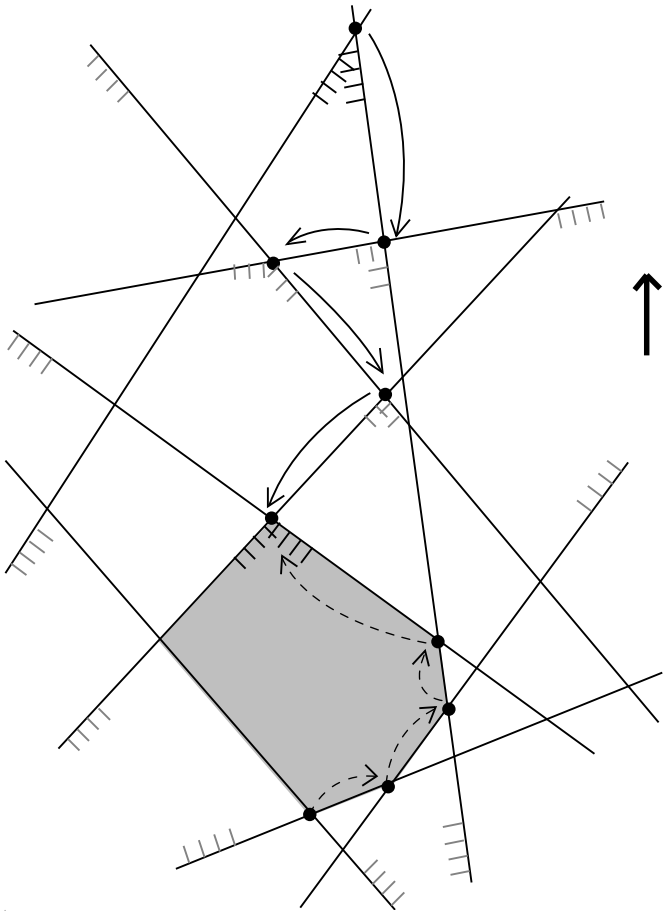

Figure 1: Primal (dashed) and Dual (solid) Simplex Algorithm for $d = 2$

Lecture May, 13th

0.1 Primal and Dual Simplex

In the previous sections we have seen an algorithm to determine the highest feasible point x^* for a collection of constraints $\mathscr{H} = \{a_i^T x \leqslant \beta_i\}$ by approaching x^* from the infeasible region and jumping to v-shapes of decreasjng height. This algorithm was called *(Primal) Simplex* algorithm. In the following we will briefly present the so-called *(Dual) Simplex* algorithm, which approaches x^* via vertices of the feasible region defined by $\mathscr{H}$.

	Dual (v-shape) Simplex	Primal Simplex
Idea	jump to lower and lower vertices of v-shapes	jump to higher and higher feasible vertices
Basis B (vertex repr.)	d lin.indep. constraints	d lin.indep. constraints
Invariant	always have v-shape, i.e. c is conic combination of a_i's def. v-shape	always corner of feasible region, i.e. $\nexists i$ s.t. $a_i^T x_v > \beta_i$
Non-Opt.	v-shape not feasible, i.e. $\exists i$ s.t. $a_i^T x_v > \beta_i$	staying on $d-1$ constraints of B one can move away from x_v and improve without becoming infeasible
Pivot	determine new v-shape including a_i and drop one of the old constraints	determine which constraint blocks this movement, include that in basis, remove constraint which we moved away from
Degen.	constraints orthogonal to c (can be fixed by perturbing c)	vertices determined by more than d constraints (fixed by perturbing constr.)
Start	some v-shape; if not given consider auxiliary problem $a_i^T x \leqslant 0$ with add. constraints $x_i \leqslant 1$ for $c_i \geqslant 0$, $-x_i \leqslant 1$ for $c_i < 0 \ldots$	some corner of the feasible region; if not given $\rightarrow$ Exercise !

Basically both variants are equivalent and some people also call primal simplex what we call dual simplex and vice versa.

Exercise 1. Fill in the details of the dual simplex algorithm.

Pivot Step

Let us first try to figure out which of the $d-1$ halflines to follow to improve our objective function value. Our current solution is given by d constraints $a_1^T x \leqslant \beta_1 \ldots a_d^T x \leqslant \beta_d$. We can write this system of inequalities as equalities by introducing slack variables $s = (s_1, \ldots s_d) \in \mathbb{R}^d_{\geqslant 0}$:

$$\begin{pmatrix} a_{11} & \ldots & a_{1d} \\ \ldots & \ldots & \ldots \\ a_{d1} & \ldots & a_{dd} \end{pmatrix} x + \begin{pmatrix} s_1 \\ \vdots \\ s_d \end{pmatrix} = \begin{pmatrix} \beta_1 \\ \vdots \\ \beta_d \end{pmatrix}$$

We will write this as $A_B x + s = \beta_B$ in short. The coordinates x' of the current vertex are determined as

$$x = A_B^{-1}\beta_B - A_B^{-1}s$$

with $s = (0, \ldots, 0)$, i.e. all d constraints are tight. Moving away from one constraint corresponds to increasing its slack variable from zero to some positive value. We want to figure out which slack variable we can increase to get a higher objective function value. For the objective function we have:

$$c^T x = c^T (A_B^{-1}\beta_B - A_B^{-1}s) = c^T x' - \alpha_1 s_1 - \cdots - \alpha_d s_d$$

that means expressed in dependence of s, the objective function value is a constant (the value at our current vertex position) and a sum of s_i variables with respective coefficients. If we have for one i $\alpha_i < 0$, then increasing this slack variable – i.e. moving away from the corresponding constraint and sliding along the half-line determined by the remaining constraints – improves the objective function. If no such i exists, we are optimal !

Now assume we have found a constraint $a_i^T x \leqslant \beta_i$ which is going to leave the basis (as we can increase the objective function value thereby). It remains to find the constraint which blocks this movement (if no such constraint exists, the problem is clearly unbounded).

$x' = A_B^{-1}\beta_B$ is the current vertex, choosing $x'' = A_B^{-1}\beta_B - (A_B^{-1})_{\cdot i}$ (this is a point moved one unit in the direction we have just determined), we can move on the improving ray ($\lambda \geqslant 0$)

$$\overrightarrow{r} = x' + \lambda \cdot (x'' - x') = x' - \lambda (A_B^{-1})_{\cdot i}$$

and ask by which other constraint j this movement is blocked first. We can plug in a point $r(\lambda)$ on the ray in every constraint l:

$$a_l^T (x' - \lambda (A_B^{-1})_{\cdot i}) \leqslant \beta_l$$

$$\Leftrightarrow a_l^T x' - \lambda a_l^T (A_B^{-1})_{\cdot i}) \leqslant \beta_l$$

If $a_l^T (A_B^{-1})_{\cdot i}) >= 0$ this constraint will never block our movement, otherwise we obtain the following bound on λ:

$$\lambda \leqslant \frac{\beta_l - a_l^T x'}{-a_l^T (A_B^{-1})_{\cdot i})}$$

If no constraint gives a bound on our movement, the problem is unconstrained, otherwise the constraint j which determines the smallest upper bound on λ is the first constraint hit when moving along $\overrightarrow{r}$. This constraint enters the basis in exchange for constraint i.

Perturbation

One needs to show that one can perturb all constraints by adding some small constant to

the right such that no $d+1$ constraints intersect in one point.

Starting Solution
Consider the auxiliary problem min λ s.t. $a_i^T x - \lambda \leqslant \beta_i$, $\lambda \geqslant 0$. Take any d lin.indep. constraints. Either they already form a feasible solution or otherwise compute their line l of intersection (parametrized in λ). The smallest $\lambda > 0$ that is necessary to make a point on l feasible is determined by some constraint. Take this and the other d constraints and use them as first feasible basis of the auxiliary problem. If the auxiliary problem has a solution with $\lambda = 0$, original problem is feasible and we can also read off a first feasible basis, o.w. original problem was infeasible.

0.2 Linear Programming by Prune and Search

0.2.1 Prune and Search in $\mathbb{R}^2$

In the following we will first restrict to the two-dimensional case. Higher dimensions will be treated at the end of this part.

We are given a set $\mathscr{H}$ of n constraints $a_i^T x \leqslant \beta_i$, $x \in \mathbb{R}^2$ and want to determine the highest point $x^*(x_1^*, x_2^*)$ (i.e. maximizing $c^T x$ with $c^T = (0,1)$) which is feasible or certify that the set $\mathscr{H}$ is infeasible. The optimal solution is determined by $d = 2$ constraints, so in some sense all other constraints are not really important. $\Rightarrow$ Idea: drop more and more of the constraints of which we can be sure that they do not define the optimum until only d constraints are left.

Remark 0.1. *This strategy is different from the simplex approach, where kicked out constraints can reenter the basis lateron (easy for the primal simplex, even in $d = 2$, for the dual simplex only for $d \geqslant 3$).*

Exercise 2. Give an example for $d = 2$ and a sequence of valid pivot steps where one of the two simplex algorithms presented has a constraint leaving and then reentering into the basis.

Before we can describe the algorithm let us partition $\mathscr{H} = \mathscr{H}^+ \uplus \mathscr{H}^-$ where $\mathscr{H}^+ = \{a_i^T x \leqslant \beta_i, a_{i2} < 0\}$, and $\mathscr{H}^- = \{a_j^T x \leqslant \beta_j, a_{j2} > 0\}$. For the further description of the algorithm it will be useful to define a function

$$g(x_1) = \min_{j \in \mathscr{H}^-} \{x_2 : a_{j1}x_1 + a_{j2}x_2 = \beta_j\} - \max_{i \in \mathscr{H}^+} \{x_2 : a_{i1}x_1 + a_{i2}x_2 = \beta_i\}$$

Clearly if $\exists x_1$ with $g(x_1) \geqslant 0$, the problem is feasible, if such x_1 does not exist, the problem is infeasible. The algorithm we present in the following will

- compute the highest $x^*(x_1^*, x_2^*)$ if the problem is feasible and bounded
- report if the problem is feasible and unbounded
- compute some x_1^* with $g(x_1^*)$ maximal in $\mathbb{R}$ if the problem is infeasible.

Throughout the algorithm we will maintain an interval $I = (l, r)$ where x_1^* of the potential optimal solution or the "least violated" position is contained. At the beginning $I = (-\infty, +\infty)$ or set according to potential vertical constraints (they are ignored in the following).

Pruning Pair all the constraints arbitrarily within $\mathscr{H}^+$ and $\mathscr{H}^-$. Look at one of the resulting $\lfloor n/2 \rfloor$ intersection points. If some oracle could tell us where the optimum/least violating solution lies w.r.t. the vertical line through the intersection point, we could drop one of the two constraints!

Searching Given some vertical line l inside our current range, we are interested in the question whether the optimum/least violating solution is to the left or to the right of l. To determine this, consider the order of all constraints on l, in particular the highest constraint $h^+ \in \mathscr{H}^+$ and the lowest constraint $h^- \in \mathscr{H}^-$. In the following we assume that they are uniquely defined. We look at the intersection points $p^+ = h^+ \cap l$ and $p^- = h^- \cap l$. We have several cases to consider:

1. $p_y^+ > p_y^-$: then our problem is infeasible at this position. If there h^+ and h^- are parallel there is clearly no feasible region anywhere and the violation cannot decrease. Otherwise, the region of feasibility or the region where the violation is less than at the current position has to be on the same side of l as the intersection point of h^+ and h^-.

2. $p_y^+ \leqslant p_y^-$: our problem is feasible at this position but we have not necessarily found an optimum solution. But looking at the environment of the intersection point $h^- \cap l$ we can deduce, on which side we might get higher.

The Algorithm Clearly we could call the search step for each of the paired constraints, but as one search step costs O(n) time, we will not end up with anything better than $\Omega(n^2)$. So the idea is to use one search step to perform many prune steps.

PruneAndSearch ($\mathscr{H}$, (l,r))

1. if $|\mathscr{H}|$ constant, solve trivially
2. partition $\mathscr{H}$ into $\mathscr{H}^+ \uplus \mathscr{H}^-$
3. pair constraints within $\mathscr{H}^+$ and $\mathscr{H}^-$ to obtain set of intersection points P
4. for all intersection points $p \notin (l,r)$, drop one of the defining constraints from $\mathscr{H}$ and delete p from P
5. for the remaining intersection points compute their x_1-median m; use the search step to determine on which side of the vertical median $x_1 = m$ line the optimum/least violating solution lies
6. replace (l,r) by (m,r) or (l,m) depending on the outcome of the search step
7. prune intersection points which are outside the new range
8. recurse on remaining constraints and new range: **PruneAndSearch($\mathscr{H}'$,(l',r'))**

Analysis Starting with $n = |\mathscr{H}|$ constraints, the algorithm spends O(n) time and one recursive call on n' constraints to solve the problem. How large can be n' ?

Ignoring floors and ceilings, we have $n/2$ intersection points, at least half of which fall outside the new range (l',r') of the recursive call. That means at least $n/4$ constraints can be removed from $\mathscr{H}$. Therefore $n' \leqslant \frac{3}{4}n$ and hence we get an overall running time of O(n).

Exercise 3. Assume that excluding the recursive call the algorithm spends $c \cdot n$ work in one call, and exactly $1/4$th of the constraints is pruned in each step. Give an upper bound for the running time of the algorithm (no O-notation!), i.e. solve the recursion $T(n) = c \cdot n + T(\frac{3}{4}n)$ with $T(1) = 1$.

Lecture May, 15th

Recap: Primal/Dual picture, P'n S in 2 dim.; treatment of degeneracies

0.2.2 Prune and Search in $\mathbb{R}^3$ and higher dimensions

The same approach presented for the two-dimensional setting basically also works in 3 and higher dimensions. In the following we will briefly sketch a solution of a simplified version of the problem in 3 dimensions.

We are given a set $\mathscr{H}$ of n constraints $a_i^T x \leqslant \beta_i$, $x \in \mathbb{R}^3$ and want to determine the highest point $x^*(x_1^*, x_2^*, x_3^*)$ (i.e. maximizing $c^T x$ with $c^T = (0,0,1)$) which is feasible or certify that the set $\mathscr{H}$ is unbounded. The optimal solution is determined by $d = 3$ constraints, so all other constraints not defining the optimum are not really important. Again the idea will be to drop more and more of the constraints of which we can be sure that they do not define the optimum until only $d = 3$ constraints are left.

As a simplification we assume that all constraints are of the form $a_i^T x \leqslant \beta_i, a_{i3} > 0$, i.e. all constraints are halfplanes with the feasible region "below". Clearly for this special case, the linear program is definitely feasible (we just have to go down far enough). The core idea of prune and search in three dimensions is already exhibited in this simplified formulation. The general case where all types of constraints are allowed can also be solved but involves some technicalities.

Our algorithm will compute a highest point x^* or report that the problem is unbounded.

Pruning Pair all the constraints arbitrarily as we have done in the two-dimensional case. Look at one of the resulting $\lfloor n/2 \rfloor$ lines of intersection, let's call it l. If some oracle could tell us where the optimum lies w.r.t. to the vertical plane through l, we could clearly drop one of the two constraints.

Searching Given some vertical plane w, we are interested in the question whether the optimum solution is to the left or to the right of w.

To answer this, consider the intersection of all constraints with this vertical plane w. In w these intersections form a feasible two-dimensional highest point problem. Using the algorithm presented for $d = 2$ we can solve this problem in $O(n)$ time and either determine a optimal point x^* or certify that the problem is unbounded. In the latter case we are finished as then our problem in three dimensions is also unbounded. In the former case, consider the two halfplanes h_1, h_2 that together with w intersect in x^*. Looking at the slope of the intersection line $h_1 \cap h_2$ we can easily determine on which side of w the optimum lies.

The Algorithm Again, as in the two-dimensional case, we could call the search step for each of the paired constraints, but as one search step costs O(n) time, we will not end up with anything better than $\Omega(n^2)$. So the idea is to use a constant number of search steps to perform linearly many prune steps.

PruneAndSearch ($\mathscr{H}$)

1. if $|\mathscr{H}|$ constant, solve trivially
2. pair all the constraints to obtain set of intersection lines L
3. project all lines in L into the x_1x_2-plane and transform them such that half of them have slope > 0 (these are the lines L^+), half of them slope < 0 (these are the lines L^-).
4. pair all the lines, always one from L^+, one from L^- and consider the set P of resulting $n/4$ intersection points in the x_1x_2-plane
5. Query the search oracle with the vertical plane $x_1 = m_1$ where m_1 is the x_1 median of P
6. W.l.o.g. assume that the first query reports that the optimum lies in $x_1 > m_1$, consider all the intersection points P with $x_1 < m_1$. Query the search oracle with the vertical plane $x_2 = m_2$, where m_2 is the x_2 median of all intersection points from P with $x_1 < m_1$.
7. W.l.o.g. assume the second query returns that x^* lies in $x_2 > m_2$, then for all $n/16$ intersection points with $x_1 < m_1, x_2 < m_2$, one constraint can be dropped
8. recurse on remaining constraints and new range: **PruneAndSearch($\mathscr{H}'$)**

Exercise 4. Given a set of n lines in the plane by their line equations (all have different slopes, $n = 2k$ for some k). Describe a procedure to rotate the whole arrangement of lines such that half of the lines have slope > 0, half of the lines slope < 0.

Analysis Starting with $n = |\mathscr{H}|$ constraints, the algorithm spends O(n) time and one recursive call on n' constraints to solve the problem. How large can be n' ?

Ignoring floors and ceilings, we have $n/2$ lines of intersection, and $n/4$ intersection points. For a quarter of them one of the four defining constraints can be dropped. That means at least $n/16$ constraints can be removed from $\mathscr{H}$. Therefore $n' \leqslant \frac{15}{16}n$ and hence we get an overall running time of O(n).

Higher Dimensions Basically the same approach also works in higher dimensions, but the fraction of constraints that can be dropped in one recursive call drops drastically, in fact doubly exponentially in d. Remember, in $d = 2$ using one oracle query, we could decide $\frac{n}{4}$ sideness problems, in $d = 3$, using two oracle queries, we could decide on $\frac{n}{16}$ sideness problems. In general, using 2^{d-2} oracle queries, $\frac{n}{2^{2^{d-1}}}$ sideness tests can be decided. So for the running time we get

$$T_d(n) = c_d n + T_d(1 - \frac{1}{2^{2^{d-1}}}) \leqslant c_d \cdot 2^{2^{d-1}} n$$

0.2.3 Seidel's LP algorithm for fixed dimension

In the presentation of the v-shape-Simplex algorithm, there was some ambiguity in the formulation (which did not affect its correctness, though). Given some v-shape v, there might be many constraints which are violated by the vertex x_v; we did not specify which of these constraints to select into our basis. Roghly speaking, Seidel's LP algorithm can be regarded as a refinement of the v-shape-Simplex algorithm which gives more precise rules for which violating constraint to choose. His algorithm will also run in O(n) time for fixed dimension d, but with a better dependence on d, namely O($d!n$).

For the following presentation we assume:

- the feasible region is non-empty and is contained in the positive orthant ($x_i \geqslant 0$)
- and we are looking for the lowest point in the feasible region (this implies that the origin is already a v-shape).
- non-degeneracy

The intuition behind the algorithm is that if the dimension is very small compared to the number of constraints, most of the constraints are not important for the optimum solution, so when dropping one random constraint, chances are good that the optimum solution does not change.

A call SeidLP(H) to Seidel's LP algorithm returns the optimal solution to the set of constraints H. It can be stated as follows:

SeidLP(H)

1. if $|H| = 1$ or $|d = 1$ return OPT(H)
2. choose $h \in H$ uniformly at random
3. $v \leftarrow$ SeidLP($H - \{h\}$)
4. if $v \notin h$ then $v \leftarrow$ SeidLP($\{H - \{h\}\} \downarrow_h$)
5. return v

OPT(H) is the optimal solution of the constraints in H and the $x_i \geqslant 0$ constraints.

To show correctness, we only need the following small Proposition:

Proposition 0.2. *Let H be a set of constraints, $h \in H$. Denote by $OPT_{H-\{h\}}$ the optimal solution for the set $H-\{h\}$, and OPT_H the optimal solution of the set H with basis B_H.*

Then we have $v_{H-\{h\}} \notin h \Leftrightarrow h \in B_H$.

Exercise 5. How could infeasibility be detected in the course of the algorithm ?

Analysis For the analysis observe that step one can be performed in O(n) or O(d) time respectively. Picking a random constraint can be done in $O(1)$ time, the violation test takes O(d) time. If the violation test fails, $O(dn)$ time and the one recursive call in one dimension lower is required.

For a call SeidLP(H) let d denote the dimension of the constraints in H. The crucial step in the analysis is Nr.4. We are interested how likely it is that this second recursive call occurs. Assuming non-degeneracy, the optimum solution of H is determined by a unique set of d constraints. According to our proposition, the violation test fails if and only if the picked constraint h is in the optimum solution of H. As h was chosen uniformly at random, the probability for that is at most d/n. So The following recursion describes the running time of the algorithm for a call on n constraints in dimension d:

$$T(n,d) \leqslant T(n-1,d) + O(d) + \frac{d}{n}(O(dn) + T(n-1, f-1))$$

Theorem 0.3. *The running time of SeidLP for n constraints in d dimensions is $O(d!n)$.*

Proof. Invariant: $T(n,d) \leqslant 2 \cdot d!n\sum_{i=1}^{d} \frac{i^2}{i!}$ for $n+d < x$.

Base Cases: $T(1,d) = d$ and $T(n,1) = n$.

Induction Step:

$$T(n,d) \leqslant T(n-1,d) + d + \frac{d}{n}(n \cdot d + T(n-1,d-1))$$

$$\leqslant 2 \cdot d!(n-1)\sum_{i=1}^{d} \frac{i^2}{i!} + d + d^2 + \frac{d}{n} \cdot 2 \cdot (d-1)!(n-1)\sum_{i=1}^{(d-1)} \frac{i^2}{i!}$$

$$= 2 \cdot d!(n-1)\sum_{i=1}^{d} \frac{i^2}{i!} + d + d^2 + 2 \cdot d!\frac{(n-1)}{n}\sum_{i=1}^{d} \frac{i^2}{i!} - 2 \cdot d!\frac{(n-1)}{n}\frac{d^2}{d!}$$

$$\leqslant 2d!n\sum_{i=1}^{d} \frac{i^2}{i!} + d + d^2 - 2 \cdot d!\frac{(n-1)}{n}\frac{d^2}{d!} \leqslant 2 \cdot d!n\sum_{i=1}^{d} \frac{i^2}{i!}$$

□

Lecture May 20th

Recap SeidLP

We have changed notation, now always optimizing downwards, assuming feasible region is in positive orthant $x_i \geqslant 0$.

Important Ideas:

1. *Given set of constraints* H*, optimum is determined by* d *constraints, i.e. if removing one random constraint* h *from* H*, and* $|H|$ *large, it is very likely, that the optimal solution of* $H - \{h\}$ *is the optimum solution of* H *(lucky case). We have bad luck only with probability* $\frac{d}{n}$.

2. *If we have bad luck, still we know that the "unlucky" constraint* h *has to be part of the optimum solution of* H*; so we are not that unlucky after all.*

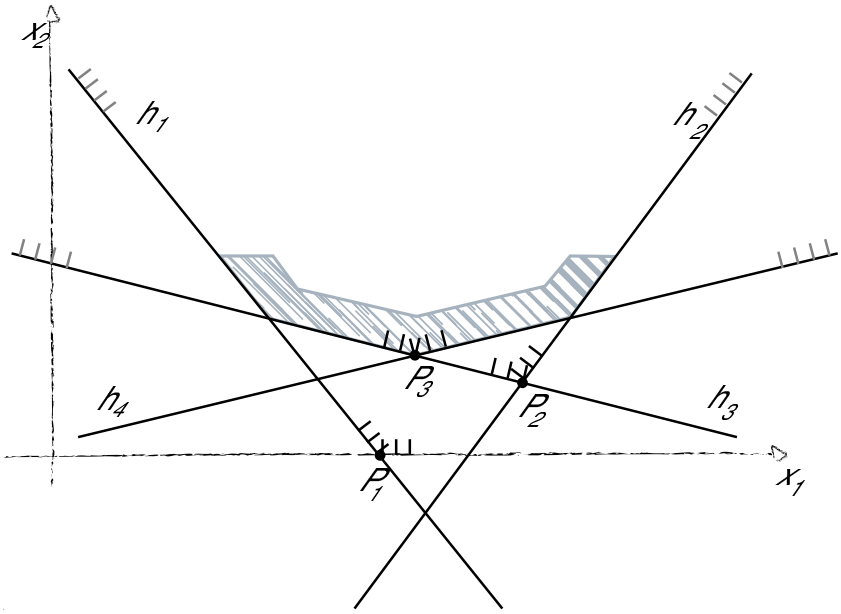

Figure 2: An example for Seidel's algorithm

Example for SeidLP in $\mathbb{R}^2$ We consider the example in Figure **??** and assume that always the constraint with the highest index is chosen "randomly".

SLP(h_1, h_2, h_3, h_4)

(a) SLP(h_1, h_2, h_3)

i. SLP(h_1, h_2)

A. SLP(h_1) $\Rightarrow$ Base Case: return P_1

B. $P_1 \in h_2 \Rightarrow$ return P_1

ii. $P_1 \notin h_3 \Rightarrow$ SLP($\{h_1, h_2\} \downarrow_{h_3} \Rightarrow$ Base Case: return P_2

iii. return P_2

(b) $P_2 \notin h_4 \Rightarrow$ SLP($\{h_1, h_2, h_3\} \downarrow_{h_4}$) $\Rightarrow$ Base Case: return P_3

(c) return P_3

Exercise 6. Take the same set of constraints in Figure **??** but now assume that the order of the constraints is $h_2 < h_3 < h_1 < h_4$ and always the "largest" constraint in this order is chosen "randomly". Follow the SeidLP algorithm step by step.

In the "unlucky" case, Seidel's algorithm uses the fact that the optimum solution to $H - \{h\}$ is violated by h, to conclude that h is in the optimum solution of H. All other information, in particular, all other constraints in the optimal basis of $H - \{h\}$ are thrown away. Maybe many of the constraints in $OPT_{H-\{h\}}$ are also in the optimum solution to H. In fact, this is the basic idea of the algorithm by Matousek, Sharir, Welzl. By making use of this additional information, they obtain even subexponential running time in d.

0.2.4 The LP algorithm by Matousek, Sharir, Welzl

MSW(H, B)

1. if $H = B$ return B
2. choose $h \in H - B$ uniformly at random
3. $B' \leftarrow$ MSW($H - \{h\}, B$)
4. if $v_{B'} \notin h$ then $B' \leftarrow$ MSW(H, pivot(B', h))
5. return B'

MSW((H, B)) computes the optimal basis for the set of constraints H starting from a tentative basis (v-shape) B. Correctness and termination holds for the same reasons as for the v-shape algorithm: the tentative basis improves with each pivot step and only finishes when the current tentative basis is not violated by any constraint. In fact, MSW is also a variant of the v-shape simplex algorithm.

Analysis Similar to the analysis of the SeidLP algorithm, first we are interested in the probability of a "bad luck" choice for the constraint h. As we choose h from $H-B$, this probability is at most $\frac{d}{n-d}$. But in contrast to Seidel's algorithm we can be even a bit more precise. If $d-j$ constraints of the optimal basis of H are in the current tentative basis B, the probability of bad luck is bounded by $\frac{j}{n-d}$.

So it seems somewhat straightforward to measure progress by taking into account the constraints of the current tentative basis B that are also part of the optimal basis of H. This means in particular hoping that if f constraints of the optimal basis are in the basis B, after the return of the first recursive call and the pivot step, there are $> f$ constraints of the optimal basis in pivot(B',h), i.e. progress has been made in that respect. Unfortunately that is not true ! In fact it may well be that in B' there are *less* constraints of the optimal basis than there were in B. So counting the constraints that are also in the optimal basis is not a good measure of progress.

Exercise 7. Show an example in $\mathbb{R}^2$ where no progress in terms of constraints of the optimal basis can be measured when executing the MSW algorithm,i.e. an example where B contains one constraint of the optimal solution but B' none. As in the pivot step one constraint of the optimum basis enters again, we can only show "no progress" in $\mathbb{R}^2$, but no real regression; this is only possible in higher dimension.

Fortunately, if we measure progress by only counting those constraints that will never leave the basis again, we can be sure that progress will be made as we will see in the following.

Definition 0.4. The *hidden dimension* k of a basis B w.r.t. to a set of constraints H is defined as $\mathrm{k}(H,B) = d - |\{h \in H : v_{H-\{h\}} < v_B\}|$.

Intuitively the hidden dimension denotes the number of constraints in the optimal basis that still need to be discovered during the course of the algorithm.

Lemma 0.5. *For any basis $B \subseteq H$ we have $\{h \in H : v_{H-\{h\}} < v_B\} \subseteq B$.*

Proof. Assume one of the h is not in B. Then we have $B \subseteq v_{H-\{h\}}$ and therefore $v_B \leqslant v_{H-\{h\}}$ which is a contradiction. □

Corollary 0.6. *All constraints counted in $\{h \in H : v_{H-\{h\}} < v_B\}$ will never leave the basis again as in the course of the algorithm only higher bases are encountered..*

Let us now consider a call of MSW(H, B) with hidden dimension k, i.e. $d-k$ constraints in B will never leave the basis again during the course of the algorithm. The algorithm first picks some $h \in H - B$ and solves the remaining problem recursively. If h is not in the optimal basis of H, we are done (lucky case). Furthermore none of the constraints in $\{h \in H : v_{H-\{h\}} < v_B\}$ can be picked (as they are in B), so the probability of "bad luck" is at most $\frac{k}{n-d}$.

It remains to argue about the hidden dimension $k(H, \text{pivot}(H \cup \{h\}))$ that will be passed to the second recursive call in case of "bad luck". Our claim is that the hidden dimension $k(H, \text{pivot}(H \cup \{h\}))$ is randomly distributed between $0 \ldots k-1$, which means that basically on the average, the hidden dimension is halved.

We order the d constraints that define the optimum basis of H in the following manner:

$$v_{G-\{h_1\}} \leqslant v_{G-\{h_2\}} \leqslant \ldots \leqslant v_{G-\{h_{d-k}\}} < v_B \leqslant v_{G-\{h_{d-k+1}\}} \leqslant \ldots \leqslant v_{G-\{h_{d-1}\}} \leqslant v_{G-\{h_d\}}$$

This order might not be unique, the parameter k determined by this ordering is unique, though. As we have seen $h_1 \ldots h_{d-k}$ are in B and will never leave the basis again. Hence in our set $H-B$ where h is drawn from, there might be at most k constraints whose choice yield the "unlucky" case. If $k = d$, B has made no measurable progress yet, and all d definining constraints might be in $H-B$, if $k = 0$, B is already the optimal basis.

So assuming that we're unlucky, h is a random constraint amongst $h_{d-k+1} \ldots h_d$. Assume $h = h_i$ was the bad constraint picked, i.e. i is random in $d-k+1, \ldots, d$, then the first recursive call MSW($H - \{h\}, B$) returns with $v_{H-\{h_i\}}$. So in particular, $h_1, \ldots, h_{i-1}$ are now part of the returned B'. The pivot step brings h_i into the basis B' and throws out some other constraint (but none of $h_1, \ldots, h_{i-1}$!). Therefore the hidden dimension of B' after the pivot step is $d-i$ or in other words, the new hidden dimension is random in $0, \ldots k-1$.

The only fact we need to consider before we can write down the recursion is that the hidden dimension is monotone, i.e. for $B \subseteq F \subseteq H$ the hidden dimension of B w.r.t. F does not exceed the hidden dimension of B w.r.t. H, as $h_1, \ldots h_{d-k} \in B$ (and so in F) and $v_{F-\{h\}} \leqslant v_{H-\{h\}}$ for $F \subseteq H$.

Denote by $b(n,k)$ the expected number of basis calls when calling MSW on a set H with n constraints and a tentative basis B of hidden dimension k. According to our discussion above we obtain the following recursion:

$$b(n,k) \leqslant b(n-1,k) + \frac{1}{n-d} \sum_{i=1}^{\min\{k,n-d\}} (1 + b(n,k-i))$$

For the base case we have $b(d,k) = 0$.

Exercise 8. Show that $b(n,k) \leqslant 2^k(n-d)$.

As any tentative basis has hidden dimension of at most d, the MSW algorithm gives an expected running time of $O(2^k n)$, which is already better than the prune and search as well as Seidel's approach. A more elaborate analysis yields a bound of:

$$1 + b(n,k) = e^{O(\sqrt{k \ln(n-d)})}$$

Regarding the number of violation tests, observe that for any computed tentative basis, we check each constraint at most once with this basis, hence the number of basis computations is bounded by $(n-d)b(n,k)$.

In the following we will see another LP algorithm which decreases the number of constraints such which together with the subexponential bound just seen, yields the best subexponential combinatorial algorithm to solve the linear programming problem known so far.

Example for MSW in $\mathbb{R}^2$ We consider the same problem as in Figure **??** assuming that always the constraint with highest index is chosen "randomly".

MSW($\{x_1 \geqslant 0, x_2 \geqslant 0, h_1, h_2, h_3, h_4\}, \{x_1 \geqslant 0, x_2 \geqslant 0\}$)

(a) MSW($\{x_1 \geqslant 0, x_2 \geqslant 0, h_1, h_2, h_3\}, \{x_1 \geqslant 0, x_2 \geqslant 0\}$)

i. MSW($\{x_1 \geqslant 0, x_2 \geqslant 0, h_1, h_2\}, \{x_1 \geqslant 0, x_2 \geqslant 0\}$)

A. MSW($\{x_1 \geqslant 0, x_2 \geqslant 0, h_1\}, \{x_1 \geqslant 0, x_2 \geqslant 0\}$)

- MSW($\{x_1 \geqslant 0, x_2 \geqslant 0\}, \{x_1 \geqslant 0, x_2 \geqslant 0\}$)
  $\Rightarrow$ base Case: return $\{x_1 \geqslant 0, x_2 \geqslant 0\}$
- $(0,0) \notin h_1 \Rightarrow$ MSW($\{x_1 \geqslant 0, x_2 \geqslant 0, h_1\}, \{x_1 \geqslant 0, h_1\}$)
  $\Rightarrow$ after unrolling return $\{x_1 \geqslant 0, h_1\}$

B. $\{x_1 \geqslant 0, h_1\} \in h_2 \Rightarrow$ return $\{x_1 \geqslant 0, h_1\}$

ii. $\{x_1 \geqslant 0, h_1\} \notin h_3 \Rightarrow$

iii. ...

Exercise 9. Take the set of constraints in Figure **??** but now assume that the order of the constraints is $h_2 < h_3 < h_1 < h_4$ and always the "largest" constraint in this order is chosen "randomly". Follow the MSW algorithm step by step.

0.2.5 Clarkson's Algorithms for Linear Programming

The basic idea of Clarkson's algorithms is to reduce the number of constraints that have to be considered by a random sampling procedure. When the problem size has been reduced far enough, the MSW algorithm is used to solve the "small" instances.

Clarkson 1

We start with his first algorithm which repeatedly takes samples of size $d\sqrt{n}$ until the optimum has been found.

CL1(H)

1. if $|H| \leqslant 9d^2$ return CL2(H)
2. $r \leftarrow d\sqrt{n}$; $G \leftarrow \emptyset$
3. repeat
   - choose random $R \in \binom{H}{r}$
   - $v \leftarrow$ CL2($G \cup R$)
   - $V \leftarrow \{h \in H : v \notin h\}$
   - if $|V| \leqslant 2\sqrt{n}$ $G \leftarrow G \cup V$

   until $V = \emptyset$
4. return v

Clearly this procedure computes the optimum solution provided the subroutine CL2(H) works as intended. But why is this procedure efficient ? First, as we will show, the expected size of $|V|$ is $\sqrt{n}$ and hence it takes only two rounds in expectation until some progress is made. Furthermore, in each round at least one constraint of the optimal basis B_{opt} of H is added to G . So the expected number of rounds of the repeat-loop is bounded by $2d$.

Exercise 10. Why is there at least one constraint of the optimal basis added to G in each round ?

Solution: Otherwise $v_H = v_{B_{opt}} \leqslant v_{B_{opt} \cup G \cup R} = v_{G \cup R} \leqslant v_H$.

Corollary 0.7. *For $n = |H| > 9d^2$, CL1 computes v_H with an expected number of $O(d^2 n)$ arithmetic operations, and an expected number of at most $2d$ calls to CL2 with at most $3d\sqrt{n}$ constraints.*

In the following we prove the required Lemma on the expected size of V of violated constraints. It will be formulated more general for later use.

Lemma 0.8. *Let G be a set of constraints in d-space, H a multiset of n constraints in d-space and $1 \leqslant r \leqslant n$. Then for random $R \in \binom{H}{r}$, the expected size of $V_R = \{h \in H | v_{G \cup R}$ violates $h\}$ is bounded by $d\frac{n-r}{r+1}$.*

Proof. Consider the characteristic function $\chi_G(R,h)$, which is 1 if $v_{G\cup R}$ violates h, 0 otherwise. Then

$$\binom{n}{r} E(|V_R|) = \sum_{R \in \binom{H}{r}} \sum_{h \in H-R} \chi_G(R,h)$$

$$= \sum_{Q \in \binom{H}{r+1}} \sum_{h \in Q} \chi_G(Q - \{h\}, h) \leqslant \sum_{Q \in \binom{H}{r+1}} d = d\binom{n}{r+1}$$

$E(|V_R|) \leqslant d\frac{n-r}{r+1}$ follows immediately. □

Exercise 11. *Alternative Proof of the Sampling Lemma (restate lemma in exercise):* Consider the following bipartite graph (A,B), where there is a node $a \in A$ for each r-subset R_a of H and a node $b \in B$ for each $(r+1)$-subset R_b in H. There is an edge (a,b) if and only if, $R_b = R_a \cup h$ and h violates $v_{G\cup R_a}$. **Task 1:** What is the meaning of the degree of a node $a \in A$ in this bipartite graph ? What is the meaning of the average degree of a node $a \in A$ in this bipartite graph ? To determine the average degree of this graph, we count the total number of edges by looking at the nodes in B. **Task 2:** What is the degree of a node $b \in B$? **Task 3:** Use the results of Task 1 and Task 2 to make a statement about the average degree of a vertex $a \in A$. Why does this prove the sampling Lemma ?

In the algorithm above, we apply this Lemma with $r = d\sqrt{n}$ and therefore get $E(|V_R|) < \sqrt{n}$.

Clarkson 2

This algorithm proceeds very similar to the first one. It chooses samples of size $6d^2$ and computes v_R by the MSW algorithm. The expected number of violators V_R is bounded by $\frac{1}{6d}n$ according to the above Lemma.

Exercise 12. Proof that in Clarkson's second algorithm CL2, the expected number of violators is bounded by $\frac{1}{6d}n$ by applying the sampling Lemma. **Hint:** Consider H as a multiset of n elements where a random sample of size $6d^2$ is taken from.

Instead of forcing these violators for the next iterations (as before) we just increase their probability to be chosen by assigning a multiplicity μ_h to each violator h. We start with $\mu_h = 1$ for all $h \in H$ and double it each time h is a violator. The analysis shows that for the constraints of the optimal basis, their multiplicities increase so rapidly that after a logarithmic number of rounds they are chosen with high probability. In the following we view H as a multiset, each h with multiplicity μ_h. $\mu(H) = \sum_{h \in H} \mu_h$. The random sample $R \in \binom{H}{r}$ is also a multiset.

CL2(H)

1. if $|H| \leqslant 6d^2$ return MSW(H)
2. $r \leftarrow 6d^2$
3. repeat
   - choose random $R \in \binom{H}{r}$
   - $v \leftarrow$ MSW(R)
   - $V \leftarrow \{h \in H : v \notin h\}$
   - if $\mu|V| \leqslant \frac{1}{3d}\mu(H)$ then $\forall h \in V : \mu_h \leftarrow 2\mu_h$

   until $V = \emptyset$
4. return v

From the above Lemma and Markov's inequality it follows that the expected number of attempts to get a small enough V is at most 2. We will now bound the number of successful iterations where H gets reweighted.

Lemma 0.9. *Let k be some positive integer. After kd successful iterations we have*

$$2^k \leqslant \mu(B) < ne^{k/3}$$

for the optimal basis B of H.

Proof. Every successful iteration adds a weight of at most $\frac{1}{3d}\mu(H)$ to H, therefore we get

$$\mu(B) \leqslant \mu(H) \leqslant n(1+\frac{1}{3d})^{kd} < ne^{k/3}$$

For the lower bound observe that in each round, at least one of the violators is a constraint in B. That means there is a constraint that has been doubled at least k times hence $\mu(B) \geqslant 2^k$. As $2 > e^{1/3}$, for large enough k the lower bound exceeds the upper bound, hence the solution must be found before. □

Lemma 0.10. *For $n = |H| > 6d^2$, CL2 computes v_H with an expected number of $O(d^2 n \log n)$ arithmetic operations, and an expected number of at most $6d \ln n$ calls to MSW with at most $6d^2$ constraints.*

Proof. For $k = 3\ln n$ we get $2^k = n^{3/\log e} > n^2 = ne^{k/3}$. Therefore there are at most $2 \cdot 3d \ln n$ iterations in expectation. Each iteration costs O(dn) arithemtic operations and one call to MSW. □

0.2.6 Plugging together CL1, CL2, and MSW

We got a running time of $O((d^2+nd)e^{O(\sqrt{d \ln n})})$ for MSW. Plugging this into the running time of CL2 (for $= O(6d^2)$), we obtain $O(d^2 n \log n + e^{O(d \ln d)} \log n)$. Plugging this into CL1, we get finally

Theorem 0.11. *The optimal nonnegative vertex v of a linear program with n constraints H in d-space can be computed by a randomized algorithm with an expected number of $O(d^2 n + e^{O(\sqrt{d \ln d})})$ steps.*

0.2.7 Summary

We have seen several algorithms for solving linear programs of the form $\max c^T x$ s.t. $Ax \leqslant b$.

Exercise 13. All of the algorithms assumed that the linear programming formulations are of the form $\max\ c^T x$ s.t. $Ax \leqslant b$, some of them even that $x_i \geqslant 0$. What can we do if we are only able to solve problems of the form $\max\ c^T x$ s.t. $A^T x \leqslant b$, $x_i \geqslant 0$ but our given problem instance looks as follows:

1. $\min\ c^T x$ s.t. $A^T x \leqslant b$, $x_i \geqslant 0$
2. $\max\ c^T x$ s.t. $A^T x \geqslant b$, $x_i \geqslant 0$

3. max $c^T x$ s.t. $A^T x = b$, $x_i \geqslant 0$
4. max $c^T x$ s.t. $A^T x \leqslant b$, $x_i \in \mathbb{R}$

Let us briefly the different algorithms that we used to solve a linear program of the form max $c^T x$ s.t. $Ax \leqslant b$.

Primal Simplex

The main ideas of the primal simplex are:

- current solution is maintained as a set of d lin.indep. constraints and their intersection point
- the current solution is always feasible
- as long as we can improve locally, we do $\Rightarrow$ we improve in each step ! if we can't improve, we are optimal

The primal simplex algorithm starts at a a corner of the feasible region.

V-shape or *Dual* Simplex

For the same set of constraints the Dual or V-Shape simplex follows a different strategy. The core ideas are:

- current solution is maintained as a set of d lin.indep. constraints and their intersection point
- the current solution is always optimal for a subset of constraints $\Leftrightarrow$ current basis is a v-shape $\Leftrightarrow$ c can be represented as as conic combination of the constraints defining the current basis
- as long as there is a violated constraint, we update the current solution $\Rightarrow$ we get worse solution in each step ! if there are no violations, we are optimal

The dual simplex starts with a locally optimal solution (v-shape).

In fact, the structure of the dual simplex algorithm could also be stated as a linear program — the so-called *Dual Program*

$$\min b^T y \text{ s.t. } A^T y = c, y \geqslant 0$$

The constraints $A^T y = c$, $y \geqslant 0$ correspond to the invariant of the dual simplex that we always have a v-shape as current solution, $b^T y$ expresses the height of the current v-shape, since $y = (A_B^T)^{-1} c$ and therefore $b^T y = b^T (A_B^T)^{-1} c = (A_B^{-1} b)^T c = x'^T c$, where A_B is the submatrix corresponding to the constraints that define the current v-shape and x' the intersection point of those constraints. We have seen that minimizing the height of the v-shape yields the same solution as aiming for the highest feasible point, and hence our dual program directly formalizes the invariants and the goal of the dual simplex algorithm.

Then, the primal simplex on this program is the dual simplex for the original program. In general, one can easily derive the dual program from the primal (and vice versa as the dual of the dual is the primal). If the primal is of the form max $c^T x$, the dual is of the form min $b^T y$ and the following relationship between variables and constraints of primal and dual hold:

	Primal: max $c^T x$	Dual: min $b^T y$
i-th Variable in (P) $\Leftrightarrow$ i-th Constraint in (D)	$x_i \geqslant 0$ $x_i \in \mathbb{R}$ $x_i \leqslant 0$	$A_{\cdot i} y \geqslant c_i$ $A_{\cdot i} y = c_i$ $A_{\cdot i} y \leqslant c_i$
j-th Constraint in (P) $\Leftrightarrow$ j-th Variable in (D)	$A_{j\cdot} x \geqslant b_j$ $A_{j\cdot} x = b_j$ $A_{j\cdot} x \leqslant b_j$	$y_j \leqslant 0$ $y_j \in \mathbb{R}$ $y_j \geqslant 0$

Furthermore unboundedness of one of the problems implies infeasibility for its dual.

Exercise 14. We have seen the equivalence for a primal program of the standard form (PS) max $c^T x$ s.t. $Ax \leqslant b$ and its dual program of the standard form (DS) min $b^T y$ s.t. $A^T y = b, y \geqslant 0$ via the primal and dual simplex algorithm.

Show that the dual of (DS) is again (PS). **Hint:** First transform (DS) such that it is of the same form as (PS) and apply the equivalence that we have seen. Transform the outcome such that the original problem (PS) is obtained. Check whether you would have obtained the same result when applying the rules of the table above.

For both, primal and dual simplex, no polynomial time bound is known. An obvious bound is exponential, as no basis is revisited (can be enforced for both primal and dual simplex).

An Example for Duality (put on exercise sheet)

In the following we will give a small example for duality and its "real-world" interpretation. The goal of our problem will be on one hand to minimize cost of healthy nutrition

(primal/consumer's viewpoint) and on the other hand to maximize profit when selling artificial nutrition products (dual/producer's viewpoint).

In our simplified model a human being needs per day at least 11 units of carbohydrates, 7 units of proteins and 5 units of fat to keep a healthy diet. These ingredients can be acquired by eating *meat* (one unit of which contains 1 unit of carbohydrates, 3 units of proteins and 5 units of fat), *tofu* (2:2:0), *bread* (4:1:0) or *cheese* (1:4:2). These food items can be bought in the local supermarket for 7 EUR per unit of meat, 3 EUR/Tofu, 2 EUR/Bread, 4 EUR/Cheese. The goal is to minimize the daily cost of a healthy nutrition. So we can set up the following linear program to solve this problem:

$$\begin{pmatrix} \min & 7x_m & + & 3x_t & + & 2x_b & + & 4x_c & & \\ \text{s.t.} & 1x_m & + & 2x_t & + & 4x_b & + & 1x_c & \geqslant & 11 \\ & 3x_m & + & 2x_t & + & 1x_b & + & 4x_c & \geqslant & 7 \\ & 5x_m & + & 0x_t & + & 0x_b & + & 2x_c & \geqslant & 5 \\ & x_i \geqslant 0 & & & & & & & & \end{pmatrix}$$

On the other hand consider a producer of artificial nutrition products who produces pure carbohydrate/protein/fat pills and wants to determine the maximum price he can ask for such that people who are only aiming for a minimal cost diet still buy his pills instead of "ordinary" food. Let y_C, y_P, y_F be the respective prices per unit of carbohydrates/proteins/fat. As constraints, the producer can assume that the prices should be determined such that for any "natural" food item, like meat, it is cheaper to buy his pills to get the same amount of nutrition. Furthermore he wants to maximize the profit he gets from one customer who wants to keep his healthy diet (and this customer needs 11 units of carbohydrates, etc.) . So he will set up the following linear program:

$$\begin{pmatrix} \max & 11y_C & + & 7y_P & + & 5y_F & & \\ \text{s.t.} & 1y_C & + & 3y_P & + & 5y_F & \leqslant & 7 \\ & 2y_C & + & 2y_P & + & 0y_F & \leqslant & 3 \\ & 4y_C & + & 1y_P & + & 0y_F & \leqslant & 2 \\ & y_j \geqslant 0 & & & & & & \end{pmatrix}$$

which happens to be the dual of the linear progrm for the customer. By strong duality, we know that the maximum profit the producer can achive is equivalent to the most cost-effective diet-plan for the customer.

Prune and Search

The main ideas of the Prune and Search approach are:

- as the optimum solution is determined by d constraints only, most of the n constraints are redundant
- remove a constant fraction of constraints in each round (fraction depends on dimension, though)

The running time of the prune and search approach is linear in the number of constraints but doubly exponential in d.

SeidLP / MSW

The main ideas of the SeidLP/MSW are:

- if $n >> d$ dropping one constraint at random and solving remaining problem most of the time gives optimal solution
- in case of ”bad luck”, we can use some information from the first call to get to optimum
- SeidLP: only use fact that violating constraint has to be in optimal basis
  MSW: same as SeidLP, but also make use of other constraints that are also in optimal basis
- MSW is a specific instance of the dual simplex

The running time of the SeidLP and MSW is linear in the number of constraints. The dependence on d is exponential for SeidLP, but subexponential for MSW when combined with Clarkson's algorithms.

Clarkson's algorithms

The main ideas of the Clarkson's algorithms:

- reduces the size of the linear program by random sampling on the constraint set
- if one takes a random subset of the constraints, the expected number of violators is not too high and dependent on the size of the subset

The running time of Clarkson's algorithms together with MSW is linear in n and subexponential in d; the so far best known bound for simplex-type algorithms.

Give Overview; Primal/Dual Simplex, Prune and Search, SeidLP, MSW, Clarkson, running times, primal/dual LP

0.3 Approximation Algorithms

0.3.1 Cardinality Vertex Cover

A simple Example Consider a number of cities connected by a system of roads which are are to be used by cars which only have enough fuel to travel *two* road segments before requiring a refuel.

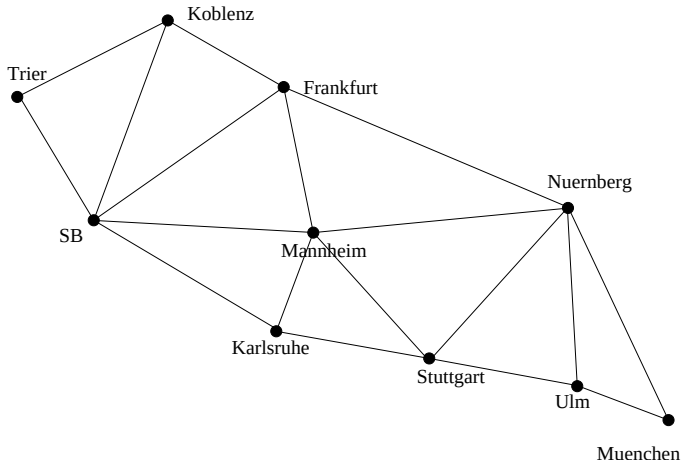

Figure 3: The Gas Station problem

Problem: How to make this network accessible by car, i.e. how to place gas stations such at some cities such that a car driving around never runs out of fuel, irrespectively which tour it takes. the goal is to *minimize* the number of gas stations.

Vertex Cover Problem: Given a graph $G = (V, E)$ with a weight function $c : V \to \mathbb{R}^+$ on the vertices, determine a set $S \subseteq V$ of minimum weight such that for any edge at least one of its endpoints is in S. If $c(v) = 1 \ \forall v \in V$ we call this the *Cardinality Vertex Cover* problem.

The Cardinality Vertex cover expresses our Gas station problem in an abstract way. Unfortunately the CVC is NP-complete, and even worse, one can show that under the assumption that $P \neq NP$, no polynomial time algorithm can approximate the problem better than 1.1666.

Algorithm Proposal

1. $C \leftarrow \emptyset$
2. while $E \neq \emptyset$ do

(a) choose $e = (v, w) \in E$

(b) $C \leftarrow C \cup \{u, v\}$

(c) remove all edges adjacent to u, v

3. od

4. return C

Lemma 0.12. *The above algorithm yields a* 2*-approximation to the cardinality vertex cover problem, i.e.* $|C| \leqslant 2|C_{opt}|$*, where* C_{opt} *denotes the optimal, i.e. smallest set that is a vertex cover for G.*

Proof. The algorithm picks a set of non-adjacent edges – a so-called *matching*. This matching is *maximal*, i.e. no further edge can be added. Consider C_{opt}; clearly C_{opt} must contain at least one vertex for each edge of the matching, i.e. $|C_{\text{opt}}| \geqslant |C|/2$ or in other words $|C| \leqslant C_{\text{opt}} \cdot 2$. □

Can we improve ?

Maybe the algorithm is a lot better than its analysis, and performs much better in reality. Unfortunately that is not true. Consider the complete bipartite graph $B_{n,n}$. A maximal matching M always has size n, C computed by our algorithm therefore $2n$. C_{opt} consists of the vertices on one side only, and therefore has size n, so our approximation guarantee is tight for this example.

In our algorithm we used the size of a maximal matching as a lower bound for the optimal solution; so maybe we can design a different algorithm which computes a cover C which is closer to the size of M and therefore improve the approximation ratio. e.g. $|C| = \frac{3}{2}|M|$; unfortunately this does not work either.

Consider the complete graph $K_{n,n}$, n odd. A maximal matching can have size at most $\frac{n-1}{2}$, an optimal cover has size $n-1$, though. So using matching as a lower bound we will never be able to show a better bound than 2.

Exercise 15. Consider the following factor 2 approximation algorithm for the cardinality vertex cover problem: Find a depth first search tree in the given graph G and output the set, say S of of all the non-leaf vertices of this tree. Show that S is indeed a vertex cover for G and $|S| \leqslant 2|C_{\text{opt}}|$. **Hint:** Show that G has a matching of size $|S|$.

0.3.2 Some Notation

Notion	In our example
Optimization Problem Π	"min. vertex cover in graphs"
Instances ξ_Π	"all graphs"
feasible solutions $S(I)$ of an instance $I \in \xi_\Pi$	"for a graph ($\approx$ instance) all vertex covers"
objective function $\phi : S(I) \to \mathbb{R}$	"$\lvert S(I)\rvert = \lvert V\rvert$ cardinality of vertex cover"
optimization criterion: MIN/MAX	"MIN"

Goal $\forall I \in \xi_\Pi$, find $s \in S(I)$ called $\mathrm{OPT}(I)$ s.t.

$$\text{(MAX) } \phi(s) \geqslant \phi(s')\forall s' \in S(I)$$

$$\text{(MIN) } \phi(s) \leqslant \phi(s')\forall s' \in S(I)$$

Typically this is hard for most problems (in particular real-world!); so we are also happy with δ-approximations; i.e. an Algorithm such that $\forall I \in \xi_\Pi$, we get an $S(I)$ with:

$$\text{(MAX) } \phi(s) \geqslant \delta \cdot \phi(s')\forall s' \in S(I), \delta \in [0,\ldots,1]$$

$$\text{(MIN) } \phi(s) \leqslant \delta \cdot \phi(s')\forall s' \in S(I), \delta \geqslant 1$$

We call such an algorithm δ-approximation algorithm.

0.3.3 Precedence Constraint Scheduling

Example Assume you want to assemble a car from its numerous parts. You know where each part belongs to, but still you cannot just start assembling as some parts have to be put together before others. These are *dependencies* like for example you do not want to mount the engine into the car before the engine itself has been assembled from its sub-parts. Furthermore you have *m* engineers whose job is it to put together the whole car, so at any given point in time, at most *m* tasks can be worked on simultaneously. For reasons of simplicity we assume that each task takes one timeunit. The goal is to complete the assembly of the car as soon as possible.

More formally the problem can be expressed as a directed acyclic graph $G = (V,E)$ where each node $v \in V$ corresponds to a task, and there is an directed edge (v,w) if task v has to be completed before task w. We are looking for a function (schedule) $S : V \to \mathbb{N}$, which assigns each task a time at which this task is worked on, but under the condition that:

- $\forall k : |\{v : S(v) = k\}| \leqslant m$ ("at no point in time no more than m jobs are currently being worked on")
- $\forall (v,w) \in E$, $S(v) < S(w)$ ("precedence constraint is fulfilled")

The goal is to minimize $\max\{S(v) : v \in V\}$, i.e. the last point in time when tasks are being worked on.

For simplicity we assume that there is a unique source node s which has precedence before all other tasks that have no incoming edge.

An Algorithm

1. $\forall v \in V$ compute the longest path from s to v, call this distance the label of v
2. let d_i be the number of vertices labelled with i; use the first $\lceil \frac{d_1}{m} \rceil$ time units for vertices at level 1, the next $\lceil \frac{d_2}{m} \rceil$ time units for level 2, and so on ..

Lemma 0.13. *Let L be the length of the constructed schedule of our algorithm, i.e. $L = \max\{S(v) : v \in V\}$, then we have $L \leqslant 2 \cdot L_{opt}$.*

Proof. Let t be the max. level of a node, then we have

1. $L_{\text{opt}} \geqslant t$
2. $L_{\text{opt}} \geqslant \frac{n}{m} = \frac{d_1 + d_2 + \cdots + d_t}{m}$
3. $L \leqslant \frac{d_1 + d_2 + \cdots + d_t}{m} + t$

And therefore clearly $L \leqslant 2 \cdot L_{\text{opt}}$. □

So this simple algorithm yields a 2-approximation. One can show that under the assumption that $P \neq NP$ it is not possible to get a polynomial time algorithm that yields a better approximation guarantee than $4/3$.

The $4/3$ Lower Bound We will use the following NP-hard problem to show that no better approximation guarantee than $4/3$ for the Precedence Constraint Scheduling problem can be obtained unless $P = NP$.

***k*-Clique:** Given a graph $G(V,E)$ and an integer k, decide whether there exists a clique of size k in G.

The idea will be as follows: For a given Instance $(G(V,E),k)$ of the k-Clique problem we construct an instance of the precedence constraint scheduling problem which

- can always be solved in 4 time units
- can be solved in 3 time units if and only if G contains a k-clique

Therefore, if a (polynomial-time) approximation algorithm for the PCS problem had an approximation guarantee better than $4/3$, it would have to report the 3 time units schedule for graphs containing a k-clique $\Rightarrow$ we would have found a polynomial-time algorithm for a NP-hard problem.

The set of tasks will consist of $V \cup E \cup F_1 \cup F_2 \cup F_3$, where F_i are filler tasts for step i. The precedence constraints are:

- $v \rightarrow$ e if v is one of the endpoints of e (there will be only v's in the first round)
- all tasks in F_i before all tasks in F_{i+1}, $i \in \{1,2\}$

Furthermore we choose the number of machines m and $|F_i$ as follows:

1. $m = k + |F_1|$ ($\Rightarrow$ in first round, only k 'real' things from V can be processed)
2. $m = |F_2| + (n-k) + \frac{k(k-1)}{2}$
3. $m = |E| - \frac{k(k-1)}{2} + |F_3|$
4. $|F_i| \geqslant 1$

Lemma 0.14. *The constructed precedence constraint scheduling problem can always be processed in* 4 *rounds.*

Proof. First round: F_1 and k nodes of V. Second round: F_2 and $(n-k)$ nodes of V. Third round: F_3 and some of the edges. Fourth round: remaining edges. □

Lemma 0.15. *If there exists a clique of size k in G, we can process the problem in* 3 *rounds.*

Proof. First round: The k nodes of the clique and F_1. Second round: The edges of the clique and the remaining nodes and F_2. Third round: The remaining edges and F_3. □

Lemma 0.16. *If we can process the problem in* 3 *rounds, there exists a clique of size k in G.*

Proof. A 3-round solution has to process m things in each round. In the first round, only vertices and F_1 can be processed. Le V' be the set of processed vertices in round 1, $|V'| = k$. In round 2 one has to process F_2, the remaining nodes and $m - |F_2| + (n-k) = \frac{k(k-1)}{2}$ edges. But that many edges are only allowed to be processed, if they are all between the vertices $|V'|$, i.e. V' has to be a clique. □

0.3.4 Independent/Stable Set

Example Consider the following problem. Given a collection of circles in the plane – possibly intersecting – select a subset S of these circles such that none of the circles in S intersect. Choose S as large as possible.

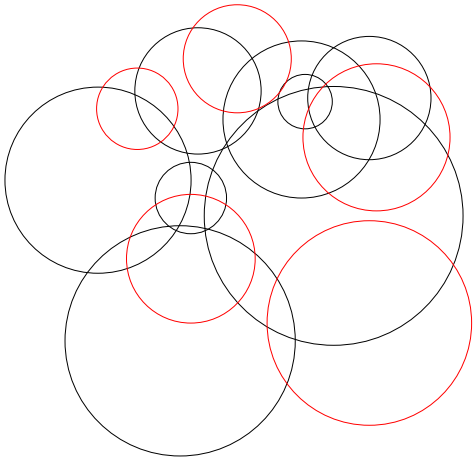

Figure 4: A collection of circles, some of them selected (red).

We can formalize this problem as a graph problem by associating a vertex with each circle and putting an edge (v,w) if the corresponding circles intersect. The goal is then to find a large subset of vertices such that none of them is connected by an edge.

Independent/Stable Set Problem: Given a graph $G = (V,E)$, determine a set $S \subseteq V$ of maximum cardinality such that $\forall v,w \in S$, $(v,w) \notin E$.

Unfortunately this problem is not only NP-hard but basically unapproximable, so no polynomial time approximation algorithm with $\delta = n^{1-\varepsilon}$ for any $\varepsilon > 0$ is known unless some complexity classes coincide which are believed not to. The problem is equivalent to the maximum clique problem on the complementary graph (which might be better known as a hard problem).

Still this does not mean that for practical purposes there is no hope to get good solutions for independent set problems. If the underlying graph has some special properties,

one can even show some theoretical guarantees.

The most obvious idea for a heuristic for the stable set problem in a graph — especially if it is a graph of bounded maximum degree – is to use the following greedy approach:

1. $S \leftarrow \emptyset$
2. $v \leftarrow$ some vertex in G with minimum degree
3. $S \cup \{v\}$; $G \leftarrow G - \{v\} - \bigcup_{(w,v)\in E} w$
4. if $G \neq \emptyset$ goto 2.

The algorithm always takes the vertex with smallest degree and adds it to the stable set S determined so far, then removes this vertex and all its neighbours and repeats. Clearly, the output of this algorithm is a stable set in G. Furthermore if there is an upper bound on the degree of the vertices in G, we obtain:

Lemma 0.17. *Let $G = (V,E)$ be a graph with maximum degree Δ. Then the greedy algorithm computes a independent set of size at least $1/\Delta \cdot OPT$.*

Proof. Exercise ! □

For example, it sometimes helps to consider the ILP formulation of the problem:

$$\begin{aligned} \max \quad & \sum_{v\in V} y_v && (0.1)\\ \{u,v\} \in E: \quad & y_u + y_v && \leqslant 1 \\ u \in V: \quad & y_u && \in \{0,1\}. \end{aligned}$$

Of course, solving this ILP is still NP-hard for most graph classes, but the linear relaxation can be solved in polynomial time and possibly used to obtain good solutions in practice.

Lecture July, 3rd
Recap: Cardinality Vertex Cover, Precedence Constraint Scheduling, Indep./Stable Set, APX-Algs.; non-approx proof for PCS

0.3.5 The Metric Travelling Salesperson Problem

Given a complete undirected graph $G = (V,E)$, $V = \{v_1, v_2, \ldots, v_n\}$ with costs c_{ij} for each edge $\{i,j\}$ and $c_{ij} + c_{jk} \geqslant c_{ik}$, determine a closed path $\pi = v_{i_0} v_{i_1} \ldots v_{i_{n-1}} v_{i_0}$ such that $\bigcup v_{i_j} = V$ and $\sum_{j=0\ldots n-1} c_{i_j i_{(j+1)\%n)}}$ is minimized.

This problem is a classical NP-complete problem, and even worse, unless $P \neq NP$, there is no polynomial-time approximation scheme for this problem.

In the following we will first describe a 2-approximation and then improve this to a 1.5-approximation algorithm.

A 2-approximation

1. Compute a minimum spanning tree in G
2. Double all edges of the tree and construct a closed tour by a depth-first search
3. Eliminate reoccurrences of nodes

Elimination of reoccurrences of nodes Let $v_{i_1}, v_{i_2}, \ldots v_{i_j}, \ldots, v_{i_k}, \ldots$ be the current tour, k minimal with $v_{i_j} = v_{i_k}$, $j < k$ (the first repeated node). Then repeat the subsequence $v_{i_{k-1}} v_{i_k} v_{i_{k+1}}$ by $v_{i_{k-1}} v_{i_{k+1}}$. We still have a tour through all nodes and the cost cannot increase since the triangle inequality holds.

Clearly the minimum spanning tree is a lower bound on the cost of an optimal tour. The first initial tour constructed therefore has at most twice the weight of the optimal tour and is only decreased during the elimination steps $\Rightarrow$ 2-Approximation.

Lecture July, 8rd
Recap: 2-APX for TSP

A 1.5-approximation

One can improve the above approach as follows:

1. Compute a minimum spanning tree T in G
2. Let V_{odd} be the set of vertices with odd degree ($|V_{\text{odd}}|$ is even!)
3. Construct a matching M of the vertices in V_{odd} of minimal weight, add them to the tree edges
4. Construct a closed path on the tree and matching edges
5. Eliminate reoccurrences of nodes

Lemma 0.18. *The weight of M is at most half the weight of the optimal TSP tour in G.*

Proof. Consider the optimal TSP tour in G and cut it at the nodes in V_{odd} into several sequences. As $|V_{\text{odd}}|$ even, we have an even number of sequences. Consider the set A consisting of the 1st,3rd, 5th, ... sequence and the set B consisting of the 2nd, 4th, 6th, ... sequence. Either A or B must have weight at most half the weight of the optimal TSP tour in G. Assume this is A. Then replace each chosen sequence by one direct edge connecting its endpoints. This edge is not longer than the sequence due to the triangle inequality. The set of all these edges form a matching for the vertices in V_{odd} of weight no more than half the weight of the optimal TSP tour in G. □

Other Variations of the problem

- G is a geometric graph, i.e. the vertices correspond to a set of points in the plane and the edge weights to their respective distances. For this case, a $(1+\varepsilon)$ approximation scheme exists.
- G is a directed graph, possibly with asymmetric edge costs, but satisfying the triangle inequality. For this case, a $0.999 \log n$ approximation exists.
- For arbitrary graphs, no approximation is possible unless $P \neq NP$.
- One can also consider the *Maximization* version of the problem.

Exercise 16. Show that the TSP problem on graphs with symmetric edge costs but not satisfying the triangle inqequality cannot be approximated within any factor. **Hint:** Assume there is a polynomial-time approximation algorithm which computes a solution within a factor of α of the optimal solution (where α does not need to be a constant). Use this approximation algorithm to solve the Hamiltonian Cycle problem.

0.3.6 Scheduling of independent tasks

Given a set of jobs with associated processing times $w_1, w_2, \ldots w_n$ and m machines, assign each job to a machine such that the makespan is minimized, i.e. determine $S : \{1, \ldots n\} \to \{1, \ldots m\}$ with $\max_k \sum_{i;S(i)=k} w_i$ is minimal.

Consider the following simple approximation algorithm:

1. Sort the jobs in decreasing size, i.e. $w_1 \geqslant w_2 \geqslant \ldots \geqslant w_n$
2. Schedule the jobs one after another, assign w_i to the machine which is least used at that moment.

Let L be the length of the computed schedule, L_{opt} the length of the optimal schedule. We claim:

Lemma 0.19. $L \leqslant (\frac{4}{3} - \frac{1}{3m})L_{\text{opt}}$

Proof. We will use induction over n, the number of jobs. Clearly, the claim holds for $n = 1$.

$n \to n+1$**:** Look at the schedule and in particular the job $n+1$. If it finishes before time L, consider the problem of the first n tasks. Here we have

$$L(w_1,\ldots,w_n) \leqslant (\frac{4}{3} - \frac{1}{3m})L_{\text{opt}}(w_1,\ldots,w_n) \leqslant (\frac{4}{3} - \frac{1}{3m})L_{\text{opt}}(w_1,\ldots,w_n,w_{n+1})$$

Now assume that job $n+1$ is completed at time L, i.e. job $n+1$ determines (maybe with others) the makespan of the computed schedule. Observe that up to time $L - w_{n-1}$, all machines are fully loaded, therefore $w_1 + w_2 + \ldots w_n \geqslant m \cdot (L - w_{n+1})$. Furthermore we have $L_{\text{opt}} \geqslant \frac{w_1+w_2+\ldots w_n+w_{n+1}}{m}$. Hence

$$L \leqslant \frac{w_1 + w_2 + \ldots w_n}{m} + w_{\text{n+1}} = \frac{w_1 + w_2 + \ldots w_n}{m} + (1 + \frac{1}{m} - \frac{1}{m})w_{\text{n+1}} \leqslant L_{\text{opt}} + \frac{m-1}{m}w_{\text{n+1}}$$

Now consider two cases:

Case 1 – $\frac{m-1}{m}w_{n+1} \leqslant (\frac{1}{3} - \frac{1}{3m})L_{\text{opt}}$**:** ok

Case 2 – $\frac{m-1}{m}w_{n+1} > (\frac{1}{3} - \frac{1}{3m})L_{\text{opt}}$**:** This means $w_{n+1} > \frac{L_{\text{opt}}}{3} \Leftrightarrow L_{\text{opt}<3w_{n+1}}$, so the optimal schedule processes $\leqslant 2$ jobs on each machine. Assume w.l.o.g. that $n+1 = 2m$ (otherwise add jobs of length 0). Our algorithm pairs job i with job w_{2m-i+1} on machine i, $1 \leqslant i \leqslant m$. We claim that this schedule is optimal. Assume otherwise and let i_0 be the machine that maximizes $w_i + w_{2m-i+1}$. The optimal schedule has to process 2 jobs on each machine, so let us ask with which other job does the optimal schedule pair jobs $1,2,\ldots,i_0$? Clearly it cannot pair them amongst each other; in fact if the optimal schedule is shorter, it has to schedule all these i_0 jobs with some jobs after job $2m - i_0 + 1$. But there are only $i_0 - 1$ of them. Hence one of the first i_0 jobs has to be paired with a longer job and hence yields a even larger makespan.

□

So we have essentially a $4/3$-approximation algorithm. But maybe we can do even better ...

An approximation Scheme

Basic idea: schedule $w_1, \ldots, w_k$ optimally using a brute-force approach ($O(m^k)$), then complete with the above approximation algorithm. We claim the following lemma:

Lemma 0.20. $L \leqslant (1+\frac{m}{k})L_{\text{opt}}$

Proof. Let L' be the length of the optimal schedule for $w_1, \ldots, w_k$. If $L' = L$ we are done, otherwise let $j > k$ be a job that determines L. Then we have $L = L - w_j + w_j \leqslant L_{\text{opt}} + w_{k+1} \leqslant (1+\frac{m}{k})L_{\text{opt}}$. The last inequality holds because $w_j \leqslant w_{k+1}$ and at time $L - w_j$ all machines are used and hence also at time $L - w_{k+1}$. So we get $L_{\text{opt}} \geqslant \frac{w_1+\ldots w_{k+1}}{m} \geqslant \frac{(k+1)w_{k+1}}{m}$. □

Using this scheme, we can get as good an approximation as we want. For a given $\varepsilon > 0$, we choose k such that $\frac{m}{k} \leqslant \varepsilon$. Hence for $k = \lceil m/\varepsilon \rceil$ we obtain an algorithm with $L \leqslant (1+\varepsilon)L_{\text{opt}}$ and a running time of $O(m^{\frac{m}{\varepsilon}} + n \log m)$. This is called an *approximation scheme*.

0.3.7 The Knapsack problem

Imagine you are a thief and on a burglary trip. You have entered a house with a lot of valuable things, but unfortunately more than you can carry in your knapsack. So there are n items each of which has a value c_i and a weight w_i. Clearly your goal is to steal a set $I \subseteq \{1, \ldots, n\}$ of things which maximizes your profit, i.e. $\max P(I) = \max \sum_{i \in I} c_i$ under the constraint that all the items fit in your knapsack of size k, i.e. $w(I) = \sum_{i \in I} w_i \leqslant k$. We assume $c_i, w_i, k \in \mathbb{R}$.

We will first describe an algorithm which solves the problem exactly. Conceptually we fill a table of size $n \times c_{\text{opt}}$, where at position $w(l,c)$ we store the minimal weight (and the content) of a knapsack of cost c using only a subset of items from $\{1, \ldots, l\}$. So clearly $w(1,c) = w_1$ for $c = c1$, $w(1,0) = 0$ and $w(1,c) = \infty$ otherwise. The value of $w(l,c)$ can then be determined as $w(l,c) = \min\{w(l-1,c), w(l-1,c-c_l) + w_l\}$.

After filling the table it remains to find the maximal c with $w(n,c) \leqslant k$. This is in fact the optimal solution. The running time of this algorithm is the size of the table, $n \cdot c_{\text{opt}}$. Note that this is only a *pseudo-polynomial* algorithm as it depends on the size of the input/the optimal solution.

This approach is called *dynamic programming* and is a very useful paradigm that can be applied to many other optimization problems (at least as a subroutine).

Exercise 17. In the lecture you have seen a dynamic programming approach for the knapsack problem which one by one computed the 'lightest' subset of items $\{1,\ldots,l\}$ whose value was exactly c. Another approach would be to compute one by one the most valuable subset of items $\{1,\ldots,l\}$ whose weight is exactly s. Describe such a scheme in detail and analyse its running time.

An approximation Scheme

In the follwing we will describe a approximation scheme that will run in polynomial time and can achieve an arbitrary good approximation to the optimum solution.

1. Choose a scaling factor $S \in \mathbb{N}$
2. Determine the optimal solution I' for the modified problem with profits $\lfloor \frac{c_i}{S} \rfloor$
3. Return I'

We now want to bound the obtained solution in terms of the optimal solution set I for the original problem.

$$L = \sum_{i\in I'} c_i = S \cdot \sum_{i\in I'} (\lfloor \frac{c_i}{S} \rfloor + \varepsilon_i) = S \cdot \sum_{i\in I'} \lfloor \frac{c_i}{S} \rfloor + S \cdot \sum_{i\in I'} \varepsilon_i$$

$$\geqslant S \cdot \sum_{i\in I} \lfloor \frac{c_i}{S} \rfloor \geqslant S \cdot \sum_{i\in I} (\frac{c_i}{S} - \varepsilon_i) = S \cdot \sum_{i\in I} \frac{c_i}{S} - S \cdot \sum_{i\in I} \varepsilon_i \geqslant L_{\text{opt}} - S \cdot n$$

So if we choose S such that $\frac{S}{L_{\text{opt}}} \leqslant \varepsilon$, i.e. $S = \lfloor \frac{\varepsilon L_{\text{opt}}}{n} \rfloor$, we get $L \geqslant (1-\varepsilon) L_{\text{opt}}$ with a running time of $O(n \cdot c_{\text{opt}_{\text{skal}}}) = O(n \cdot \frac{c_{\text{opt}}}{S}) = O(n^2/\varepsilon)$.

This is called a *full approximation scheme* as the dependency on $1/\varepsilon$ is *polynomial* (compare to the approximation scheme for the scheduling of independent tasks problem; there, $1/\varepsilon$ appeared in the exponent !). Still, there is a slight problem with the choice of S, as L_{opt} is not known in advance, of course.

Exercise 18. Consider the following greedy algorithm for the knapsack problem:

1. Sort the items according to their profit-to-weight ratio c_i/w_i.
2. Pick the items one-by-one starting with the highest profit-to-weight ratio, until the knapsack is full

Can you prove a constant approximation guarantee for this algorithm ? If so, what is it ? If no, why not ?

Exercise 19. Consider the following modified greedy algorithm for the knapsack problem:

1. Sort the items according to their profit-to-weight ratio c_i/w_i.
2. Pick the items one-by-one starting with the highest profit-to-weight ratio, until the knapsack is full
3. Return the current knapsack or the most profitable single element that fits into the knapsack, whichever is more profitable.

Prove that this algorithm is a 0.5-approximation. **Hint:** Consider the fractional version of the knapsack problem where items might be selected fractionally.

The Nemhauser/Ullman algorithm (Presentation due to Beier/Vöcking)

In the following we present a different method to solve the knapsack problem exactly due to Nemhauser/Ullman. Their algorithm is based on the notion of *pareto-optimal* solutions. A solution $I' \subseteq \{1,\ldots,n\}$ is called *pareto-optimal* or a *dominating set*, if $\nexists I'' \subseteq \{1,\ldots,n\}$ with $w(I'') \leqslant w(I')$ and $p(I'') > p(I')$ ("there is no more profitable solution of no more weight"). A set I' that is not a dominating set cannot be optimal for the knapsack problem, regardless of the specified knapsack capacity.

The following algorithm computes the sequence of dominating sets. For $i \in \{1,\ldots,n\}$, let $S(i)$ denote the sequence of dominating subsets of $\{1,\ldots,n\}$ in increasing order of their weights. Given $S(i)$, $S(i+1)$ can be computed in the following way. First duplicate all subsets in $S(i)$ and then add item $i+1$ to each of the duplicated sets. So we obtain two ordered sequences of sets. We merge the two sequences (as in merge sort) and remove dominated subsets on the way. The result is the ordered sequence $S(i+1)$ of dominating sets of $\{1,\ldots,i+1\}$.

For the purpose of illustration and a better understanding, let us take a different view on this algorithm. For $i \in \{1,\ldots,n\}$, let $f_i : \mathbb{R} \to \mathbb{R}$ be a mapping from weights to profits such that $f_i(t)$ is the maximum profit over all subsets of $\{1,\ldots,i\}$ with weight at most t. Clearly f_i is a non-decreasing step function changing at weights that correspond to dominating subsets. In particular, the number of steps in f_i equals the number of dominating

sets over the items in $\{1,\ldots,i\}$. In the algorithm by Nemhauser/Ullman, f_{i+1} is then determined by the upper envelop of f_i and f_i shifted by the vector (w_i, p_i).

So what is the running time of the Nemhauser/Ullman algorithm ? $S(i+1)$ can be computed in time linear in $|S(i+1)|$, i.e. linear in the number of dominating subsets over the items $1,\ldots,i$. As the optimal knapsack is one of the subsets in the list $S(n)$, generating $S(n)$ solves the knapsack problem exactly.

Theorem 0.21. *For every $i \in \{1,\ldots,n\}$ let $q(i)$ denote an upper bound on the number of dominating sets over the items in $1,\ldots,i$. Then the Nemhauser/Ullman algorithm computes an optimal knapsack filling in time $O(\sum_{i=1}^{i=n-1} q(i))$. If we have $q(i+1) \geqslant q(i)$ this is $O(n \cdot q(n))$.*

In the worst-case, the number of dominating sets is $\Omega(2^n)$ (the problem is NP-hard after all). But one has observed that for random instances of the problem, the number of dominating sets is rather low and therefore the Nemhauser/Ullman algorithm runs very fast. Only recently Beier/Vöcking could actually prove this behaviour theoretically. They show that for arbitrarily chosen weights (by an adversary) and profits drawn uniformly at random from $[0,1]$, the expected length of S_i is $O(n^3)$ and hence the Nemhauser/Ullman algorithm runs in $O(n^4)$. In fact they prove an even stronger result for arbitrary probability distributions.

Theorem 0.22. *For arbitrary weights and profits chosen uniformly at random in $[0,1]$, the knapsack problem can be solved exactly in expected time $O(n^4)$.*

Details of analysis left out

Lecture June, 15th: Recap: Knapsack, in particular Nemhauser/Ullman; state result by Beier/Vöecking

0.3.8 Set Cover

Consider the following problem. Given a universe U of n elements, a collection of subsets of U, $\mathcal{S} = \{S_1,\ldots,S_k\}$, and a cost function $c : \mathcal{S} \to \mathbb{Q}^+$, find a minimum cost subcollection of $\mathcal{S}$ that covers all elements of U.

In the following we will see two algorithms for this problem which both rely (at least in the analysis) on the integer linear programming (ILP) formulation of the problem and its dual. How could we formulate the set cover problem as an ILP ?

$$\begin{aligned} \min \quad & \sum_{S\in\mathcal{S}} x_S \cdot c_S && \\ u \in U: \quad & \sum_{u\in S} x_S && \geqslant 1 \\ S \in \mathcal{S}: \quad & x_S && \in \{0,1\}. \end{aligned} \tag{0.2}$$

Solving ILPs in general is quite hard, so one often considers the LP relaxation, where the integrality constraints are dropped, here we get the following *fractional covering LP*:

$$\begin{aligned} \min \quad & \sum_{S\in\mathcal{S}} x_S \cdot c_S && \\ u \in U: \quad & \sum_{u\in S} x_S && \geqslant 1 \\ S \in \mathcal{S}: \quad & x_S && \geqslant 0 \end{aligned} \tag{0.3}$$

Of course, this relaxation might allow better (i.e. cheaper solutions) as the integral problem, so this only yields a *lower* bound in terms of the cost of the solution. Consider the following example: $U = \{e, f, g\}$ and $S_1 = \{e, f\}$, $S_2 = \{f, g\}$, $S_3 = \{e, g\}$, each of unit cost. An integral cover must pick two of the sets for a cost of 2. But picking each set by an extent of $1/2$ yields a feasible fractional cover of cost $3/2$.

We have learned that every LP has its dual which in this case looks as follows:

$$\begin{aligned} \max \quad & \sum_{u\in U} y_u && \\ S \in \mathcal{S}: \quad & \sum_{u\in S} y_u && \leqslant c_S \\ u \in U: \quad & y_u && \geqslant 0 \end{aligned} \tag{0.4}$$

This LP describes a packing problem, where the goal is to pick as many elements from the universe such that none of a certain number of subsets is 'overpacked'. We have seen a very similar LP before for the stable set problem, which is also a special instance of a packing problem. In fact, set cover and packing problem are dual to each other. Furthermore, we have learned $\sum_{u\in U} y_u \leqslant \sum_{S\in\mathcal{S}} x_S c_S$ for all feasible solutions x, y of the two LPs.

A greedy algorithm

The following greedy algorithm computes a cover (not necessarily optimal):

1. $C \leftarrow \emptyset$
2. while $C \neq U$ do
   - find the most cost-effective set in the current iteration, say S
   - let $\alpha = \frac{\text{cost}(S)}{|S-C|}$, i.e. the cost-effectiveness of C
   - pick S ($x_S = 1$) and for each $u \in S - C$, set price$(u) = \alpha$ ($y_u = \frac{\alpha}{H_n}$)
   - $C \leftarrow C \cup S$
3. output the picked sets

Lemma 0.23. *The greedy algorithm achieves an approximation ratio of H_n.*

Proof. We show the approximation ratio by showing that x and y as set by the greedy algorithm form feasible solutions to the set cover and packing LP and $H_n \cdot \sum_{u \in U} y_u = \sum_{S \in \mathcal{S}} x_S c_S$.

Feasibility of x follows from the description of the algorithm. Equally clear is that at all times $H_n \cdot \sum_{u \in U} y_u = \sum_{S \in \mathcal{S}} x_S c_S$ holds. So it remains to prove that for every $S \in \mathcal{S}$, the respective packing constraint in the dual LP is satisfied.

Sort the elements of S according to when they are covered during the course of the algorithm $u_1, \ldots, u_l$. Consider the iteration when our algorithm covers element e_i. At this point, S contains at least $k - i + 1$ uncovered elements. Therefore S could cover e_i at this time at an average cost of at most $c_S/(k-i+1)$. As the algorithm always picks the most cost effective set, we get $y_{e_i} \leqslant \frac{1}{H_n} \frac{c_S}{k-i+1}$. So overall we get

$$\sum_{i=1}^{l} y_{e_i} \leqslant \frac{c_S}{H_n} \cdot (\frac{1}{l} + \frac{1}{l-1} + \cdots + \frac{1}{1}) = \frac{H_l}{H_n} c_S \leqslant c_S$$

□

In the greedy algorithm, the (I)LP formulation was only used to proof a bound on the approximation ratio, but not part of the algorithm. In the following we will see an approach which actually computes the solution to this LP to obtain a set cover solution.

Remark: In the lecture have seen an algorithm for solving LPs in subexponential, but not polynomial time. There are algorithms, though, which can solve linear programming problems in polynomial time (Ellipsoid and interior point methods). So for all algorithms that follow and make use of an LP solution, we assume that it can be computed

in polynomial time. In practice though, the Simplex methods – in spite of the superpolynomial worst-case running-time – perform very well and are often superior to the above guaranteed-polynomial-time methods.

Simple LP rounding for the Set Cover problem

In the greedy algorithm, the LP relaxation was only used for the analysis, but given an optimal solution to the LP relaxation, how could we turn that into a feasible integral solution to the set cover problem, possibly with an approximation guarantee ?

In the following denote by f the maximal number of occurrences of an element $u \in U$ in the sets $S_1, \ldots, S_k$, i.e. $f = \max_{u \in U} |\{S_i : u \in S_i\}|$. We consider the following simple algorithm:

1. Find an optimal solution to the LP relaxation of the set cover problem
2. Pick all sets S for which $x_S \geqslant 1/f$

Lemma 0.24. *The simple $1/f$ LP-rounding algorithm achieves an approximation factor of f.*

Proof. Let $\mathcal{C}$ be the collection of picked sets. Consider an arbitrary element u. Since u is in at most f sets, at least on of the respective sets must be picked to an amount of $\geqslant 1/f$. Hence $\mathcal{C}$ is a valid cover. On the other hand, the value of each x_S is increased by at most a factor of f, so the overall objective function value increases by at most a factor of f. □

Exercise 20. Consider the vertex cover problem that we have seen before. What approximation guarantee does the above algorithm yield for the vertex cover problem ?

Randomized LP rounding for the Set Cover problem

A more sophisticated way to turn the LP solution into a feasible integral solution is to interpret the LP values as probabilities and flip coins accordingly.

Let $x^* = p$ be the optimal solution to the LP relaxation of the set cover problem, OPT_{LP} its objective function value. For each set S we pick it with probability $x^*_S = p_S$ into $\mathcal{C}$. What is the expected cost of $\mathcal{C}$?

$$\mathbf{E}[\mathrm{cost}(\mathcal{C})] = \sum_{S \in \mathcal{S}} p_S \cdot c_S = \mathrm{OPT}_{LP}$$

The collection $\mathcal{C}$ not necessarily covers U, but let us look at the probability by which an element $u \in U$ is covered. Assume u appears in l sets $S_1, \ldots, S_l$. As u is fractionally covered, we have $p_1 + p_2 + \cdots + p_l \geqslant 1$. As the probability for u being covered is minimized if $p_i = 1/l$, we get the following bound:

$$\Pr(a \text{ covered by } \mathcal{C}) \geqslant 1 - (1 - \frac{1}{l})^l \geqslant 1 - \frac{1}{e}$$

The last inequality follows from $(1 + \frac{1}{n})^n < e < (1 + \frac{1}{n})^{n+1}$ since $e < (\frac{n+1}{n})^{n+1} \Leftrightarrow (\frac{n}{n+1})^{n+1} < 1/e \Leftrightarrow (\frac{n+1}{n+1} - \frac{1}{n+1})^{n+1} < 1/e$. This means, each element from u is covered with a constant probability. To obtain a valid set cover, independently pick $c \log n$ such subcollections and compute their union, say $\mathcal{C}'$, where c is a constant such that

$$(\frac{1}{e})^{c \log n} \leqslant \frac{1}{4n}$$

at this point we have $\Pr(\text{a not covered by } \mathcal{C}') \leqslant \frac{1}{e}^{c \log n} \leqslant \frac{1}{4n}$ and summing over all $u \in U$ we get:

$$\Pr(\mathcal{C}' \text{ not a valid set cover}) \leqslant n \cdot \frac{1}{4n} = \frac{1}{4}$$

For the cost of the obtained solution we get

$$\mathbf{E}[\text{cost}(\mathcal{C})'] \leqslant c \cdot \log n \cdot \text{OPT}_{LP}$$

Applying Markov's inequality ($\Pr(X \geqslant t) \leqslant \frac{E(X)}{t}$) with $t = \text{OPT}_{LP} \cdot 4c \log n$ we get

$$\Pr[\text{cost}(\mathcal{C})' \geqslant 4 \cdot c \log n \cdot \text{OPT}_{LP}] \leqslant \frac{1}{4}$$

So the union of the two bad events happens with probability at most $1/2$, and therefore

$$\Pr(\mathcal{C}' \text{ a valid set cover with cost } \leqslant 4 \cdot c \log n \cdot \text{OPT}_{LP}) \geqslant \frac{1}{2}$$

We can verify in polynomial time whether $\mathcal{C}'$ satisfies both conditions and repeat if not. The expected number of repetitions needed until we succeed is at most 2.

0.4 Online Algorithms

So far we have looked only at algorithms which receive their entire inputs at the beginning. We will now turn to algorithms that receive and process their inputs in partial amounts. These algorithms are called *Online Algorithms*. We will analyse them with respect to the best *offline algorithm* which knows the whole input in advance.

0.4.1 The Online Paging Problem

Consider a computer memory organized in two levels: there is a *cache* or fast memory that can store k memory items and a slower main memory, that can potentially hold an infinite number of items. Each item represents a page of virtual memory (the cache can store k of them). A paging algorithm decides which k items to keep in the cache at each point in time. We have a sequence of requests, each of which specifies a memory item. If the item requested is currently in the cache, we call this a *hit* and no additional cost is incurred. If the item is not present in the cache, it has to be loaded from main memory at unit cost and – if necessary – one other item has to be evicted from the cache. The cost measure for paging is the number of misses on a sequence of requests.

The crucial action of an online algorithm is to decide which item to evict from the cache. While an offline algorithm knows all future requests and can make use of this knowledge for the decision, the online algorithm cannot.

The following are some typical deterministic online algorithms that are used in computer systems:

LRU Least-Recently-Used: evict the item whose most recent request occurred furthest in the past

FIFO First-In-First-Out: evict the item that has been in the cache for the longest period

LFU Least-Frequently-Used: evict the item in the cache that has been requested least often

Now consider a sequence $\rho_1, \rho_2, \ldots, \rho_N$ of requests. What is a good offline algorithm which minimizes the number of misses ? **MIN** always evicts the item whose next request occurs furthest in the future. **MIN** needs to know about the request sequence in advance and in fact is optimal w.r.t. the number of misses. For an algorithm A to be examined, let $f_A(\rho_1, \ldots, \rho_N)$ be the number of misses which happen when using algorithm A to evict items, $f_{OPT}(\rho_1, \ldots, \rho_N)$ the number of misses for algorithm **MIN**.

Definition 0.25. A deterministic online paging algorithm A is called *C-competitive* if there exists a constant b such that on every sequence of requests $\rho_1, \ldots, \rho_N$ we have

$$f_A(\rho_1, \ldots, \rho_N) \leqslant C \cdot f_{OPT}(\rho_1, \ldots, \rho_N) + b$$

where C and b must be independent of N.

So the competitiveness measures the performance of an online algorithm in terms of the worst-case ratio of its cost to that of the optimal online algorithm on the *same* request sequence. This type of analysis is also called *competetive analysis*. Another way of analysing an online algorithm is to look at its expected number of misses that occur on a request sequence generated according to a probability distribution. This is called *average case analysis*. We will focus on *competetive analysis* first.

Lemma 0.26. *The LRU algorithm is k-competetive.*

Proof. Consider a request sequence $\rho = \rho_1, \ldots, \rho_N$. Without loss of generality assume that LRU and MIN start with the same cache contents.

We partition ρ into phases $P(0), P(1), \ldots$ such that LRU has at most k misses on $P(0)$ and exactly k misses on $P(i)$ for $i > 0$. We start at the end of ρ and scan the request sequence. Whenever we have seen k misses made by LRU we switch to a new phase. It remains to show that MIN has at least one miss per phase.

For phase $P(0)$ no argument is required as MIN gets a miss when LRU does (they start with the same cache contents). Now look at a phase $P(i)$, $i > 0$. Let ρ_{t_i} be the first and $\rho_{t_{i+1}-1}$ the last request in that phase. Furthermore let $x = \rho_{t_i-1}$. We claim that in $P(i)$ there are requests to k distinct items that are all different from x. If so, MIN clearly also has a miss which finishes the proof.

If the k misses that LRU produces are all caused by distinct items (also different from x), we are done. Otherwise, assume there is an item y in $P(i)$ on which LRU misses twice: so assume LRU has a miss on $\rho_{s_1} = y$ and $\rho_{s_2} = y$ with $t_i \leqslant s_1 < s_2 < t_{i+1}$. After the request at time s_1, y is in the cache. When it is thrown out at some time $s_1 < t < s_2$, this happens because it is the least recently requested item at that time, i.e. between times $s_1 + 1$ and t, at least k distinct items must have been requested, all of which were different from y. So including y, at least $k+1$ different items must have been requested during phase $P(i)$ at least k of which are different from x.

If there is no item y that is missed twice, but one of the misses of LRU is on x during phase $P(i)$, the same argumentation holds.

□

As it turns out, one cannot do (deterministically) than LRU as the following lemma proves.

Lemma 0.27. *Let A be a deterministic online paging algorithm which is C-competetive. Then $C > k$.*

Proof. Let $S = \{x_1, \ldots, x_k, x_{k+1}\}$ be a set of $k+1$ items. W.l.o.g. we assume that A and MIN have $x_1, \ldots, x_k$ in their cache at the beginning. Consider the request sequence where the next request is always the one that is not in A's cache.

Clearly A misses on each request. When MIN has a miss on request ρ_t, it evicts an item that is not requested in the next $k-1$ requests, so for k consecutive requests MIN produces only one miss. □

Still we have not shown that MIN is indeed an optimal algorithm for the paging problem.

Lemma 0.28. *On every request sequence MIN produces the minimum number of misses.*

Proof. The idea of the proof will be to consider any other algorithm A which serves the first t requests, $t \geqslant 0$ in the same way as MIN but produces less misses than MIN, and modify it to an algorithm A' which serves the first $t+1$ the same way MIN does without increasing the number of misses. Repeating this idea several times shows that A can be transformed into MIN without increasing the number of misses.

Let x be the item requested by $\rho(t+1)$. Assume MIN evicts u for serving this request, A evicts v, $v \neq u$. Define A' as follows: A' serves $\rho(t+1)$ by evicting u and then works in the same way as A until one of the two events happens:

- there is a miss at a request to page y, $y \neq v$ and A evicts u. In this case, A' evicts v and A' is in the same state as A and has incurred the same number of misses.
- there is a miss at a request to page v, and A evicts z. In this case A' evicts z and loads u; now A' is in the same state as A and has incurred the same number of misses

By definition of MIN, a request to u cannot occur earlier than a request to v. □

0.4.2 Randomized Paging

We have seen that LRU with its k-competetiveness is the best possible. But the proof that no better competitiveness than k is possible only holds for *deterministic* algorithms. So if we allows randomness in the decisions of the paging algorithm, we might achieve better bounds.

We start with a formal definition what c-competetiveness means for a randomized algorithm.

Definition 0.29. A randomized online paging algorithm A is called *C-competitive* if there exists a constant b such that on every sequence of requests $\rho_1, \ldots, \rho_N$ we have

$$E[f_A(\rho_1, \ldots, \rho_N)] \leqslant C \cdot f_{OPT}(\rho_1, \ldots, \rho_N) + b$$

where the expectation is taken over the random choices of A and C and b must be independent of N.

Now consider the simplest randomized paging algorithm called **RANDOM** which always evicts a *random* page from the cache when needed. Unfortunately this algorithm will not help a lot as the following lemma by Raghavan/Snir shows.

Lemma 0.30. *RANDOM is no better than k-competitive.*

Proof. Consider the request sequence $\rho = x_1x_2x_3\ldots x_k(y_1x_2\ldots x_k)^l(y_2x_2\ldots x_k)^l\ldots$.

Clearly OPT has one miss in each subsequence $(y_1x_2\ldots x_k)^l$. At the beginning of each subsequence, RANDOM has at most $k-1$ of the requested items in memory. So a miss has to occur during that subsequence. We call such a miss a *good miss*, if the item not requested in that subsequence is evicted. During the l repetitions of the subsequence, at least one miss occurs in each round until a good miss happens. If $k-1$ of the requested items are in cache, the probability for evicting the right item when a miss occurs is $1/k$. The expected number of rounds until this happens is determined by E[# rounds till good miss]$\geqslant \sum_{i=0}^{l} i \cdot (\frac{k-1}{k})^{i-1} \cdot \frac{1}{k} = \frac{1}{k-1}\sum_{i=0}^{l} i \cdot (\frac{k-1}{k})^{i} = \frac{1}{k-1}\sum_{i=0}^{l} i \cdot c^i = \frac{1}{k-1} \cdot \frac{lc^{l+2}-(l+1)c^{l+1}+c}{(1-c)^2} = O(k)$. □

So simple application of randomization does not help, but a more sophisticaed variant performs better than any deterministic algorithm.

Algorithm MARKING: The algorithm processes a request sequence in phases. At the beginning of each phase, all items in the cache are unmarked. Whenever a page is requested, it is marked. On a miss, a page is chosen uniformly at random from among the unmarked in the cache and evicted. A phase ends when all items in the cache are marked and a miss occurs. Then all marks are erased and a new phase is started.

Lemma 0.31. MARKING *is $2H_k$-competitive.*