems: knapsack, scheduling, and bin packing. With each section the algorithm becomes more difficult. Polynomial time approximation schemes usually contain many technical details but the main ingredients are often standard. Rounding data and dynamic programming are two standard techniques.

- Rounding input data is often used to turn an instance into a *rounded* instance in which all numbers are polynomially bounded. For example, in the knapsack problem of Section 3.1 we need to select a subset of the items of maximum value under some packing constraint. Items that have a very small value may be deleted from the instance if their total value is only an ϵ fraction of the optimal value. We are left with an instance where the ratio of the largest and the smallest number (value) is polynomially bounded (in this case, bounded by n/ϵ). Then, by scaling and rounding we may assume that all numbers are polynomially bounded integers.

  In the scheduling problem of Section 3.2, the numbers are partitioned into large and small numbers. In this case, *long* and *short* jobs. The short jobs are not deleted from the instance (as we do for knapsack) but are scheduled in a greedy way. The trick is to make this partition such that the number of long jobs is small enough to solve the problem on this restricted instance

efficiently while, at the same time, the short jobs are small enough such that it is fine to schedule these in a greedy (list scheduling) way.

- Dynamic programming is used to find an optimal solution for the simplified instance. For knapsack that is the rounded instance, for scheduling that is the instance restricted to the long jobs. In principle, any algorithm that solves these problems in polynomial time can be used, but dynamic programming is often the straightforward approach.

3.1 The knapsack problem

KNAPSACK:

Instance:	A set of items $I = \{1, 2 \ldots, n\}$, a capacity B and for each item i a value v_i and size $s_i \leq B$
Solution:	$S \subseteq I$ such that $\sum_{i \in S} s_i \leq B$.
Value:	$\sum_{i \in S} v_i$
Goal:	Find solution of maximum value.

The knapsack problem is $\mathcal{NP}$-hard but not *strongly* $\mathcal{NP}$-hard since it can be solved in *pseudo polynomial* time. The algorithm is a dynamic program. This DP is used later to obtain a PTAS. The DP works as follows:

A_j: the set of all pairs (t, w) such that there is a subset of items in $\{1, \ldots, j\}$ with size exactly $t \leq B$ and value exactly w.

By this definition, $A_1 = \{(0, 0), (s_1, v_1)\}$. For $j \geq 2$, set A_j can easily be computed from A_{j-1}. The pair (t, w) that we are looking for is the one in A_n with maximum value w.

A pseudopolynomial time Dynamic Program:
$A_1 = \{(0, 0), (s_1, v_1)\}$

For $j = 2$ to n do:

(1) $A_j \leftarrow A_{j-1}$ (Add all pairs in A_{j-1} to A_j)

(2) For each $(t, w) \in A_{j-1}$ do: If $t + s_j \leq B$ then add $(t + s_j, w + v_j)$ to A_j.

The logic behind the first step follows directly from the definition of A_j above: If $(t, w) \in A_{j-1}$ then also $(t, w) \in A_j$ since we have the option not to include item j. The second step corresponds with adding item j if it still fits in

the knapsack. The number of pairs (t, w) in each set A_j is no more than BV, where $V = \sum_{i=1}^{n} v_i$. Hence, the total running time of this dynamic program is $O(nBV)$. The algorithm does not give us an optimal solution but only the optimal value. A solution is easily obtained by storing for each $(t, w) \in A_j$ in the DP one corresponding subset of items.

Lemma 3. *The knapsack problem can be solved (exactly) in $O(nBV)$ time.*

Improved running time We can improve the running time by adding an extra step (3) in which dominated pairs are removed. A pair (t, w) dominates another pair (t', w') if $t \leq t'$ and $w \geq w'$. By removing each pair that is dominated by another pair, the number of pairs in A_j is at most $\min\{B, V\}$. Now, step (1) and (2) and (3) can be done in $O(\min\{B, V\})$ time in each iteration.

Theorem 3.1. *The knapsack problem can be solved (exactly) in $O(n \cdot \min\{V, B\})$ time.*

To obtain the PTAS, we apply the DP above to a rounded instance. Let

$$M = \max_i v_i, \text{ and } \mu = \frac{\epsilon M}{n}, \text{ and round values to } v'_i = \lfloor v_i/\mu \rfloor.$$

The maximum value in the rounded instance is $\lfloor M/\mu \rfloor = \lfloor n/\epsilon \rfloor$. Hence, in the rounded instance all v'_i are in $\{0, 1, \ldots, \lfloor n/\epsilon \rfloor\}$.

A PTAS for knapsack:

(1) Round the values as above: $v_i \to v'_i$

(2) Apply the DP to the rounded instance.

The running time of th DP is $O(nV')$ time where $V' = \sum_i v'_i = O(n^2/\epsilon)$. Hence the running time is $O(n^3/\epsilon)$. For the approximation guarantee, note that (intuitively) the error that we make by this rounding is at most μ for each item. This gives a total error of at most $n\mu = \epsilon M \leq \epsilon \textsc{Opt}$.

This intuitive argument needs a formal proof. Let S be the set of items found by the DP and let O be an optimal set of items for the original (unrounded) instance. For any item i we have

$$v_i \geq \mu v'_i > \mu(v_i/\mu - 1)) = v_i - \mu.$$

The value of the final solution is

$$\begin{aligned}
\sum_{i\in S} v_i &\geq \sum_{i\in S} \mu v_i' \\
&\geq_{(1)} \sum_{i\in O} \mu v_i' \\
&\geq \sum_{i\in O} (v_i - \mu) \\
&= \textsc{Opt} - |O|\mu \\
&\geq \textsc{Opt} - n\mu \\
&= \textsc{Opt} - \epsilon M \\
&\geq_{(2)} \textsc{Opt} - \epsilon\textsc{Opt} = (1-\epsilon)\textsc{Opt}.
\end{aligned}$$

(1): S is optimal for the rounded instance with value v_i'.
(2): $\textsc{Opt} \geq M$ since taking the item with largest value is a feasible solution.

3.2 Scheduling on identical machines

In Section 2.3 we have seen that the list scheduling algorithm gives a $(2-1/m)$-approximation and that list scheduling in order of longest processing time (LPT) even gives a 4/3-approximation. Here it is shown that also a PTAS is possible. However, the running time will be much larger than that of the simple LPT algorithm. We shall use without loss of generality that $1/\epsilon$ is an integer. (See the remark at the end.)

PTAS:

Choose T^* be some number satisfying $T^* \leq \textsc{Opt}$. Say that a job j is *long* if $p_j > \epsilon T^*$ and call it *short* otherwise.

Step 1. Find a schedule for the long jobs of length at most $(1+\epsilon)\textsc{Opt}$.

Step 2. Add the short jobs one by one, each time placing the job on the least loaded machine.

To prove that this gives a PTAS we need to show that the algorithm: (1) can be implemented in polynomial time, (2) gives a feasible solution, and (3) returns a schedule of length at most $1+\epsilon$ times the optimal length. Clearly, any solution is feasible. Let $P = \sum_j p_j$ be the total length of all jobs and let $C_{\max}$ be the length of the final schedule.

Lemma 4. *The PTAS gives a schedule of length* $C_{\max} \leq (1+\epsilon)\textsc{Opt}$.

Proof. Let l be the job that completes last in the final schedule.
Case 1. Job l is a long job. Then $C_{\max} \leq (1+\epsilon)\text{OPT}$.
Case 2. Job l is a short job. The analysis is the same as for the List Scheduling. Let S_l be the start time of job l. Then no machine is idle before time S_l, which implies $S_l \leq P/m \leq OPT$.

$$C_{\max} = S_l + p_l \leq \text{OPT} + p_l \leq \text{OPT} + \epsilon T^* \leq \text{OPT} + \epsilon\text{OPT} = (1+\epsilon)\text{OPT}.$$

□

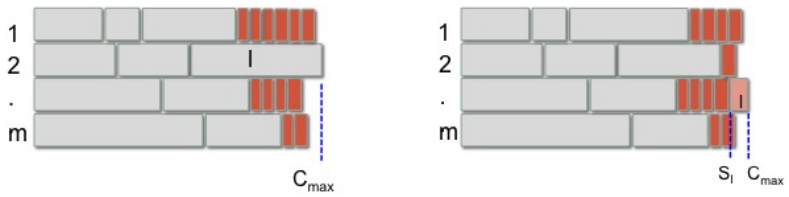


Figure 15: *Left, a long job completes last. Right, a short job completes last.*

It remains to prove that the algorithm can be made to run in polynomial time. Clearly, step 2 can be done in polynomial time. For step 1, this is easy if we assume the number of machines m to be constant. In that case, we can even find an optimal schedule in constant time! This is shown in the next lemma. Subsequently, we see how the running time can be improved to a polynomial in m, which will be a lot more complicated.

From Section 2.3 we know that $P/m \leq \text{OPT}$. So we may choose $T^* = P/m$ in the PTAS.

Lemma 5. *If $T^* = P/m$ then an optimal schedule of the long jobs can be found in $O(m^{m/\epsilon})$ time.*

Proof. Since the total processing time of all jobs is P, the number of long jobs is at most $P/(\epsilon P/m) = m/\epsilon$. To find the optimal schedule of the long jobs we just try all possible schedules and take the best. Note that the order of jobs on a machine is not important for the length of the schedule. Therefore, the number of possible schedules is at most $m^{m/\epsilon}$. □

Theorem 3.2. *If the number of machines m is assumed constant then, for $T^* = P/m$, the PTAS runs in polynomial time.*

Reducing the running time The running time of the PTAS above if exponential in m. Next, we show how to get a PTAS with running time polynomial in m. We show how to get a value $T^* \leq \textsc{Opt}$ and a schedule for the long jobs of length at most $(1+\epsilon)\textsc{Opt}$. For this we use the standard technique of dynamic programming.

In general, dynamic programming is used to find an *optimal* solution for some optimization problem but it may also be used in a more relaxed way where the DP takes as input a value T and is only guaranteed to find a *feasible* solution of value $\leq T$ if $\textsc{Opt} \leq T$ (for a minimization problem). Such relaxed DP is easier to obtain than one that solves the problem directly but it will be enough for our purpose. This general technique is known as a *relaxed decision procedure*. (See also Exercise 2.4.) Assume we have an algorithm B_ϵ with the following property:

Input: Target value T.
Output: A schedule for the long jobs of length at most $(1+\epsilon)T$ if $\textsc{Opt} \leq T$.

Assume for now that such an algorithm B_ϵ can be implemented in polynomial time. Then, by applying it for different values of T, we can find the *smallest* T, say T^*, such that the algorithm returns a schedule of length at most $(1+\epsilon)T^*$. Since the algorithm is guaranteed to work for all $T \geq \textsc{Opt}$, we must have $T^* \leq \textsc{Opt}$ and the obtained schedule has length at most $(1+\epsilon)\textsc{Opt}$. To find this T^*, we can use binary search. Assuming that all processing times are integer, we have $\textsc{Opt} \in \{1, 2, \ldots, P\}$. The number of iterations for binary search is then $O(\log P)$ which is polynomial.

We see that the PTAS works out if we have an algorithm B_ϵ with the property as stated above. Note that in this PTAS, step 1 and the value T^* are now computed simultaneously. Step 2 stays the same. It remains to show that such an algorithm B_ϵ can be made to run in polynomial time. We use dynamic programming to a rounded instance just like we did for the knapsack problem. By rounding, we lose a factor $1+\epsilon$ in the approximation but the advantage of rounding is that it speeds up the DP. Given T, the processing times are rounded as follows.

$$\mu = \epsilon^2 T, \text{ and } p'_j = \lfloor \frac{p_j}{\mu} \rfloor \mu.$$

Algorithm B_ϵ
Given T, find an optimal assignment for the rounded long jobs and take that as solution for the unrounded long jobs.

We need to show that:

(i) The length of the schedule by B_ϵ is at most $(1+\epsilon)T$ if $\textsc{Opt} \leq T$.
(ii) An optimal schedule for rounded long jobs can be found in polynomial time.

(i) Assume $\textsc{Opt} \leq T$. The length of the constructed optimal rounded schedule is at most $\textsc{Opt} \leq T$ since $p'_j \leq p_j$ for all jobs j. Rounding back gives an increase in processing time for each job j of

$$p_j - p'_j = p_j - \lfloor \frac{p_j}{\mu} \rfloor \mu < \mu = \epsilon^2 T.$$

A job j is long if $p_j \geq \epsilon T$. Since $\textsc{Opt} \leq T$, there can be at most $1/\epsilon$ long jobs per machine in our schedule. Hence, the total increase in processing time for each machine is at most $(1/\epsilon)\epsilon^2 T = \epsilon T$ and the total length of the schedule for the long jobs is at most $(1+\epsilon)T$.

(ii). The rounded processing times are multiples of μ. Let $r\mu$ be the largest processing time in the rounded instance. Then

$$p'_j \in \{0, \mu, \ldots, r\mu\}.$$

- Say that a job is of type i if its rounded processing time is $i\mu$.
- A vector $(n_1, \ldots, n_r)$ represents a set of jobs for which n_i is the number of jobs of type i.
- Let $\mathcal{C}$ be the set of all possible vectors for which the total rounded processing time is at most T, i.e., for which the jobs fit on one machine.
- Denote the number of jobs of type i in the rounded instance by $\hat{n}_i$.
- For any $k \in \{1, \ldots, m\}$ and vector $(n_1, \ldots, n_r)$, let $F_k(n_1, \ldots, n_r)$ =TRUE if there is a schedule on k machines of length at most T for the job set $(n_1, \ldots, n_r)$.

Then,

- $F_1(n_1, \ldots, n_r)$ is TRUE if and only if $(n_1, \ldots, n_r) \in \mathcal{C}$ and
- $F_k(n_1, \ldots, n_r)$ is TRUE if and only if there is an $(s_1, \ldots, s_r) \in \mathcal{C}$ such that $F_{k-1}(n_1 - s_1, \ldots, n_r - s_r)$ is TRUE.

We need to compute $F_m(\hat{n}_1, \ldots, \hat{n}_r)$ and check if its value is TRUE. The corresponding schedule can be found if in the DP we also store for each vector $n_1, \ldots, n_r$ a corresponding schedule. It remains to show that the DP runs in polynomial time. This follows from the following observations.

1. The number r of types is constant; Since $T \geq \textsc{Opt} \geq p_{\max}$ (where $p_{\max} = \max_j p_j$), the largest rounded processing time is $\lfloor \frac{p_{\max}}{\mu} \rfloor \mu \leq \lfloor \frac{T}{\mu} \rfloor \mu = \mu/\epsilon^2$. Hence, $r \leq 1/\epsilon^2$, which is a constant.

2. The number of rounded long jobs per machine is constant; For any long job j we have $p_j \geq \epsilon T = \mu/\epsilon$. Then, $p'_j = \lfloor \frac{p_j}{\mu} \rfloor \mu \geq \lfloor \frac{1}{\epsilon} \rfloor \mu = \frac{\mu}{\epsilon} = \epsilon T$ (where we used that $1/\epsilon$ is integer). We conclude that the number of rounded long jobs per machine is at most $1/\epsilon$, which is a constant.

3. From 1. and 2. we see that $|\mathcal{C}|$ is constant.

4. Each value $F_k(n_1, \ldots, n_r)$ can be computed in $O(|\mathcal{C}|)$ time, which is constant.

Note that $\hat{n}_i \leq m/\epsilon$ since each of the m machines has at most $1/\epsilon$ long jobs. That means that the number of evaluations $F_k(n_1, \ldots, n_r)$ that we need to make is at most $m(m/\epsilon)^r \leq m(m/\epsilon)^{1/\epsilon^2}$, which is polynomial in m. From 4. we know that each of these evaluations can be done in constant time.

Remarks We assumed here that $1/\epsilon$ is integer. The book does not assume this and defines $k = \lceil \frac{1}{\epsilon} \rceil$ and works with k instead of ϵ. But note that if we have a polynomial time $(1+\epsilon)$-approximation for all $\epsilon > 0$ with $1/\epsilon$ integer then we automatically have a $(1+\epsilon)$-approximation for all $\epsilon > 0$. For example, if we want a 1.3 approximation algorithm ($\epsilon = 0.3$) then simply take $\epsilon' = 0.25$ and use algorithm $A_{\epsilon'}$.

The DP in the book is slightly different. There, $\textsc{Opt}(n_1, \ldots, n_r)$ is the minimum number of machines for which there is a schedule of length at most T for the corresponding set of jobs. In our DP, the number of machines is added as a parameter k, which makes the recursion a bit easier since there is no minimization done inside the recursion.

4 Deterministic rounding of linear programs

4.1 Sum of completion times on a single machine.

This section gives a 2-approximation algorithm for minimizing the total completion time on a single machine. Preemption of jobs is not allowed. If preemption is allowed then the processing of a job may be interrupted and continued on a later moment. In this section, the preemptive schedule is used to construct a non-preemptive schedule.

Let OPT be the optimal value and let OPT^P be the optimal value for the preemptive problem. The optimal non-preemptive schedule is a feasible schedule for the preemptive problem.

$$\text{OPT}^P \leqslant \text{OPT}.$$

The optimal preemptive schedule is found by the Shortest Remaining Processing Time rule.

SRPT: At any moment t, process the job with the smallest remaining processing time among the available jobs, i.e., among the jobs j with $r_j \geqslant t$.:

Lemma 6. *The SRPT rule gives an optimal preemptive schedule.*

Proof. The book gives no proof but a sketch of a possible proof is as follows. Assume that a schedule σ does not satify the SRPT property. Then there must be two jobs, say j and k, such that the pair j, k does not satify the SRPT property. Let $t = \max\{r_j, r_k\}$. Assume that the remaining processing time of job j at time t is not larger than the remaining processing of job k at time t. Now reschedule as follows: for every moment $s \geqslant t$ that σ processes j or k, now process job j at time s until j completes. From that moment only process job k whenever σ processes j or k. Let C_j^σ and C_k^σ be the completion times in σ and let C'_j and C'_k be the new completion times. Then,

$$C'_j < \min\{C_j^\sigma, C_k^\sigma\} \text{ and } C'_k = \max\{C_j^\sigma, C_k^\sigma\}.$$

Hence, the sum of the two completion times decreases by rescheduling the two jobs in SRPT order. That shows that σ is not optimal. Hence, any optimal schedule must be an SRPT schedule. Also note that there may be more SRPT schedules (this happens when two jobs have the same remaininig processing times at some moment) but all have the same value. □

Algorithm

Step 1 : Construct a preemptive schedule using the SRPT rule. Let C_j^p be the completion time of j in this schedule. Relabel the jobs such that $C_1^p \leqslant C_2^p \leqslant \dots C_n^p$.

Step 2 : Schedule the jobs non-preemptively and as early as possible in the order $1, 2, \dots, n$. Denote the schedule by σ^N and let C_j^N be the completion time of job j in this schedule.

Theorem 4.1. *The algorithm above is a 2-approximation algorithm.*

Proof. Consider an arbitrary job j. Since jobs are scheduled in σ^N in the order $1, 2, ...$ we have that

(i) only jobs $k \leqslant j$ are scheduled before time C_j^N in σ^N.

Further, note that at time C_j^N all jobs $k \leqslant j$ have been released ($r_k \leqslant C_k^P \leqslant C_j^P$). Since jobs are scheduled as early as possible, we have that

(ii) there is no idle time in σ^N between time C_j^P and C_j^N.

From (i) and (ii) we see that the

$$C_j^N \leqslant C_j^P + \sum_{k \leqslant j} p_k \leqslant C_j^P + C_j^P = 2C_j^P.$$

The last inequality above follows since all jobs $k \leq j$ are scheduled before time C_j^P in the SRPT schedule. □

4.2 Weighted sum of completion times.

This section gives a 3-approximation for the weighted version of Section 4.1. Some notation:

For any set of jobs $S \subseteq \{1, 2, \dots, n\}$ denote $p(S) = \sum_{j \in S} p_j$.

Lemma 7. *For any feasible schedule and for any set of jobs $S \subseteq \{1, 2, \dots, n\}$:*

$$\sum_{j \in S} p_j C_j \geqslant \frac{1}{2} p(S)^2.$$

Proof. To simplify notation, label the jobs such that $S = \{1, 2, \dots, j\}$, with $j = |S|$ and assume that they are scheduled in the order $1, 2, \dots, j$. Then, $C_1 \geqslant p_1$ and in general $C_k \geqslant p_1 + \dots + p_k$ for $k \leqslant j$.

$$\begin{aligned}\sum_{k=1}^{j} p_k C_k &\geqslant p_1 p_1 + p_2(p_1+p_2) + \cdots + p_j(p_1+\cdots+p_j) \\ &= \frac{1}{2}(p_1+\cdots+p_j)^2 + \frac{1}{2}p_1^2 + \cdots + \frac{1}{2}p_j^2 \\ &\geqslant \frac{1}{2}(p_1+\cdots+p_j)^2 = \frac{1}{2}p(S)^2.\end{aligned}$$

□

With the lemma above we see that the following LP is a relaxation of our scheduling problem. Here, there is a variable C_j for all jobs j. Note that non-negativity, $C_j \geqslant 0$, is implied by the first constraint.

$$\begin{aligned}\text{(LP)}\ \min\quad & Z = \sum_{j=1}^{n} w_j C_j \\ s.t.\quad & C_j \geqslant r_j + p_j && \text{for all jobs } j \\ & \sum_{j\in S} p_j C_j \geqslant \tfrac{1}{2}p(S)^2 && \text{for all sets } S \subseteq \{1,\ldots,n\}\end{aligned}$$

Algorithm
Step 1: Solve the LP. Assume (for the ease of notation) that $C_1^* \leqslant C_2^* \leqslant \ldots C_n^*$.
Step 2: Schedule the jobs non-preemptively and as early as possible in the order $1, 2, \ldots, n$.

Denote the obtained schedule by σ^N and let C_j^N be the completion time of job j in this schedule. Note that it is not clear yet that the LP can be solved in polynomial time since it has exponentially many constraints: the number of subsets S is 2^n. In Section 4.3 it is shown that this is indeed possible. There it is shown that if the second constraint is violated for some set S, then there must be a set of the form $\{1, 2, \ldots, j\}$ that violates the constraint. Thus, to check feasibily, we only need to check the constraint for all these sets, of which there are n. Using the elipsoid algorithm, the LP can be solved in polynomial time.

Example Consider an instance with two jobs of length 1, weight 1, and released at time zero: $p_1 = p_2 = 1$, $w_1 = w_2 = 1$ and $r_1 = r_2 = 0$. In the optimal schedule, one job completes at time 1 and the other at time 2: $\textsc{Opt} = 3$. On the other hand, the optimal LP-solution is $C_1 = C_2 = 1$ and has value $Z_{LP}^* = 2$. The ratio $\textsc{Opt}/Z_{LP}^* = 3/2$ and can be made arbitrarily close to 2 by taking more jobs.

Lemma 8. *$C_j^N \leqslant 3C_j^*$ for every job j.*

Proof. The first part of the proof is exactly the same as in Section 4.1. Consider an arbitrary job j. Since jobs are scheduled in σ^N in the order $1, 2, ...$ we have that

(i) only jobs $k \leqslant j$ are scheduled before time C_j^N in σ^N.

Further, since at time C_j^* all jobs $k \leqslant j$ have been released and jobs are scheduled as early as possible, we have that

(ii) there is no idle time in σ^N between time C_j^* and C_j^N.

From (i) and (ii) we see that the

$$C_j^N \leqslant C_j^* + \sum_{k \leq j} p_k. \tag{4}$$

So far, the only difference with the proof of 4.1 is that C_j^P was replace by C_j^*. Here, we can't use that $\sum_{k \leqslant j} p_k \leq C_j^*$ since this is not true in general. See the example above. The second constraint of the LP gives us a weaker inequality (Take $S = \{1, 2, \ldots, j\}$):

$$C_j^* \sum_{k \leqslant j} p_k = \sum_{k \leqslant j} C_j^* p_k \geqslant \sum_{k \leqslant j} C_k^* p_k \geq \frac{1}{2}(\sum_{k \leqslant j} p_k)^2 \quad \Rightarrow \quad C_j^* \geq \frac{1}{2} \sum_{k \leqslant j} p_k.$$

Combining this with (4) we get

$$C_j^N \leqslant C_j^* + 2C_j^* = 3C_j^*.$$

□

Theorem 4.2. *The algorithm above is a 3-approximation algorithm.*

Proof. In Section 4.3 it is shown that the LP can be solved in polynomial time. The approximation ratio folows directly from Lemma 8: The total weighted completion time of the schedule is

$$\sum_j w_j C_j^N \leqslant 3 \sum_j w_j C_j^* = 3Z_{LP}^* \leqslant 3\textsc{Opt}.$$

□

4.3 Using the ellipsoid method

The book does not explain the method and only explains the use of a separation oracle.

Separation oracle: Given a set of linear constraints in $\mathbb{Q}^n$ and a point $x \in \mathbb{Q}^n$, it either correctly states that x satisfies all constraints or it gives a constraint that is violated by x.

The lecture notes of Arora give a nice and short explanation of the Ellipsoid method (see course material). Also, the notes at the end of Chapter 4 give a short discription. For us it is sufficient to know the following.

Lemma 9. *(Khachiyan 1979) If a Linear Program has a polynomial time separation oracle, then an optimal solution of the LP can be found in polynomial time.*

The book shows that the LP of Section 4.2 has a separation oracle. The proof is technical but essentially shows that it is enough for the separation oracle to check (besides the first constraint) the second constraint only for sets $S = \{1, 2, \ldots, j\}$ for all $j = 1, \ldots, n$. That means, only $n + n$ constraints need to be checked and it is easy to see if any of these is violated.

A nicer application is the TSP problem (not in the book). Let $G = (V, E)$ be a complete graph and let c_e be the cost for edge $e \in E$. The following ILP is an exact formulation of the TSP problem.

$$
\begin{aligned}
\text{(ILP)} \quad \min Z = \quad & \sum_{e \in E} c_e x_e & \\
s.t. \quad & \sum_{e : e=(i,j)} x_e = 2 & \text{for all } i \in V, \\
& \sum_{e \in \delta(S)} x_e \geqslant 2 & \text{for all } S \subsetneq V, S \neq \emptyset, \\
& x_e \in \{0, 1\} & \text{for all } e \in E.
\end{aligned}
$$

The first constraint says that each point has degree two and the second constraint says that for any proper subset S of the vertices, at least two edges should be in the cut defined by S. (Here, $\delta(S)$ contains all edges that go from a point in S to a point in $V - S$.) In the relaxation, the constraint $x_e \in \{0, 1\}$ is replaced by $x_e \geqslant 0$. Note that the second constraint consists for exponentially many set S. Therefore, it is not obvious that this LP can be solved in a time that is polynomial in the number of variables. However, there is a polynomial

time separation oracle. Consider a solution $x \in \mathbb{Q}^{|E|}$. To check wether x is feasible, we need to check if

$$\sum_{e \in \delta(S)} x_e \geqslant 2 \text{ for all } S \subsetneq V, S \neq \emptyset$$

In other words, we need to check if the smallest cut in the graph has capacity at least 2, where x_e is the capacity of edge e. This can be checked using a polynomial time max-flow/min-cut algorithm. If the min-cut between every pair of vertices (s, t) is at least 2, then the constraint is satisfied. If not, then the max-flow/min-cut algorithm will find a cut S of capacity less than 2, i.e., it finds a violated constraint.

4.4 Prize-collecting Steiner Tree

Prize-collecting Steiner Tree:

Instance:	$G = (V, E)$ and a cost c_e for every edge $e \in E$ and a penalty π_i for every vertex $i \in V$. Also given is a root $r \in V$.
Solution:	Tree T containing r. Let $V(T)$ be the vertices in T
Cost:	$\sum_{e \in T} c_e + \sum_{i \in V - V(T)} \pi_i$.
Goal:	Find a solution of minimum cost.

In words, the goal is to find a tree containing r that minimizes the cost of all edges in the tree plus the penalty cost for the vertices that are not connected by T. It is not needed that the cost is metric since a tree will never select an edge (i, j) if there is a shorter path between i and j. An ILP for the problem is as follows.

$$\begin{aligned}
\text{(ILP)} \quad \min \quad & Z = \sum_{e \in E} c_e x_e + \sum_{i \in V} \pi_i (1 - y_i) & \\
s.t. \quad & \sum_{e \in \delta(S)} x_e \geqslant y_i, & \text{for all } i, S \text{ with } i \in S \subseteq V - r, \\
& y_r = 1, & \\
& x_e \in \{0, 1\}, & \text{for all } e \in E, \\
& y_i \in \{0, 1\}, & \text{for all } i \in V.
\end{aligned}$$

The first constraint says that we may only choose point i in the tree if there is some path between r and i in T. A natural algorithm is to solve the LP-relaxation and to add all points i to T which have a large value y_i^* in the

optimal LP solution. Let $\alpha \in [0,1]$ to be chosen appropriately later.

Algorithm :
Step 1: Solve the LP-relaxation of (ILP) $\rightarrow$ x^*, y^*, Z^*_{LP}.
Step 2: Let $U = \{i \mid y^*_i \geqslant \alpha\}$. Construct a Steiner tree T on U (using the algorithm of Exercise 7.6)

Lemma 10. *The connection cost for the Steiner tree T is*

$$\sum_{e \in T} c_e \leqslant \frac{2}{\alpha} \sum_{e \in E} c_e x^*_e.$$

Proof. See Exercise 7.6. We skip the proof for now. □

Lemma 11. *The total penalty cost is*

$$\sum_{i \in V - V(T)} \pi_i \leqslant \frac{1}{1-\alpha} \sum_{i \in V} (1 - y^*_i)\pi_i.$$

Proof. If vertex i is not in the tree T then $i \notin U$ and $y^*_i < \alpha$, which implies that $(1 - y^*_i)/(1-\alpha) > 1$. The total penalty cost is

$$\sum_{i \in V - V(T)} \pi_i \leqslant \sum_{i \notin U} \pi_i \leqslant \sum_{i \notin U} \frac{1 - y^*_i}{1-\alpha} \pi_i = \frac{1}{1-\alpha} \sum_{i \notin U} (1 - y^*_i)\pi_i \leqslant \frac{1}{1-\alpha} \sum_{i \in V} (1 - y^*_i)\pi_i.$$

□

Theorem 4.3. *The algorithm above with $\alpha = 2/3$ is a 3-approximation algorithm for the Prize-collecting Steiner Tree problem.*

Proof. The sum of connection and penalty cost is at most

$$\frac{2}{\alpha} \sum_{e \in E} c_e x^*_e + \frac{1}{1-\alpha} \sum_{i \in V} (1 - y^*_i)\pi_i \leqslant \max\left\{\frac{2}{\alpha}, \frac{1}{1-\alpha}\right\} Z^*_{LP}.$$

The value $\max\left\{\frac{2}{\alpha}, \frac{1}{1-\alpha}\right\}$ is 3 for $\alpha = 2/3$. Hence, the total cost of the solution of the algorithm is at most

$$3Z^*_{LP} \leqslant 3\textsc{Opt}.$$

N.B. It is easy to show that $\max\left\{\frac{2}{\alpha}, \frac{1}{1-\alpha}\right\}$ is indeed *minimized* for $\alpha = 2/3$, but for the proof it is enough to just *choose* $\alpha = 2/3$. □

4.5 Uncapacitated facility location

Uncapacitated Facility Location (UFL):

Instance:	Set of points F (facilities) and set of points D (clients). Opening cost f_i for each $i' \in F$ and connection cost (assignment cost) c_{ij} for each pair $i \in F, j \in D$. The connection cost is assumed to be metric.
Solution:	$F' \subseteq F$.
Cost:	$\sum_{i \in F'} f_i + \sum_{j \in D} \min_{i \in F'} c_{ij}$
Goal:	Find solution of minimum cost.

A variant is the *capacitated* facility location problem in which each facility can connect to only a bounded number of clients. The following ILP is an exact formulation of UFL.

$$
\begin{aligned}
\text{(ILP)} \quad \min \quad & Z = \sum_{i \in F} f_i y_i + \sum_{i \in F, j \in D} c_{ij} x_{ij} & \\
s.t. \quad & \sum_{i \in F} x_{ij} = 1 & \text{for all } j \in D, \\
& x_{ij} \leqslant y_i & \text{for all } i \in F, j \in D, \\
& x_{ij} \in \{0,1\} & \text{for all } i \in F, j \in D, \\
& y_i \in \{0,1\} & \text{for all } i \in F.
\end{aligned}
$$

In the LP-relaxation, replace the binary constraints by $x_{ij} \geqslant 0$ and $y_i \geqslant 0$. The dual of the LP-relaxation is:

$$
\begin{aligned}
\text{(D)} \quad \max \quad & Z = \sum_{j \in D} v_j & \\
s.t. \quad & \sum_{j \in D} w_{ij} \leqslant f_i & \text{for all } i \in F, \\
& v_j - w_{ij} \leqslant c_{ij} & \text{for all } i \in F, j \in D, \\
& w_{ij} \geqslant 0 & \text{for all } i \in F, j \in D, \\
& (v_i \text{ is free}). &
\end{aligned}
$$

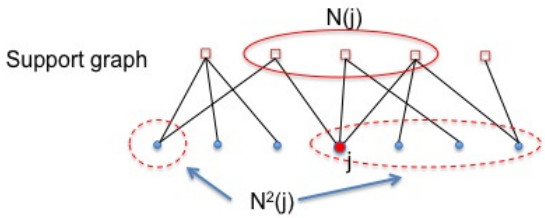
Let x^*, y^* be an optimal primal solution to the LP-relaxation and v^*, w^* be an optimal dual solution.

Lemma 12. *If $x^*_{ij} > 0$, then $c_{ij} = v^*_j - w^*_{ij} \leqslant v^*_j$.*

Proof. Directly from complementary slackness and $w^*_{ij} \geqslant 0$. □

Some definitions:

- The *support graph* of x^* is the graph with vertex set $F \cup D$ and there is an edge between $i \in F$ and $j \in D$ if $x^*_{ij} > 0$.
- $N(j)$ is the set of neighbors of client $j \in D$ in the support graph, i.e., the set of all facilities i with $x^*_{ij} > 0$.
- $N^2(j)$ is the set of all neighbors of neighbors of client $j \in D$ in the support graph.



Algorithm For $k = 1, 2, \ldots$ until all clients are connected do:
Step 1: Among the unconnected clients, choose client j_k with smallest value $v^*_{j_k}$.
Step 2: Choose facility $i_k \in N(j_k)$ with smallest value f_{i_k}.
Step 3: Connect all still unconnected clients in $N^2(j_k)$ to facility i_k.

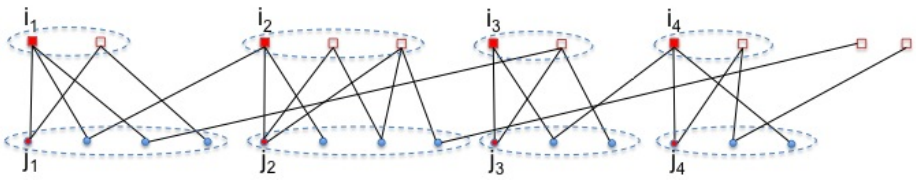


Figure 16: *Structure of algorithm explained. The graph reperesents a support graph. The vertices are shown from left to right in the order in which they are assumed to be selected by the algorithm. For example, if j_1 and i_1 are the first client and facility to be selected, then the first circle on the bottom shows the clients that are connected to i_1 by the algorithm.*

Theorem 4.4. *The algorithm above is a 4-approximation algorithm for UFL.*

Proof. The opening cost for each opened facility i_k is

$$f_{i_k} = f_{i_k} \sum_{i \in N(j_k)} x^*_{ij_k} \leq \sum_{i \in N(j_k)} f_i x^*_{ij_k} \leqslant \sum_{i \in N(j_k)} f_i y^*_i.$$

The equality above follows from the first LP-constraint. The first inequality follows by definiton of Step 2 of the algorithm and the last inequality follows from the second LP-constraint. The total opening cost is

$$\sum_k f_{i_k} \leqslant \sum_k \sum_{i \in N(j_k)} f_i y^*_i \leqslant \sum_{i \in F} f_i y^*_i \leqslant Z^*_{LP} \leqslant \textsc{Opt}. \tag{5}$$

The second inequality above follows from Step 3 of the algorithm which ensures that $N(j_k) \cap N(j_{k'}) = \emptyset$ for any pair $j_k, j_{k'}$. See also Figure 16.

For the connection cost, consider an arbitrary client $l \in D$ and assume that $l \in N^2(j_k)$ for some j_k (possibly, $l = j_k$). That means, that l is neighbor of some $h \in N(j_k)$ (possibly, $h = i_k$). The algorithm assigns client l to facility i_k. The connection cost is

$$\begin{aligned} c_{i_k l} &\leq_{(1)} c_{i_k j_k} + c_{h j_k} + c_{hl} \\ &\leq_{(2)} v^*_{j_k} + v^*_{j_k} + v^*_l \\ &\leq_{(3)} 3v^*_l \end{aligned}$$

(1): Follows from the triangle inequality.
(2): Follows from Lemma 15 (complementary slackness).
(3): From step 1 of the algorithm $v^*_{j_k} \leqslant v^*_l$. Note that both l and j_k were unassigned when j_k was chosen in Step 1.

The total connection cost is at most

$$3 \sum_{l \in D} v^*_l = 3Z^*_{\text{Dual}} \leqslant 3Z^*_{LP} \leqslant 3\textsc{Opt}. \tag{6}$$

From (5) and (6), we conclude that the sum of opening and connection cost is at most $\textsc{Opt} + 3\textsc{Opt} = 4\textsc{Opt}$.

□

5 Random sampling and randomized rounding of LP's

Definition 3. *A randomized α-approximation algorithm for an optimization problem is a polynomial-time algorithm that for all instances of the problem produces a solution for which the expected value is within a factor α of the optimal value.*

To show that a randomized algorithm ALG is an α-approximation algorithm we need to show three things:

[1] The algorithm runs in polynomial time.

[2] The algorithm always produces a feasible solution.

[3] The expected value is within a factor α of the value of an optimal solution.

In Section 1.7 we have seen a randomized algorithm for set cover that did not satisfy [2]. There, it was shown that the algorithm produces a feasible solution with *high probability*. An other example are the Las Vegas algorithms which always find a feasible solution but the running time is only *polynomial in expectation*. In this Chapter we only use the definition above.

5.1 Max Sat and Max Cut

Max Sat

An example of an instance of the Maximum Satisfiability problem:

$$x_1 \vee x_2, \qquad \neg x_1, \qquad x_2 \vee \neg x_2 \vee x_3, \qquad \neg x_2 \vee x_4, \qquad \neg x_2 \vee \neg x_4.$$

Some notation:

- There are $n = 4$ *boolen variables* $x_i \in \{\texttt{true}\ , \texttt{false}\ \}$.
- The first *clause* is $x_1 \vee x_2$ and the number of clauses is $m = 5$.
- The third clause has 3 *literals* but only 2 variables.
- x_i is called a *positive* literal and $\neg x_i$ is *negative* literal.
- $x_i = \texttt{true} \ \Leftrightarrow \neg x_i = \texttt{false}$.
- A clause is `true` or 'satisfied' if at least one of the literals is true.

The goal in the maximum satisfiability problem is to find a `true` /`false` - assignment of the variables such that the number of satisfied clauses is maximized. (In a decision variant of the problem, called *satisfiability* (SAT), the question is wether there is an assignment that satisfies *all* clauses. That problem was the first problem to be shown $\mathcal{NP}$-complete, Cook 1971, Levin 1973).

Algorithm Set all variables independently to true with probability 1/2.

Theorem 5.1. *The algorithm is a* 1/2*-approximation for the Max Sat problem.*

Proof. Let l_j be the number of literals in clause C_j. Then,

$$\Pr(C_j \text{ is satisfied}) = 1 - (1/2)^{l_j} \geq 1/2. \tag{7}$$

Let W be the total number of satisfied clauses. Then,

$$\mathbb{E}[W] = \sum_{j=1}^{m} \Pr(C_j \text{ is satisfied}) \geqslant \frac{1}{2}m \geqslant \frac{1}{2}\textsc{Opt}.$$

□

In the *weighted* version of the problem, each clause j is given a weight w_j and the goal is to maximize the total weight of the satisfied clauses. If all weights are 1 then we have exactly the unweighted version. The algorithm and proof work exactly the same:

$$\mathbb{E}[W] = \sum_{j=1}^{m} w_j \Pr(C_j \text{ is satisfied}) \geqslant \frac{1}{2}\sum_{j=1}^{m} w_j \geqslant \frac{1}{2}\textsc{Opt}.$$

Remark If each clause contains at least k literals then from the proof above we see that the approximation guarantee is at least $1 - (1/2)^k$.

Max Cut

The Max Cut problem is the maximization version of the Min Cut problem. It is known that the minimum cut in a graph can be found in polynomial time by solving a maximum flow problem (min-cut max-flow theorem). Finding the maximum cut in a graph is an $\mathcal{NP}$-hard problem.

Algorithm Assign each vertex independently and uniformly at random to one of the two sides.

Theorem 5.2. *The algorithm is a* 1/2*-approximation for the MAX CUT problem.*

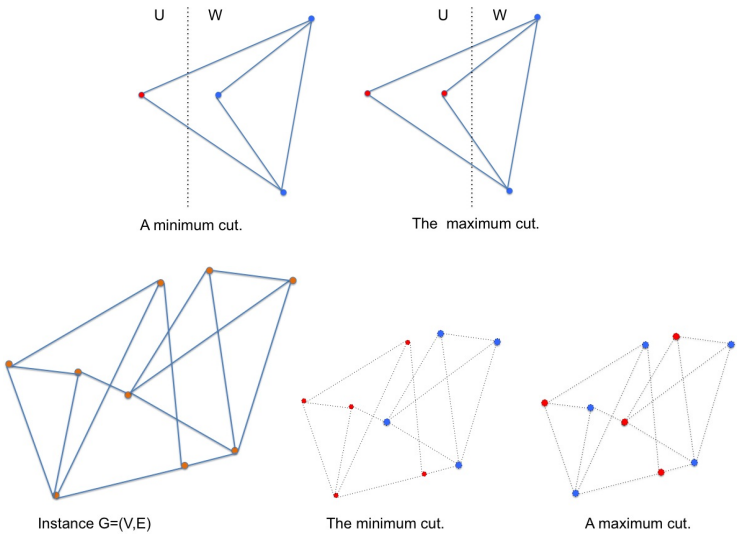


Figure 17: *Finding a maximum cut is in general much harder than finding a minimum cut. The graph $G = (V, E)$ has 15 edges. It is fairly easy to see that the minimum cut has value 2. The maximum cut, has value 11. The red vertices and blue vertices indicate the two sides, U and W, of the cut. You can think of the problem as coloring the vertices with two colors so as to minimize/maximize the number of edges with different colored endpoints.*

Proof. The probability that an edge (i, j) ends up in the cut is exactly $1/2$. Let Z be the total number of edges in the cut found by the algorithm. Then

$$\mathbb{E}[Z] = \sum_{(i,j)\in E} \Pr((i,j) \text{ in the cut}) = \frac{1}{2}|E| \geqslant \frac{1}{2}\textsc{Opt}.$$

□

In the *weighted* version of the problem each edge (i, j) has a given weight w_{ij} and the goal is to maximize the total weight of the edges in the cut. The algorithm and proof work exactly the same:

$$\mathbb{E}[Z] = \sum_{(i,j)\in E} w_{ij}\Pr((i,j) \text{ in the cut}) = \frac{1}{2}\sum_{(i,j)\in E} w_{ij} \geqslant \frac{1}{2}\textsc{Opt}.$$

5.2 Derandomization.

Sometimes, a randomized algorithm can easily be *derandomized*. This is the case for the algorithms of the previous section. The idea of the derandomization is to make our choices one by one and each time making a choice which gives the highest expected objective value. It is best explained by an example. In the max cut algorithm, the vertices are assigned independently at random. To derandomize the algorithm, assign the vertices one by one in arbitrary order, for example, in the order $v_1, \ldots, v_n$. Assume we have already assigned the vertices $v_1, \ldots, v_i$ and denote the partial assignment by S_i. Suppose we assign the other vertices at random as before and denote the expected value of the cut as

$$\mathbb{E}[Z|S_i].$$

Next, S_i is extended by assigning vertex v_{i+1} either to U or W. If we assign the remaining vertices at random as before then the expected values are denoted by

$$\mathbb{E}[Z|S_i \text{ and } v_{i+1} \to U]$$

$$\mathbb{E}[Z|S_i \text{ and } v_{i+1} \to W]$$

From probability theory we know that in general for a stochastic variable Z and event B it holds that $\mathbb{E}[Z] = \mathbb{E}[Z|B]\Pr(B) + \mathbb{E}[Z|\overline{B}]\Pr(\overline{B})$. In this case

$$\begin{aligned}\mathbb{E}[Z|S_i] &= \mathbb{E}[Z|S_i \text{ and } v_{i+1} \to U] \cdot \Pr(v_{i+1} \to U) \ + \\ &\quad \mathbb{E}[Z|S_i \text{ and } v_{i+1} \to W] \cdot \Pr(v_{i+1} \to W) \\ &= \mathbb{E}[Z|S_i \text{ and } v_{i+1} \to U] \cdot \frac{1}{2} \ + \ \mathbb{E}[Z|S_i \text{ and } v_{i+1} \to W] \cdot \frac{1}{2}.\end{aligned}$$

The equality above implies that either

$$\mathbb{E}[Z|S_i \text{ and } v_{i+1} \to U] \geqslant \mathbb{E}[Z|S_i] \ \text{ or } \ \mathbb{E}[Z|S_i \text{ and } v_{i+1} \to W] \geqslant \mathbb{E}[Z|S_i].$$

In the first case we assign v_{i+1} to U and we assign it to W otherwise. Denote the extended assignment by S_{i+1}. Then we achieved that

$$\mathbb{E}[Z|S_{i+1}] \geqslant \mathbb{E}[Z|S_i].$$

If we do this for all vertices $v_1, \ldots, v_n$ then we end up with the assignment S_n and conclude that the value of the cut found by this derandomized algorithm is

$$\mathbb{E}[Z|S_n] \geqslant \mathbb{E}[Z|S_{n-1}] \geqslant \cdots \geqslant \mathbb{E}[Z|S_1] \geqslant \mathbb{E}[Z] \geqslant \frac{1}{2}\textsc{Opt}.$$

5.3 Flipping biased coins

We have seen that setting each variable to true with probability 1/2 satisfies at least half the number of clauses in expectation. Can we do better?

$$C_1 = x_1, \quad C_2 = \neg x_1$$

The example shows that no algorithm can do better than $m/2$. But, for this example the optimal value is 1 and any assignment is optimal. So can we do better than 1/2 times the optimal value? Yes! In this section we give the first example.

An obvious approach is to set the variables independently to true with probability $p \in]0, 1[$.

If $p < 0.5$ then the ratio is worse than 0.5 for the single clause instance: $C = x_1$.
If $p > 0.5$ then the ratio is worse than 0.5 for the single clause instance: $C = \neg x_1$.

This implies that flipping a biased coin won't work if we do this independently of the instance. We will see that it is enough to let p depend only on the unit clauses.

Let's consider the weighted version of MaxSat. In fact, the algorithm and analyis are easier formulated for the weighted version. For example, we may assume w.l.o.g. that no two clauses are the same since we can turn two equal clauses into one by adding the two weights. Let $w(x_i)$ and $w(\neg x_i)$ be the weight of, respectively, clause x_i and $\neg x_i$. If the instance has no such clause then we assume it has zero weight.

Algorithm 3 Choose $p \geq 1/2$. Set x_i to true with probability p if $w(x_i) \geq w(\neg x_i)$ and set x_i to true with probability $1 - p$ otherwise.

Theorem 5.3. *The approximation factor of the algorithm is at least* $\min(p, 1 - p^2)$*, which is*

$$\frac{1}{2}(\sqrt{5} - 1) \approx 0.62 \quad \textit{for } p = \frac{1}{2}(\sqrt{5} - 1).$$

Proof. Let Z_1 be the total weight of unit clauses satisfied by the algorithm and let Z_2 be the total weight of the other clauses satisfied by the algorithm. Fix an optimal solution and let Z_1^* and Z_2^* be the correspinding weights for the optimal solution. First, we show that

$$\mathbb{E}[Z_1] \geq pZ_1^*. \tag{8}$$

Assume $w(x_i) \geq w(\neg x_i)$. The contribution of these two clauses in the optimal solution is at most $w(x_i)$ since the clauses cannot both be satisfied. On the other

hand, the expected satisfied weight by the algorithm is $pw(x_i)+(1-p)w(\neg x_i) \geq pw(x_i)$.

Similarly, if $w(x_i) \leq w(\neg x_i)$ then the contribution of these two clauses in the optimal solution is at most $w(\neg x_i)$. On the other hand, the expected satisfied weight by the algorithm is $(1-p)w(x_i) + pw(\neg x_i) \geq pw(\neg x_i)$.

Next, we show that

$$\mathbb{E}[Z_2] \geq (1-p^2)Z_2^*. \tag{9}$$

Since $p \geq 1/2$, any literal is false with probability at most p. If C_j has l_j literals then

$$\Pr(C_j \text{ is satisfied}) \geq 1 - p^{l_j} \geq 1 - p^2 \quad \text{for } l_j \geq 2.$$

Hence,

$$\mathbb{E}[Z_2] \geq (1-p^2) \sum_{j:l_j \geq 2} w_j \geq (1-p^2)Z_2^*.$$

From (8) and (9) we see that the expected value of the solution is

$$\mathbb{E}[Z_1]+\mathbb{E}[Z_2] \geq pZ_1^*+(1-p^2)Z_2^* \geq \min(p, 1-p^2)(Z_1^*+Z_2^*) = \min(p, 1-p^2)\textsc{Opt}.$$

□

5.4 Randomized Rounding

For each clause C_j let P_j be the indices of the variables x_i that occur positively in the clause, and let N_j be the indices of the variables x_i that are negated in the clause. The following mixed ILP is an exact formulation of the Max Sat problem.

$$\begin{aligned}
\text{(ILP)}\ \max\quad & Z = \sum_{j=1}^{m} w_j z_j \\
s.t.\quad & \sum_{i\in P_j} y_i + \sum_{i\in N_j} (1-y_i) \geqslant z_j && \text{for all } j = 1\dots m, \\
& y_i \in \{0,1\} && \text{for all } i = 1\dots n, \\
& z_j \in \{0,1\} && \text{for all } j = 1\dots m.
\end{aligned}$$

For the relaxation, replace the last two constraints by $0 \leq y_i \leq 1$ and $0 \leq z_j \leq 1$. (Note that only changing last constraint by $0 \leq z_j \leq 1$ will still give an integral optimal solution.)

Algorithm :
Step 1. Solve the LP-relaxation $\rightarrow y^*, z^*, Z^*_{LP}$.
Step 2. Set each variable x_i to true with probability y^*_i.

Theorem 5.4. *The algorithm gives a* $(1-\frac{1}{e})$*-approximation,* $(1-\frac{1}{e} \approx 0.63)$.

Proof. Consider an arbitrary clause C_j and let l_j be the number of literals in it.

$$\begin{aligned}
\Pr(C_j \text{ not sat.}) \quad &= \quad \prod_{i\in P_j}(1-y^*_i) \prod_{i\in N_j} y^*_i \\
&\leq_{(1)} \quad \left[\frac{1}{l_j}\left(\sum_{i\in P_j}(1-y^*_i) + \sum_{i\in N_j} y^*_i\right)\right]^{l_j} \\
&=_{(2)} \quad \left[1 - \frac{1}{l_j}\left(\sum_{i\in P_j} y^*_i + \sum_{i\in N_j}(1-y^*_i)\right)\right]^{l_j} \\
&\leq_{(3)} \quad \left[1 - \frac{1}{l_j} z^*_j\right]^{l_j}
\end{aligned}$$

(1) since the arithmetic mean is at least the geometric mean (Fact 5.8 in book)
(2) rearranging and using that $|P_j| + |N_j| = l_j$
(3) follows from the LP-inequality.

From the inequality above:

$$\Pr(C_j \text{ is sat.}) \quad \geqslant \quad 1 - \left[1 - \frac{1}{l_j} z_j^*\right]^{l_j}$$

If we can derive a bound of the form $\Pr(C_j \text{ is sat.}) \geqslant \alpha z_j^*$ for some constant α then by adding weights and taking the sum over all j we get an α-approximation. Observe that the function $f(z) = 1 - (1 - \frac{1}{l_j} z)^{l_j}$ is concave on $[0, 1]$. Thus,

$$f(z) \geqslant f(0) + (f(1) - f(0))\, z = \left(1 - \left[1 - \frac{1}{l_j}\right]^{l_j}\right) z.$$

$$\Rightarrow \quad \Pr(C_j \text{ is sat.}) \quad \geq \quad \left(1 - \left[1 - \frac{1}{l_j}\right]^{l_j}\right) z_j^*. \tag{10}$$

Now use that the right side above is more than $\left(1 - \frac{1}{e}\right) z_j^*$ fo any integer $l_j \geq 1$.

$$\mathbb{E}[W] = \sum_{j=1}^{m} w_j \Pr(C_j \text{ is sat.}) > \sum_{j=1}^{m} w_j \left(1 - \frac{1}{e}\right) z_j^* = \left(1 - \frac{1}{e}\right) Z_{LP}^* \geqslant \left(1 - \frac{1}{e}\right) \textsc{Opt}.$$

□

5.5 Choosing the better of two solutions

Here, we see that if we apply for any instance both the algorithms of Section 5.4 and 5.1 and take the best of the two solutions, then the approximation ratio is better than what we have seen so far. Let W_1 and W_2 be the weight of the solution for the algorithm of, respectively, Section 5.4 and 5.1.

Theorem 5.5. $\mathbb{E}[\max(W_1, W_2)] \geqslant \frac{3}{4}\textsc{Opt}$.

Proof. From the Equations (7) of 5.1 and (10) of 5.4 it follows that

$$\begin{aligned}
\mathbb{E}[\max(W_1, W_2)] &\geqslant \frac{1}{2}\mathbb{E}[W_1] + \frac{1}{2}\mathbb{E}[W_2] \\
&\geqslant \frac{1}{2}\sum_{j=1}^{m}(1 - 2^{-l_j})w_j + \frac{1}{2}\sum_{j=1}^{m}\left(1 - \left(1 - \frac{1}{l_j}\right)^{l_j}\right) w_j z_j^*. \\
&\geqslant \frac{1}{2}\sum_{j=1}^{m}\left((1 - 2^{-l_j}) + 1 - \left(1 - \frac{1}{l_j}\right)^{l_j}\right) w_j z_j^*.
\end{aligned}$$

The last inequality above follows from $z_j^* \leqslant 1$. Note that

$$(1 - 2^{-l_j}) + 1 - \left(1 - \frac{1}{l_j}\right)^{l_j} \begin{cases} = 3/2 & \text{for } l_j = 1 \text{ or } l_j = 2, \\ > \frac{7}{8} + 1 - 1/e > 3/2 & \text{for } l_j \geqslant 3. \end{cases}$$

Thus,

$$\mathbb{E}[\max(W_1, W_2)] \geqslant \frac{1}{2}\sum_{j=1}^{m}\frac{3}{2}w_j z_j^* = \frac{3}{4}\sum_{j=1}^{m} w_j z_j^* = \frac{3}{4}Z_{LP}^* \geqslant \frac{3}{4}\textsc{Opt}.$$

□

5.6 Non-linear randomized rounding

Section 5.5 showed a 3/4-approximation algorithm by taking the best of two solutions. Here it is shown that the same ratio can be obtained if we round the LP of section 5.4 in a more sophisticated way. Instead of setting x_i to true with probability y_i^*, we set it to true with probability $f(y_i^*)$ for some appropriate function f. It turns out that this works for any function f between the following bounds.

$$1 - 4^{-y} \leqslant f(y) \leqslant 4^{y-1}. \tag{11}$$

Algorithm :
Step 1. Solve the LP-relaxation of Section 5.4 $\rightarrow y^*, z^*, Z_{LP}^*$.
Step 2. Set each variable x_i to true with probability $f(y_i^*)$ for some function $f(y)$ satisfying (11).

Theorem 5.6. *The algorithm above is a* 3/4*-approximation for (weighted) Max Sat.*

Proof.

$$\begin{aligned}
\Pr(C_j \text{ is not sat.}) &= \prod_{i \in P_j} (1 - f(y_i^*)) \prod_{i \in N_j} f(y_i^*) \\
&\leqslant \prod_{i \in P_j} 4^{-y_i^*} \prod_{i \in N_j} 4^{y_i^* - 1} \\
&= 4^{-(\sum_{i \in P_j} y_i^* + \sum_{i \in N_j} 1 - y_i^*)} \\
&\leqslant 4^{-z_j^*}
\end{aligned}$$

From the above:

$$\Pr(C_j \text{ is sat.}) \geqslant 1 - 4^{-z_j^*}.$$

Just as in Section 5.4, we replace the inequality above by a (weaker) inequality from which the approximation ratio follows immediately. Note that $g(z) = 1 - 4^{-z}$ is a concave function on $[0, 1]$. Thus,

$$g(z) \geqslant g(0) + (g(1) - g(0))z = 1 - \frac{1}{4}z \geq \frac{3}{4}z, \quad \text{for } 0 \leqslant z \leqslant 1.$$

$$\Rightarrow \quad \mathbb{E}[W] = \sum_{j=1}^{m} w_j \Pr(C_j \text{ is sat.}) \geqslant \sum_{j=1}^{m} w_j \frac{3}{4} z_j^* = \frac{3}{4} Z_{LP}^* \geqslant \frac{3}{4} \textsc{Opt}.$$

□

5.7 Prize-collecting Steiner tree

Section 4.4, gave a 3-approximation algorithm by LP-rounding. A vertex i was added to the tree if $y_i^* \geqslant \alpha$ where $\alpha = 2/3$. Now we take $\alpha \in [\gamma, 1]$ uniformly at random. The value γ is chosen appropriately later.

Algorithm :
Step 1: Solve the LP-relaxation of Section 4.4 $\rightarrow \quad x^*, y^*, Z_{LP}^*$.
Step 2: Take $\alpha \in [\gamma, 1]$ uniformly at random.
Let $U = \{i \mid y_i^* \geqslant \alpha\}$. Construct a Steiner tree T on U.

Lemma 13. *The expected connection cost for the Steiner tree T is*

$$\mathbb{E}[\sum_{e \in T} c_e] \leqslant \frac{2}{1-\gamma} \ln \frac{1}{\gamma} \sum_{e \in E} c_e x_e^*.$$

Proof. By Lemma 4.6 from the book, the expected connection cost is at most

$$\mathbb{E}\left[\frac{2}{\alpha} \sum_{e \in E} c_e x_e^*\right] = \mathbb{E}\left[\frac{2}{\alpha}\right] \sum_{e \in E} c_e x_e^*.$$

Now use that $\mathbb{E}\left[\frac{2}{\alpha}\right] = \frac{1}{1-\gamma} \int_{\alpha=\gamma}^{\alpha=1} \frac{2}{\alpha} d\alpha = \frac{2}{1-\gamma} \ln \frac{1}{\gamma}$. □

Lemma 14. *The expected total penalty cost is at most*

$$\frac{1}{1-\gamma} \sum_{i \in V} 1 - y_i^*$$

Proof. Let $U = \{i \in V : y_i^* \geqslant \alpha\}$. Note that $\Pr(i \notin U) = \begin{cases} 1 & \text{if } y_i^* \leqslant \gamma, \\ \frac{1-y_i^*}{1-\gamma} & \text{if } y_i^* \geqslant \gamma. \end{cases}$

In both cases, the probability is at most $(1 - y_i^*)/(1 - \gamma)$. The expected penalty cost is at most

$$\sum_{i \in V} \pi_i \Pr(i \notin U) \leqslant \sum_{i \in V} \pi_i \frac{1 - y_i^*}{1 - \gamma} = \frac{1}{1-\gamma} \sum_{i \in V} \pi_i (1 - y_i^*).$$

□

Form the two lemmas we conclude that the total expected cost of the solution is at most

$$\max\left\{\frac{2}{1-\gamma} \ln \frac{1}{\gamma}, \ \frac{1}{1-\gamma}\right\} Z_{LP}^*.$$

The maximum above is $1/(1 - e^{-1/2}) \approx 2.54$ for $\gamma = e^{-1/2}$.

Derandomization The algorithm makes only one random decision: it chooses α at random. So the method of computing conditional expectations of Section 5.3 is not very helpful here. It only tells us to pick an α which gives the best objective value, but there are infinitely many α's to try. An obvious approach is to show that trying a polynomial number of value's for α is enough. In our case, we only need $n + 1 = |V| + 1$ different values of α. To see this, label the vertices such that $y_1^* \leqslant \cdots \leqslant y_n^*$. Then, the solution of the algorithm is the same for all values $\alpha \in]y_i^*, y_{i+1}^*[$. In other words, when we let α vary from γ till 1 then the outcome of the rounding changes only when α becomes y_i^* for some i. It is enough to try for α all values $y_1^*, \ldots, y_n^*, 1$ and take the best solution.

Algorithm :
Step 1: Solve the LP-relaxation of Section 4.4 $\rightarrow \quad x^*, y^*, Z_{LP}^*$.
Step 2: For all $\alpha \in \{y_1^*, \ldots, y_n^*, 1\}$ do the following:
Let $U = \{i \mid y_i^* \geqslant \alpha\}$. Construct a Steiner tree T on U.
Step 3: Return the best solution found.

5.8 Uncapacitated facility location.

In Section 4.4 we have seen a 4-approximation algorithm for the uncapacitated facility location problem. The algorithm was to solve an LP-relaxation and then assigning clients one by to the cheapest neighboring facility in the support graph. The main difference here is that in stead of taking the cheapest facility, we take a facility at random using the values x_{ij} from the LP as probability distribution. First, let us repeat (just for completeness) part of Section 4.4:

$$\begin{aligned}
\text{(ILP)} \quad \min \quad & Z = \sum_{i \in F} f_i y_i + \sum_{i \in F, j \in D} c_{ij} x_{ij} \\
s.t. \quad & \sum_{i \in F} x_{ij} = 1 && \text{for all } j \in D, \\
& x_{ij} \leqslant y_i && \text{for all } i \in F, j \in D, \\
& x_{ij} \in \{0,1\} && \text{for all } i \in F, j \in D, \\
& y_i \in \{0,1\} && \text{for all } i \in F.
\end{aligned}$$

In the LP-relaxation, replace the binary constraint by $x_{ij} \geqslant 0$ and $y_i \geqslant 0$. The dual of the LP-relaxation is:

$$\begin{aligned}
\text{(D)} \quad \max \quad & Z = \sum_{j \in D} v_j \\
s.t. \quad & \sum_{j \in D} w_{ij} \leqslant f_i && \text{for all } i \in F, \\
& v_j - w_{ij} \leqslant c_{ij} && \text{for all } i \in F, j \in D, \\
& w_{ij} \geqslant 0 && \text{for all } i \in F, j \in D, \\
& (v_i \text{ is free}).
\end{aligned}$$

Let x^*, y^* be optimal primal solution and let v^*, w^* be optimal dual solution. By complementary slackness, we have the following lemma.

Lemma 15. *If $x^*_{ij} > 0$, then $c_{ij} = v^*_j - w^*_{ij} \leqslant v^*_j$.*

- *Support graph* of x^*: There is an edge if $x^*_{ij} > 0$.
- $N(j)$ is the set of neighbors of client $j \in D$ in the support graph.
- $N^2(j)$ is the set of all neighbors of neighbors of client $j \in D$.

The deterministic algorithm:
For $k = 1, 2, \ldots$ until all clients are connected do:

Step 1: Among the unconnected clients, choose client j_k with smallest value $v^*_{j_k}$.
Step 2: Choose facility $i_k \in N(j_k)$ with smallest value f_{i_k}.
Step 3: Connect all still unconnected clients in $N^2(j_k)$ to facility i_k.

In the randomized version, Step 2 is changed by taking i_k at random. Consequently, we see that the analysis work out nicely if we also make a small change in Step 1. Define the *fractional connection cost* of client j as $C^*_j = \sum_{i \in F} c_{ij} x^*_{ij}$.

The randomized algorithm:
For $k = 1, 2, \ldots$ until all clients are assigned do:
Step 1: Among the unconnected clients choose client j_k with smallest $v^*_{j_k} + C^*_{j_k}$.
Step 2: Choose facility $i_k \in N(j_k)$ at random, where $\Pr(i = i_k) = x^*_{ij}$.
Step 3: Connect all still unconnected clients in $N^2(j_k)$ to facility i_k.

Theorem 5.7. *Algorithm above is a randomized 3-approximation algorithm.*

Proof. The expected opening cost for facility opened in iteration k is

$$\sum_{i \in N(j_k)} f_i x^*_{ij_k} \leqslant \sum_{i \in N(j_k)} f_i y^*_i.$$

The inequality follows from the LP-constraint. Step 3 of the algorithm ensures that $N(j_k) \cap N(j_{k'}) = \emptyset$ for any pair $j_k, j_{k'}$. Thus, the total expected opening cost is

$$\sum_k \sum_{i \in N(j_k)} f_i y^*_i \leqslant \sum_{i \in F} f_i y^*_i.$$

For the expected connection cost, consider an arbitrary iteration k. Given

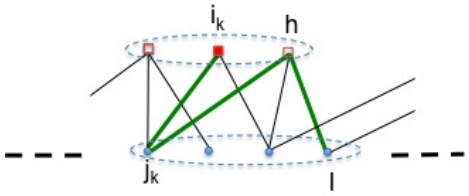


Figure 18: *Sketch of iteration k. Only the clients that get connected in this iteration are shown. Client l is an arbitrary client in $N^2(j_k)$.*

j_k, the neighborhoods $N(j_k)$ and $N^2(j_k)$ are fixed. For any client l that gets connected in that iteration (i.e., $l \in N^2(j_k)$) there is some $h \in N(j_k)$ such that

h is a neighbor of l. The distances $c_{hj_k}+c_{hl}$ are *not* random variables. However, the distance $c_{i_k j_k}$ is a random variable since the facility i_k is chosen at random.

$$\mathbb{E}[c_{i_k j_k}] = \sum_{i \in N(j_k)} x^*_{ij_k} c_{ij_k} = \sum_{i \in F} x^*_{ij_k} c_{ij_k} = C^*_{j_k}.$$

The algorithm assigns client l to facility i_k. The expected connection cost for client l is

$$\begin{aligned}
\mathbb{E}[c_{i_k l}] &\leq_{(1)} \mathbb{E}[c_{i_k j_k} + c_{hj_k} + c_{hl}] \\
&=_{(2)} \mathbb{E}[c_{i_k j_k}] + c_{hj_k} + c_{hl} \\
&= C^*_{j_k} + c_{hj_k} + c_{hl} \\
&\leq_{(3)} C^*_{j_k} + v^*_{j_k} + v^*_l \\
&\leq_{(4)} C^*_l + v^*_l + v^*_l
\end{aligned}$$

(1): Follows from the triangle inequality.
(2): Only the first is a random variable (given j_k).
(3): From the complementary slackness lemma.
(4): From Step 1. Note that j_k and l were both unassigned before j_k was chosen.

The expected value of the solution is the sum of expected opening cost and expected connection cost, which is bounded by

$$\begin{aligned}
&\sum_{i \in F} f_i y^*_i + \sum_{l \in D} C^*_l + 2 \sum_{l \in D} v^*_l \\
= &\left(\sum_{i \in F} f_i y^*_i + \sum_{l \in D} \sum_{i \in F} x^*_{il} c_{il} \right) + 2 \sum_{l \in D} v^*_l \\
= &\; Z^*_{LP} + 2 Z^*_{\mathrm{D}} \;\leqslant\; 3 Z^*_{LP} \;\leqslant\; 3\textsc{Opt}.
\end{aligned}$$

□

6 Random rounding of semidefinite programs

- Section 6.1: Semidefinite Programming
- Section 6.2: Finding large cuts
- Extra (not in book): The max 2-sat problem.
- Section 6.5: Coloring 3-colorabale graphs.

6.1 Semidefinite Programming

The book gives some properties of semidefinite matrices but not all are used in this chapter. Here we list only the ones that are used, and additionally give a definition of a *strict* quadratic program.

Definition 4. *A matrix $X \in \mathbb{R}^{n\times n}$ is positive semidefinite (psd) iff for all $y \in \mathbb{R}^n, y^T X y \geqslant 0$. Notation: $X \succeq 0$.*

Fact 1. *If $X \in \mathbb{R}^{n\times n}$ is symmetric then the following equivalence holds*

$$X \succeq 0 \;\Leftrightarrow\; X = V^T V \text{ for some } V \in \mathbb{R}^{m\times n} \text{ with } m \leqslant n.$$

We may assume that $m = n$ since if $m < n$ we can extend V with zero-rows.

Definition 5. *A semidefinite program (SDP) is a linear program with an additional constraint of the following form: X is positive semidefinite, where X is a square symmetric matrix containing all the variables.*

Example: The following LP

$$\begin{array}{ll} \min & x+y+z \\ s.t. & x+y-z=2 \\ & x,y,z \geqslant 0 \end{array}$$

with the additional constraint: $\begin{bmatrix} x & y \\ y & z \end{bmatrix} \succeq 0$, is called a semidefinite program.

The general form with m linear constraints is

$$\begin{array}{lll} \min \text{or} \max & \sum_{ij} c_{ij} x_{ij} & \\ s.t. & \sum_{ij} a_{ijk} x_{ij} = b_k & \text{for } k = 1,\ldots,m \\ & x_{ij} = x_{ji} & \text{for all } i,j \\ & X = (x_{ij}) \succeq 0. & \end{array} \tag{12}$$

Fact 2. *Semidefinite programs can be solved in polynomial time up to an arbitrarily small additive constant $\epsilon > 0$.*

A quadratic program (QP)

- is a linear program in which, additionally, also the product of two variables is allowed, and
- is called *strict* if it *only* contains constant terms and products of two variables. For example, the constraint $y_1^2 + y_1 y_2 + y_3 = 1$ is not allowed in a strict QP because of the term y_3.

Definition 6. *A vector program is obtained from a strict quadratic program by replacing all variables y_i by vectors $v_i \in \mathbb{R}^n$ (where n is the number of variables y_i) and by replacing the scalar products of variables by vector inproducts. See for an example Section 6.2.*

The general form of a vector program is

$$\begin{array}{rll} \min \text{ or } \max & \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}\ v_i \cdot v_j & \\ \text{s.t.} & \sum_{i=1}^{n}\sum_{j=1}^{n} a_{ijk}\ v_i \cdot v_j = b_k, & k = 1, \ldots, m \\ & v_i \in \mathbb{R}^n & i = 1 \ldots n. \end{array} \tag{13}$$

Theorem 6.1.

Semidefinite programs are equivalent with vector programs. Consequently, vector programs can be solved in polynomial time up to an arbitrarily small additive constant $\epsilon > 0$.

Proof. Let $X \in \mathbb{R}^{n \times n}$ be a feasible solution for SDP (12). Then $X = V^T V$ for some $n \times n$ matrix V. Let v_i be the i-th column of V. Then $v_1, v_2, \ldots, v_n$ is feasible for vector program (13) and the value stays the same. Conversely, if $v_1, v_2, \ldots, v_n$ is feasible for VP, then define

$$V = [v_1 \mid v_2 \mid \cdots \mid v_n] \text{ and } X = V^T V.$$

Then, $X = (x_{ij})$ is feasible for SDP and the value stays the same. □

6.2 Finding large cuts.

In the weighted version of max cut problem, we may assume without loss of generality that the graph is complete since we can define $w_{ij} = 0$ for any missing edge. To simplify notation we write $\sum_{(i,j)}$ to denote the summation over all

edges of the complete graph. The following quadratic program is an exact formulation of the max-cut problem.

$$\begin{aligned} \text{(QP)} \quad \max \quad & \tfrac{1}{2} \sum_{(i,j)} (1 - y_i y_j) w_{ij} \\ s.t. \quad & y_i \in \{-1, 1\} \qquad\qquad i = 1, \ldots, n. \end{aligned}$$

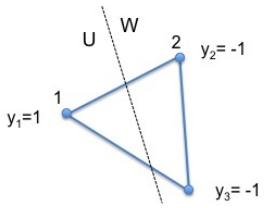


Figure 19: The maximum cut in this graph on 3 vertices has value 2.

Note that the constraint $y_i \in \{-1, 1\}$ can also be written as $y_i^2 = 1$. It is easy to see that this is indeed an exact formulation of the max cut problem: Given a feasible solution for (QP) we get a cut with the same value by taking $i \in U$ if $y_i = 1$ and taking $i \in W$ if $y_i = -1$. Conversely, given a cut (U, W) we get a feasible solution for (QP) with the same value by setting $y_i = 1$ if $i \in U$ and taking $y_i = -1$ if $i \in W$.

The quadratic program (QP) is strict and we can formulate its vector program relaxation.

$$\begin{aligned} \text{(VP)} \quad \max \quad & \tfrac{1}{2} \sum_{(i,j)} (1 - v_i \cdot v_j) w_{ij} \\ s.t. \quad & v_i \cdot v_i = 1, \ v_i \in \mathbb{R}^n \qquad i = 1, \ldots, n. \end{aligned}$$

It is easy to see that (VP) really is a relaxation of (QP): Let $y_1, y_2, \ldots, y_n$ be feasible for (QP), then $v_i = (y_i, 0, \ldots, 0)$ for all i defines a feasible solution for (VP) with the same value. In the graph of Figure 19, an optimal solution to it quadratic program is $(y_1, y_2, y_3) = (1, -1, -1)$ and the corresponding solution for (VP) is $v_1 = (1, 0, 0), v_2 = v_3 = (-1, 0, 0)$. However, this is not an optimal solution to (VP). An optimal solution is given in Figure 20.

The max-cut algorithm.

Goemans and Williamson gave the following algorithm for the max-cut problem:

- **Step 1**. Solve the VP-relaxation $\rightarrow v_1^*, v_2^*, \ldots, v_n^*, Z_{VP}^*$.

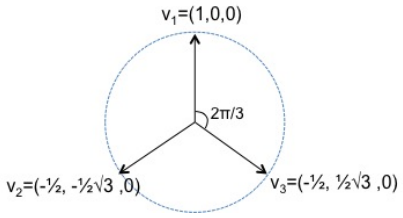


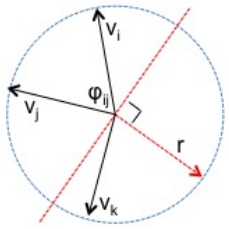
Figure 20: An optimal solution for the vector program (VP) for the graph of Figure 19. The inproduct $v_i \cdot v_j$ is -0.5 for all pairs i, j. Hence, the optimal value is $3 \cdot \frac{1}{2} 1.5 = 2.25$.

- **Step 2**. (Randomized rounding) Take a vector $r \in \mathbb{R}^n$ of length 1 uniformly at random. Add vertex i to U if $v_i^* \cdot r \geqslant 0$ and add it to W otherwise.

Analysis: Let φ_{ij} be the angle between v_i^* and v_j^*. Then, $v_i^* \cdot v_j^* = \cos \varphi_{ij}$ and we can express the optimal value of the VP in terms of φ_{ij}:

$$Z_{VP}^* = \frac{1}{2} \sum_{(i,j)} (1 - v_i^* \cdot v_j^*) w_{ij} = \frac{1}{2} \sum_{(i,j)} (1 - \cos \varphi_{ij}) w_{ij}. \tag{14}$$

On the other hand,



$$\begin{aligned} & \Pr\{ \text{ edge } (i,j) \text{ is in the cut } \} \\ = \; & \Pr\{v_i^* \text{ and } v_j^* \text{ are separated by the hyperplane perpendicular to } r.\} \\ = \; & \frac{\varphi_{ij}}{\pi} \end{aligned}$$

Let W be the weight of the cut. Then

$$\mathbb{E}[W] = \sum_{(i,j)} \frac{\varphi_{ij}}{\pi} w_{ij}. \tag{15}$$

Let

$$\alpha = \min_{0 \leqslant \varphi \leqslant \pi} \left[\frac{\varphi}{\pi} \cdot \frac{2}{1 - \cos \varphi} \right].$$

One can show that the minimum α of this function is well-defined and satisfies: $\alpha > 0.878$. Consequently, for any φ_{ij} we have

$$\frac{\varphi_{ij}}{\pi} \geqslant \alpha \frac{1 - \cos \varphi_{ij}}{2}.$$

Now we combine (14) and (15):

$$\mathbb{E}[W] = \sum_{(ij)} \frac{\varphi_{ij}}{\pi} w_{ij} \geqslant \frac{\alpha}{2} \sum_{(i,j)} (1 - \cos \varphi_{ij}) w_{ij} = \alpha Z^*_{VP} \geqslant \alpha \textsc{Opt} > 0.878 \textsc{Opt}.$$

The algorithm can be derandomized using a sophisticated application of the method of conditional expectations (See section 5.2) but this is not easy.

The algorithm above is best possible if $\mathcal{P} \neq \mathcal{NP}$ and if the so called unique games conjecture holds. (See Section 13.3 for this unique games conjecture.)

Extra (not in book): The max 2-sat problem.

Section 5.1 gives a 1/2-approximation for the maximum satisfiability problem. In particular, it gives a $1-(1/2)^k$-approximation if each clause contains at least k literals. For the max 2-sat problem, in which each clause contains exactly two literals, this yields a 3/4-approximation. Also, the linear program of Section 5.4 yields a 3/4-approximation when rounded as in Section 5.6. Here, we show that the approach used for the max-cut problem gives an α-approximation with the same factor $\alpha = 0.878...$ for the max 2-sat problem. We shall restrict to the unweighted version but the algorithm works exactly the same in case each clause had a given weight and we want to maximize the total weight of the satisfied clauses. In the example below, $\textsc{Opt} = 3$. The examples also shows the four possible types of clauses.

$$x_1 \vee x_2, \qquad \neg x_1 \vee x_2, \qquad \neg x_2 \vee x_1, \qquad \neg x_1 \vee \neg x_2.$$

Let us denote $val(x_i) = 1$ if x_i is true and let it be zero otherwise. Similarly, for clause C let $val(C) = 1$ if the clause is true and let it be zero otherwise.

An obvious choice for a quadratic program is to define binary variables $y_i \in \{0, 1\}$ with $y_i = val(x_i)$. Then the value of a clause $x_i \vee x_j$ can be expressed as

$$val(x_i \vee x_j) = 1 - (1 - y_i)(1 - y_j) = y_i + y_j - y_i y_j.$$

We can do the same for the other 3 types, $(\neg x_i \vee x_j), (\neg x_i \vee x_j), (\neg x_i \vee \neg x_j)$, and formulate a quadratic program. However, its relaxation is not convex and nor is it a strict quadratic program. We need to try another approach.

A strict quadratic program. We introduce one more binary variable, y_0 and make the following proposition:

$$\begin{aligned} & & y_i = y_0 &\Leftrightarrow x_i = \text{true} \\ y_i \in \{-1,1\} \text{ and } & & y_i \neq y_0 &\Leftrightarrow x_i = \text{false} \end{aligned}$$

Now, the value of a literal can be expressed as follows.

$$val(x_i) = \frac{1+y_0y_i}{2} \quad \text{and} \quad val(\neg x_i) = \frac{1-y_0y_i}{2}.$$

Then, the value of a clause $x_i \vee x_j$ becomes

$$\begin{aligned} val(x_i \vee x_j) &= 1 - val(\neg x_i)val(\neg x_j) \\ &= 1 - \left(\frac{1-y_0y_i}{2}\right)\left(\frac{1-y_0y_j}{2}\right) \\ &= \frac{1+y_0y_i}{4} + \frac{1+y_0y_j}{4} + \frac{1-y_iy_j}{4}. \end{aligned}$$

In the same way, we can write

$$\begin{aligned} val(\neg x_i \vee x_j) &= \frac{1-y_0y_i}{4} + \frac{1+y_0y_j}{4} + \frac{1+y_iy_j}{4} \\ val(x_i \vee \neg x_j) &= \frac{1+y_0y_i}{4} - \frac{1-y_0y_j}{4} + \frac{1+y_iy_j}{4} \\ val(\neg x_i \vee \neg x_j) &= \frac{1-y_0y_i}{4} + \frac{1-y_0y_j}{4} + \frac{1-y_iy_j}{4}. \end{aligned}$$

We see that any instance of max 2-sat can be written as a strict quadratic program of the following form. Again, to simplify notation we write $\sum_{(i,j)}$ to denote the summation over i, j with $0 \leqslant i < j \leqslant n$.

$$\begin{aligned} \text{(QP)} \quad \max \quad & \sum_{(i,j)} a_{ij}(1+y_iy_j) + b_{ij}(1-y_iy_j) \\ s.t. \quad & y_i^2 = 1, \qquad i = 0,1,\ldots,n. \end{aligned}$$

Here, a_{ij} and b_{ij} are constants depending on the instance. The vector program relaxation becomes

$$\begin{aligned} \text{(VP)} \quad \max \quad & \sum_{(i,j)} a_{ij}(1+v_i \cdot v_j) + b_{ij}(1-v_i \cdot v_j) \\ s.t. \quad & v_i \cdot v_i = 1,\ v_i \in \mathbb{R}^{n+1} \qquad i = 0,1,\ldots,n. \end{aligned}$$

Algorithm:

- **Step 1**. Solve the VP-relaxation $\rightarrow v_0^*, v_1^*, \ldots, v_n^*, Z_{VP}^*$.
- **Step 2**. Take a vector $r \in \mathbb{R}^n$ of length 1 uniformly at random. For $i = 0,\ldots,n$, let $y_i = 1$ if if $v_i^* \cdot r \geqslant 0$ and let $y_i = -1$ otherwise.

Analysis: Let φ_{ij} be the angle between v_i^* and v_j^*. Then, $v_i^* \cdot v_j^* = \cos\varphi_{ij}$. The optimal VP value can be written as follows.

$$Z_{VP}^* = \sum_{(i,j)} a_{ij}(1+v_i^*\cdot v_j^*)+b_{ij}(1-v_i^*\cdot v_j^*) = \sum_{(i,j)} a_{ij}(1+\cos\varphi_{ij})+b_{ij}(1-\cos\varphi_{ij}).$$

On the other hand, let W be the number of clauses satisfied by the algorithm. As we have seen for the max cut problem, $\Pr(y_i \neq y_j) = \varphi_{ij}/\pi$. Hence,

$$\begin{aligned} E[W] &= 2\sum_{(i,j)} a_{ij}\Pr(y_i = y_j) + b_{ij}\Pr(y_i \neq y_j) \\ &= 2\sum_{(i,j)} a_{ij}(1-\frac{\varphi_{ij}}{\pi}) + b_{ij}\frac{\varphi_{ij}}{\pi}. \end{aligned} \tag{16}$$

Let $\alpha \approx 0.878$ be defined as in the max cut analysis. That means

$$\frac{\varphi}{\pi} \geqslant \frac{\alpha}{2}(1-\cos\varphi), \text{ for all } 0 \leqslant \varphi \leqslant \pi.$$

On the other hand, when we substitute $\varphi = \pi - \theta$ then, for all $0 \leqslant \theta \leqslant \pi$

$$\begin{aligned} \frac{\pi-\theta}{\pi} \geqslant \frac{\alpha}{2}(1-\cos(\pi-\theta)) \quad &\Leftrightarrow \\ 1-\frac{\theta}{\pi} \geqslant \frac{\alpha}{2}(1+\cos\theta). \end{aligned}$$

Now, apply the two inequalities above to (16):

$$\begin{aligned} E[W] &= 2\sum_{(i,j)} a_{ij}(1-\frac{\varphi_{ij}}{\pi}) + b_{ij}\frac{\varphi_{ij}}{\pi} \\ &\geqslant \alpha\sum_{(i,j)} a_{ij}(1+\cos\varphi_{ij}) + b_{ij}(1-\cos\varphi_{ij}) \\ &= \alpha Z_{VP}^* \geqslant \alpha\textsc{Opt}. \end{aligned}$$

6.5 Coloring 3-colorabale graphs.

A graph coloring is a labeling of the vertices with labels $1, 2, \ldots$ (the colors) such that any two adjacent vertices have a different label. The chromatic number $\chi(G)$ is the minimum number of colors needed for a feasible coloring of G. If $\chi(G) \leqslant 2$ then finding a feasible 2-coloring is easy: Start with any vertex and give it color 1, then color all its neighbors with color 2, color all neighbors of neighbors with color 1, and so on. Hence, deciding if a graph is 2-colorable is easy. Also, any graph can be colored using at most $\Delta + 1$ colors if Δ is the maximum degree of the graph. (Simply color the vertices one-by-one.)

(i) Finding a 2-coloring is easy if the graph is 2-colorable.

(ii) Finding a coloring with at most $\Delta + 1$ colors is easy, where Δ is the maximum degree in the graph.

On the other hand, deciding if $\chi(G) \leqslant 3$ is NP-complete. In fact, even when we know that the given graph is 3-colorable then finding a coloring with a small number of colors is difficult. First we give a simple algorithm to color the vertices of a 3-colorable graph $G = (V, E)$ with at most $4\sqrt{n}$ colors. Let $\sigma = \sqrt{n}$.

Algorithm 1:

- **Step 1** As long as there is a vertex v with degree at least σ do the following: Use a new color for v and use two new colors to color the neighbors of v. Remove v and its neighbors from G.
- **Step 2** Color the remaining graph with at most σ colors.

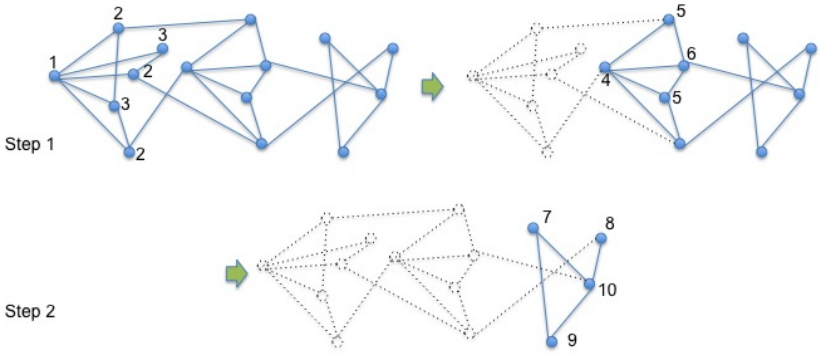


Figure 21: Application of Algorithm 1 with $\sigma = 4$.

Clearly, Step 1 can be done in polynomial time since the set of neighboring points of any point v is 2-colorable. Also, Step 2 is easy since the graph that remains after Step 1 has maximum degree $\Delta \leqslant \sigma - 1$. The total number of colors used is

$$3 \cdot n/(\sigma + 1) + \sigma < 4\sqrt{n}.$$

We can reduce the number of colors by improving (ii). Note that in general (ii) is tight since for a complete graph on n vertices we need $n = \Delta + 1$ colors. However, if the graph is 3-colorable then we can do better. We use the notation $\tilde{O}(.)$ to hide factors $\log n$. For example, $4n^2 \log^3 n = \tilde{O}(n^2)$.

Theorem 6.2. *If G is 3-colorable, then a coloring with $\tilde{O}(\Delta^{\log_3 2})$ colors in expectation can be found in polynomial time. Note,* $\log_3 2 \approx 0.631$.

Before proving Theorem 6.2, let us see how it leeds to a reduced approximation ratio. We apply the same algorithm but only need $\tilde{O}(\Delta^{\log_3 2}) = \tilde{O}(\sigma^{\log_3 2})$ colors in Step 2. That means, the total number of colors used is bounded by

$$3 \cdot n/(\sigma + 1) + \tilde{O}(\sigma^{\log_3 2})$$

Now we optimize over σ. Solving $n/\sigma = \sigma^{\log_3 2}$ gives $\sigma = n^{\log_6 3}$. Then, for both steps the number of colors used is $\tilde{O}(n/\sigma) = \tilde{O}(n^{\log_6 2})$, which is better than the $4\sqrt{n}$ we have seen earlier since $\log_6 2 \approx 0.387$.

Proof of Theorem 6.2

The idea is similar to that of the max-cut problem. There are two differences to notice. First, we do not formulate an exact quadratic program but only give a vector program relaxation for the 3-coloring problem. Second, the algorithm takes many random vectors in stead of just one.

$$\begin{array}{lll} \text{(VP)} \quad \min & \lambda & \\ s.t. & v_i \cdot v_j \leqslant \lambda, & \text{for all } (i,j) \in E \\ & v_i \cdot v_i = 1, v_i \in \mathbb{R}^n & \text{for all } i \in V. \end{array}$$

If $G = (V, E)$ is 3-colorable then the vector program above has optimal value at most $\lambda^* \leqslant -0.5$. This can be seen from the solution of the max cut example in Figure 20. Given a feasible 3-coloring, let the three vectors correspond with the three colors. If $(i, j) \in E$ then i and j have different colors which implies that $v_i \cdot v_j = -0.5$.

Remark: It is not true that $\lambda^* = -0.5$ for any graph with $\chi(G) = 3$ as can be seen from Figure 22.

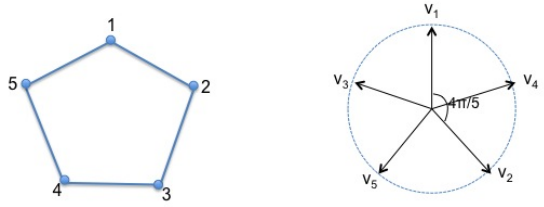


Figure 22: A cycle on 5 vertices has chromatic number 3. The optimal solution to VP in this case has value $\lambda^* = \cos(4\pi/5) < -0.5$. For an odd cycle, $\lambda^* \to -1$ for $n \to \infty$.

The idea of the algorithm is as follows. First solve the VP. Then take $t \leqslant n$ unit vectors in $\mathbb{R}^n$ at random. The corresponding hyperplanes partition $\mathbb{R}^n$ in

2^t regions. Let the regions correspond to colors. Say that a vertex is *properly colored* if all its neighbors have a different color. Remove all properly colored vertices and take a new set of t unit vectors in $\mathbb{R}^n$ at random and use 2^t new colors. Repeat until all vertices are properly colored.

Algorithm:

- **Step 1**. Solve the VP-relaxation $\rightarrow v_1^*, \ldots, v_n^*, \lambda^*$.
- **Step 2**. Repeat until the graph is properly colored:
  Take vectors $r_1, \ldots, r_t \in \mathbb{R}^n$ of length 1 uniformly at random. Each of the 2^t regions defines a color. Give point i the color of the region containing v_i^*. Say that a vertex is *properly colored* if all its neighbors have a different color. Remove all properly colored vertices from the graph.

Consider an edge $(i,j) \in E$ and assume i and j are not yet properly colored. Then, for the next iteration of Step 2:

$$\begin{aligned}
& \Pr\{ \text{ } i \text{ and } j \text{ get the same color } \} \\
= \; & \Pr\{v_i^* \text{ and } v_j^* \text{ are not separated by any of the } r_i\text{'s.}\} \\
= \; & \left(1 - \frac{\varphi_{ij}}{\pi}\right)^t \\
= \; & \left(1 - \frac{1}{\pi}\arccos(v_i^* \cdot v_j^*\right)^t \\
\leq \; & \left(1 - \frac{1}{\pi}\arccos(\lambda^*)\right)^t \\
\leq \; & \left(1 - \frac{1}{\pi}\arccos(-1/2)\right)^t = \left(1 - \frac{1}{\pi}\frac{2\pi}{3}\right)^t = \left(\frac{1}{3}\right)^t
\end{aligned}$$

The first inequality follows from (VP) and from the fact that arccos is non-increasing. The second inequality follows from $\lambda^* \leq -1/2$ and from arccos being non-increasing. Take $t = 1 + \log_3 \Delta$. Then,

$$\Pr\{ \text{ } i \text{ and } j \text{ get the same color } \} \leqslant \frac{1}{3\Delta}.$$

By the 'union bound', for any vertex i in an iteration of Step 2:

$$\Pr\{ \text{ } i \text{ is not properly colored } \} \leqslant \Delta\frac{1}{3\Delta} = \frac{1}{3}.$$

Let n' be the number of remaining vertices before some iteration of Step 2. Then, for this iteration

$$E[\text{ \# vertices not properly colored }] \leqslant \frac{n'}{3}.$$

By Markov's inequality:

$$\Pr\{ \text{ \# vertices not properly colored } \geqslant \frac{2n'}{3}\} \leqslant \frac{1}{2}.$$

Hence, in each iteration of Step 2, the number of vertices still to be colored is reduced by a factor 2/3 with probability at least 1/2. This implies that the expected number of iteration of Step 2 is $O(\log n)$ and the expected number of colors used in total is

$$O(\log n)2^t = \tilde{O}(2^t) = \tilde{O}(2^{1+\log_3 \Delta}) = \tilde{O}(\Delta^{\log_3 2}).$$

References

[1] Papadimitriou and Steiglitz, *Combinatorial optimization: Algorithms and complexity*, Dover publications, 1998.