Linear Programming

Introduction

Aim

- **Linear programming is an optimization technique for a system of linear constraints and a linear objective function.**
- Here, "linear" means an expression of the form $\sum_i a_i x_i + b$ where x_i's are variables and b is a constant.
- **The objective function** defines the quantity to be optimized. The aim of linear programming is to find the values of the variables (decision variables that will decide the output) that maximize or minimize the objective function while satisfying all the constraints.
- Linear programming finds wide use in supply chain operations, optimizing delivery routes, shelf space optimization, machine learning etc.

Algorithms

- The simplex algorithm for solving linear programming problems performs well in practice but it takes exponential time in the worst case.
- However, linear programming has been solved in polynomial time by algorithms like the ellipsoid algorithm, and the Karmarkar's algorithm which led to increased research in interior-point methods for linear programming.
- LP-duality is used as a prime proof technique in several algorithms and associated theorems which we will look at subsequently.

Modelling the problem

The general problem

Maximize (or) minimize objective function $Z = c_1x_1 + c_2x_2 + c_3x_3 + ...c_nx_n$ subject to the constraints:

$a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n \leq \text{or} = \text{or} \geq b_1$

$a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n \leq \text{or} = \text{or} \geq b_2$

...

$a_{m1}x_1 + a_{m2}x_2 + ... + a_{mn}x_n \leq \text{or} = \text{or} \geq b_m$

and the **non-negativity restrictions** $x_1, x_2, x_3, ...x_n \geq 0$.

Basic Terminology

- If x satisfies $Ax = b$, $x \geq 0$, then x is a **feasible solution**.
- A linear program is feasible if $\exists$ a feasible solution, otherwise it is said to be infeasible.
- An **optimal solution** x^* is the minimum (or maximum) out of all feasible solutions.
- A linear program is **unbounded from below (or above)** if $\forall \lambda \in$ R, $\exists$ a feasible x^* s.t. $c^T x^* \leq (or \geq) \lambda$ where c is the coefficient vector of the objective function.

Canonical and Standard Forms

Let the decision variables in a linear program be $x_1, x_2, x_3, ...x_n$, subject to linear equality or inequality constraints. To describe properties of and algorithms for linear programs, it is convenient to express them in similar forms.

Canonical form

A linear program is said to be in canonical form if it satisfies the following conditions:

- Objective function requires to be maximised
- Permits only $\leq$ constraints
- The decision variables must be non-negative

To convert any linear program to its canonical form, we

- Replace any constraint of the form $ax \geq b$ with $-ax \leq -b$.

- Equality constraint of the form $ax = b$ can be modeled with two inequalities: $ax \leq b$ and $ax \geq b$ which in canonical form reduces to $ax \leq b$ and $-ax \leq -b$.
- Change a minimization problem to a maximization problem by negating the objective function.
- Replace any unconstrained or negative variable x with the difference of two new variables $x^+ - x^-$ where $x^+, x^- \geq 0$.

Standard form

A linear program is said to be in standard form if it satisfies the following conditions:

- RHS (constant side) of the constraints needs to be non-negative
- Permits only $=$ constraints
- The decision variables must be non-negative

To convert any linear program to its standard form, we

- Replace any constraint of the form $ax \geq (or \leq) b$ where $b < 0$ with $-ax \leq (or \geq) - b$.
- Add a nonnegative variable to $\leq$ constraints (called a slack variable) and subtract a nonnegative variable from $\geq$ constraints (called a surplus variable). For example: By introducing the slack variable $y \geq 0$ in the inequality $ax \leq b$, we can convert it to $ax + y = b$ and by introducing the surplus variable $y' \geq 0$ in the inequality $a'x' \geq b'$, we can convert it to $a'x' = b' + y'$.
- Replace any unconstrained or negative variable x with the difference of two new variables $x^+ - x^-$ where $x^+, x^- \geq 0$.

Convexity

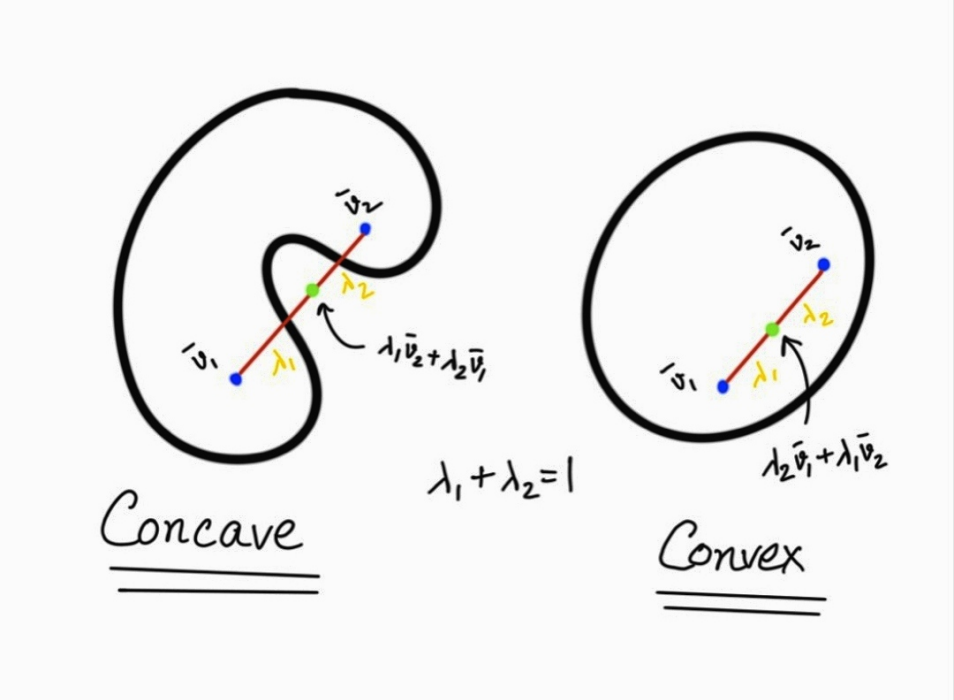
Convex Combinations

$\lambda_1 v_1 + \lambda_2 v_2 + ...\lambda_n v_n$ is a convex combination of points $v_1, v_2 ... v_n$ if $\lambda_1 + \lambda_2 + ...\lambda_n = 1$ and $\lambda_i \geq 0$ for $1 \leq i \leq n$.

Convex Set

A set S in R^n is convex if it contains all the convex combinations of the points in S.

A rough illustration for two-dimension ($n = 2$) for the sake of visualization:

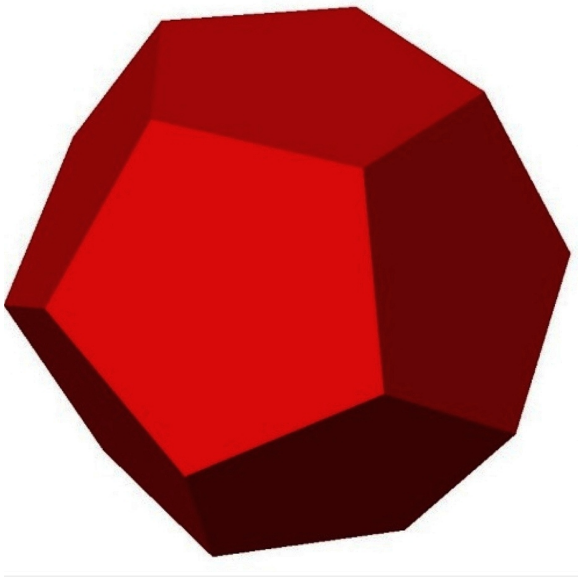


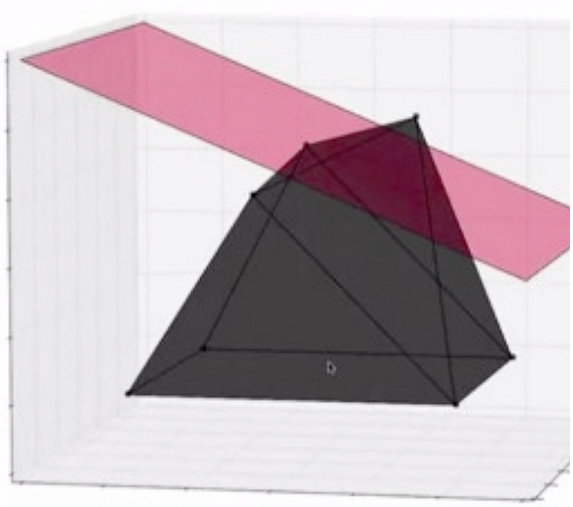
Some useful points

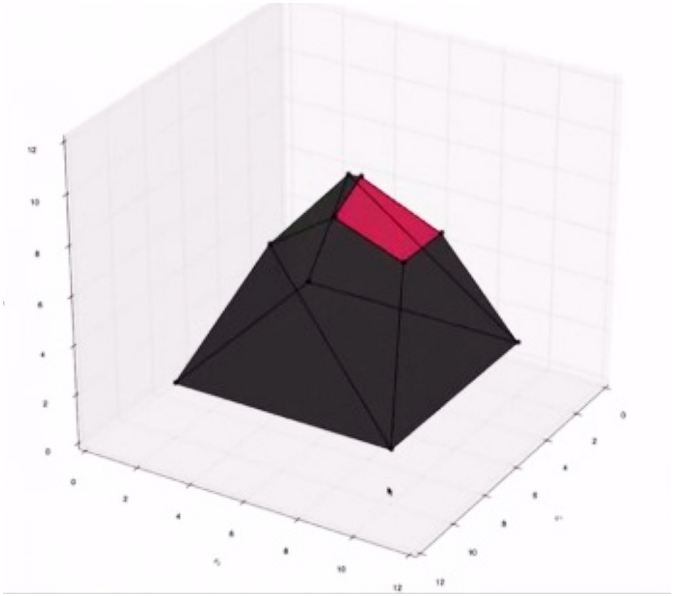
- The intersection of convex sets is a **convex set**. (easy to prove)
- A **hyperplane** is a subspace whose dimension is one less than that of its ambient space (surrounding space). For instance, a line cutting across a plane.
- A **half-space** is either of the two parts into which a hyperplane divides a space.
- A **half space** can be represented by the linear inequality $a_1x_1 + a_2x_2 + ... + a_nx_n \geq b$. It is a convex set.
- The intersection of a set of half-spaces and is called a **polyhedron**. If it is finite, it is called a **polytope**.
- A **face** is an intersection of finitely many hyperplanes.

- For n variables, a face of dimension $n-1$ (in one hyperplane) is called a **facet** and a face of dimension 0 (in n hyperplanes) is called a **vertex**.
- **Every point in a polytope can be expressed as a convex combination of its vertices.**

> A rough intuitive proof can be thought using induction. For 1 vertex (dim V = 0) its trivially true, we assume it true for dim V = k. When we add extend the polytope to the next dimension (dim V = k+1), we consider lines passing through a new points in the polytope. These lines will intersect the faces of the polytope at two points. Now, any point on the face can be expressed as a convex combination of respective vertices (as dim V = k). Hence, the new points can eventually be expressed as a convex combination of the vertices.







Geometric view

Theorem

At least one of the points where the objective function is minimal is a vertex.

💡 Intuition:
Imagine the objective function (Z) as a hyperplane. Divide the Eucliedian space into level-sets for different values which Z can take. Now we can consider maximizing/minimizing Z as driving its hyperplane up/down. Notice that when the hyperplane enters the polytope of the feasible region, the surface in common will be either a hyperplane (a side of the polytope) or a vertex (end-point of a polytope). Similarly will be the case when leaving the polytope. As a surface contains infinitely many points, so there's always at least one vertex where the objective function is minimal.

Proof

Let x^* *be the minimum. As each point in a polytope is a convex combination of the vertices* $v_1, v_2 ... v_t$*, we have*
$x^* = \lambda_1 v_1 + \lambda_2 v_2 + ... \lambda_t v_t$ where $\lambda_1 + \lambda_2 + ... \lambda_t = 1$

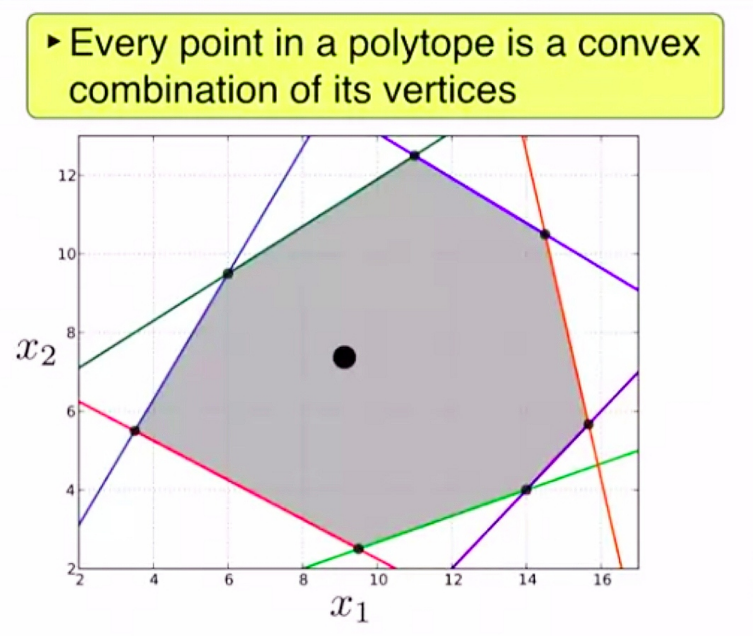
and the objective function value at optimality can be expressed as
$cx^* = \lambda_1 (cv_1) + \lambda_2 (cv_2) + ... \lambda_t (cv_t)$ *where c is transpose of coefficient vector and* x^* and v_i s are position vectors of the concerned points.

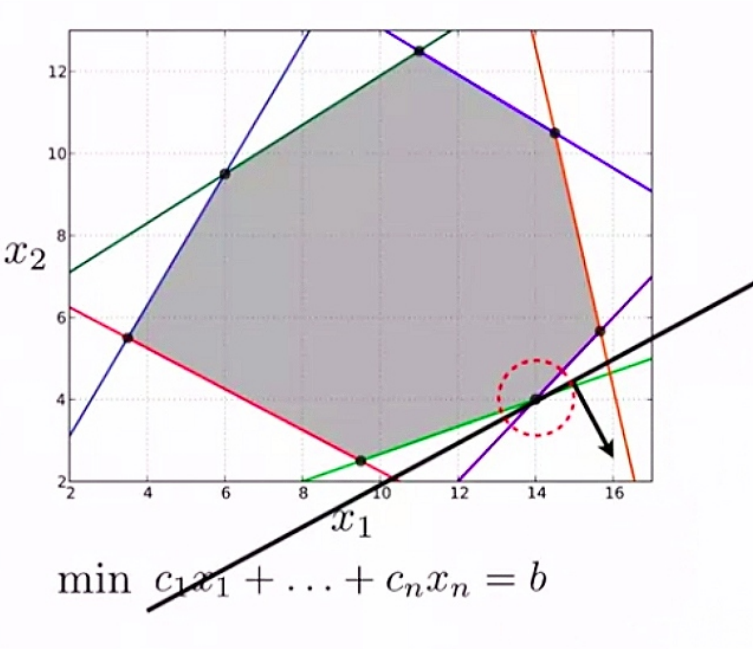
Now, let us assume that the minimum is not at a vertex, i.e.,
$cx^* < cv_i$ for $1 \leq i \leq t$.

Therefore,

$cx^* = \lambda_1 cv_1 + \lambda_2 cv_2 + ... \lambda_t cv_t > \lambda_1 cx^* + \lambda_2 cx^* + ... \lambda_t cx^* = (\lambda_1 + \lambda_2 + ... \lambda_t) cx^* = cx$ **which is a contradiction.**

Hence, x^* must be one of the v_i where $1 \leq i \leq t$.



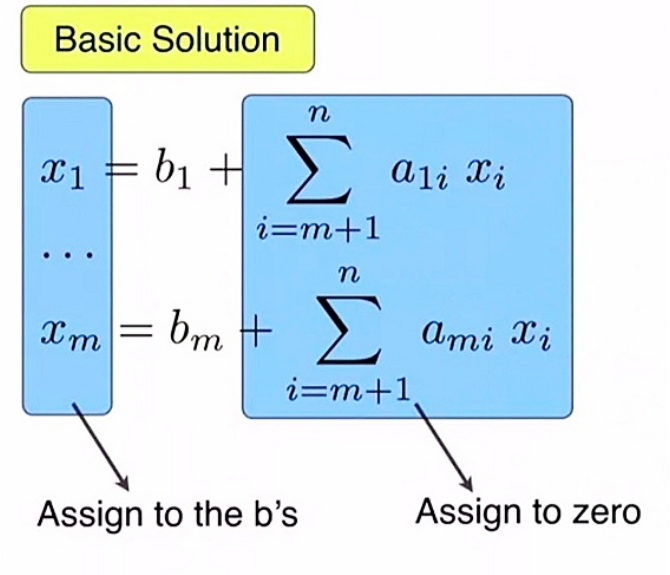


Simplex Method

Intuition

Goal: **To solve a linear program**

For a set of linear equations, **a basic solution** can be thought of as a solution where x_1 , ...x_m take the values $b_1, ...b_m$ respectively and all other x_i 's are assigned the value 0 as shown below:



Useful Points:

- A **Basic Feasible Solution (BFS)** essentially requires b_i's to be non-negative as for a linear program, all the x_i's need to take non-negative values.
- An **optimal solution is located at a vertex**.
- A **vertex is a BFS**.
- We can move from one BFS to a neighbouring BFS.
- We can detect whether a BFS is optimal.
- From any BFS, we can move to a BFS with a better cost.

Use in Linear Programs:

- For a linear program, we first express it in its canonical form and add corresponding slack variables to convert the $\leq$ inequalities into equalities. We modify the objective function so that we need to minimize it.
- *To get a basic solution, we essentially select m x_i's and put them on the LHS and express them in terms of the other variables (called non-basic variables)*

using Gaussian Elimination.

- *Then, we assign them the values of the constants on the RHS and then put the rest x_i's as 0 to get a BFS (vertex) of the linear program.*

> It is easy to see why this gives a vertex as it is the single point that intersects with all m constraints on the boundary.

Naive algorithm

The naive algorithm for this method would be to iterate over all the vertices and get the optimal solution. However, there will be ${}^{n}c_{m}$ such possible combinations that do not work well computationally for large cases.

Algorithm

Goal: **To minimise the objective function while satisfying the constraints.**

The key idea of the simple algorithm is to move from one BFS to a better BFS till we reach the optimal (smallest value of objective function). We are guaranteed to find an optimal due to the convexity of the solution space.

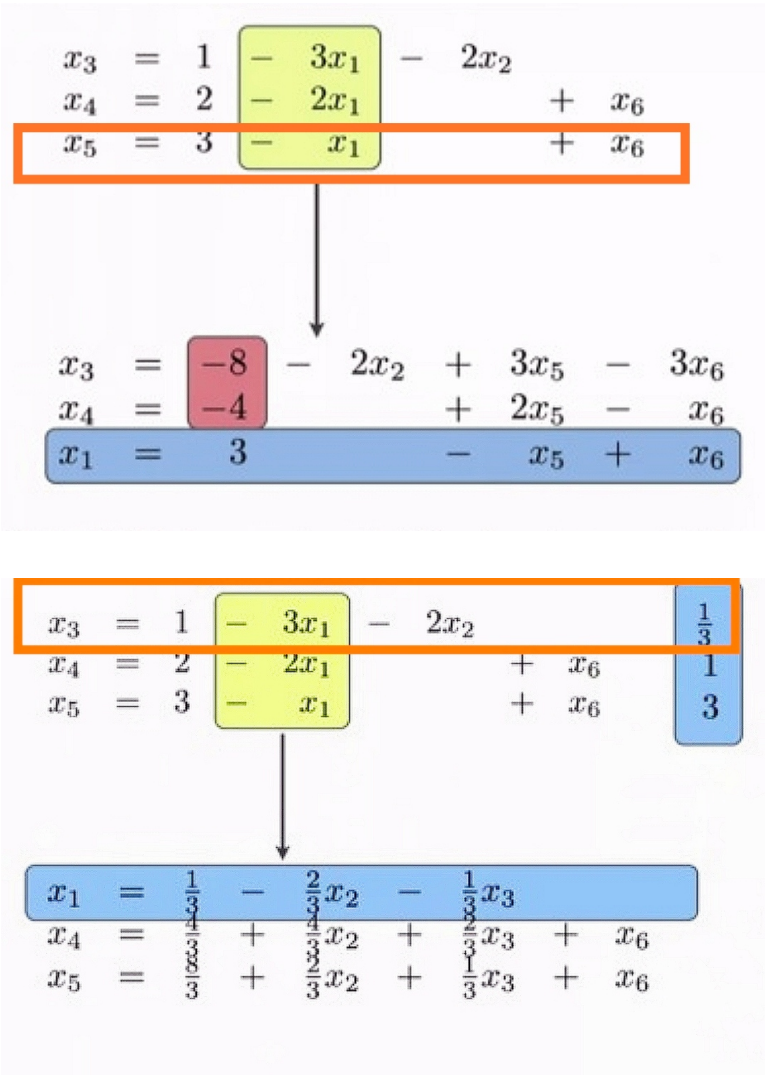
So, **the basic steps we would take to move from one BFS to another BFS given we are at an BFS already is as follows**:

- Select a non basic variable with negative coefficient in the objective function since assigning a positive value to it will drive the objective function down - entering variable
- Inroduce this variable in the basis by removing a basic variable from the equation where the coefficient of the entering variable is negative - leaving variable
- Perform Gaussian Elimination to get to the form of a basic solution and do the same with the objective function by expressing it in terms of only the non basic variables.
- **When we reach a point where the objective function is of the form $c_0 + \sum c_i x_i$ where all $c_i \geq 0$, we have reached the optimal solution. This is true because we cannot bring down the objective function any further since**

assigning a positive value to any of the variables will increase the objective function.

Things to keep in mind:

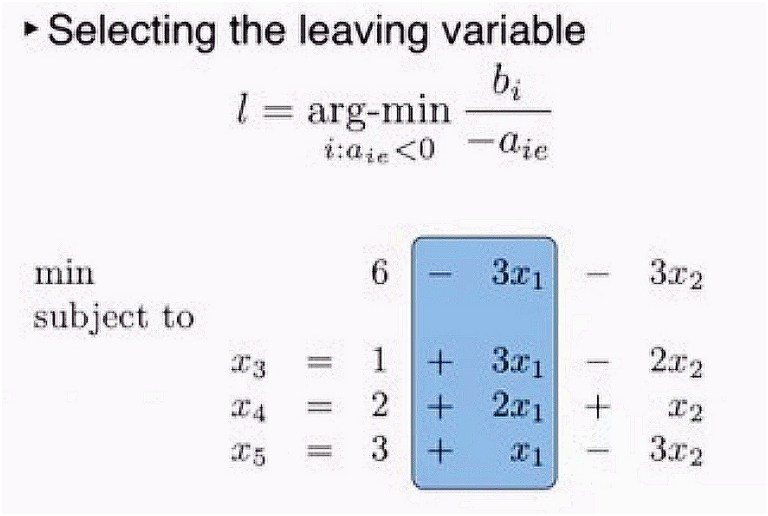
- For the entering variable, we swap the variable in the equation with minimum $\frac{b_i}{-(\text{coeff of entering variable})}$ so that we ensure that b_i s remain non-negative.



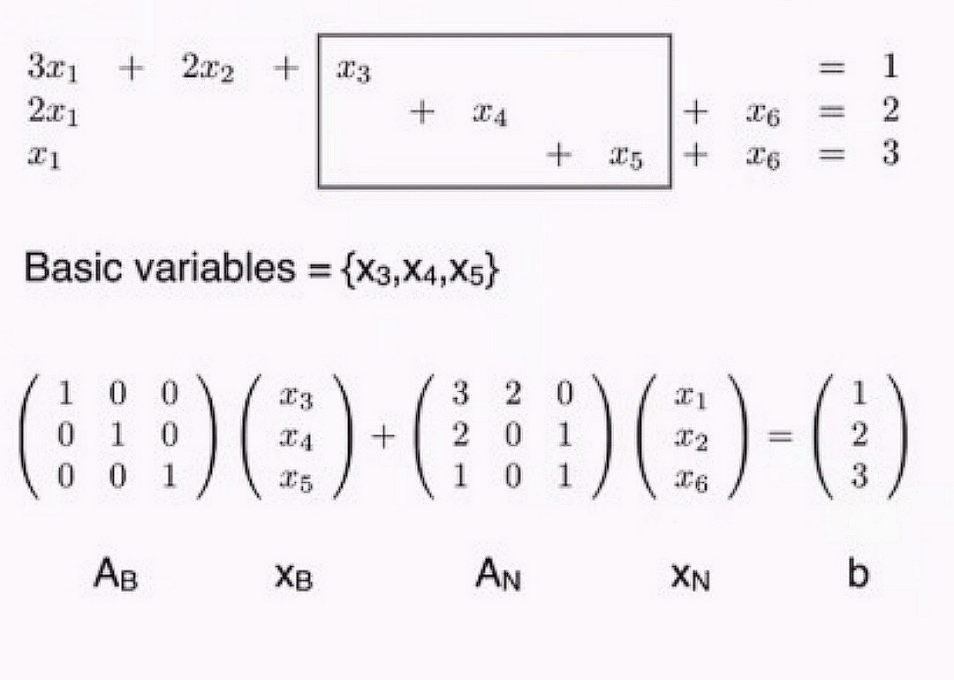
- If at a point we find, $b_i = 0$, we might get stuck between BFS with the same value. To ensure termination in this situation, we can use the following methods:-

- Bland Rule - Selecting the first entering variable lexicographically *(This guarantees termination of Simplex!)*
- Lexicographic pivoting rule
- Perturbation methods

- If at a point we see that the variable we have chosen to insert in the basis has no negative coefficient in the constraint equations, it means that we can drive the value of the variable arbitrarily low without violating any of the non negativity constraints. This implies that the linear program is unbounded and hence impractical.



Matrix representation



Simplex algorithm in matrix notation

1. Consider $Ax = b$
2. Choose m linearly independent columns A_B
   $Ax = A_B x_B + A_N x_N = b$
   $A_B x_B = b - A_N x_N$
   $x_B = A^{-1}Bb - A^{-1}BA_N x_N$
   $x_B = b^{'} - A_N^{'} x_N$
1. x_B is feasible if $b' \geq 0$
2. The matrix A_B is called a basis.
3. Cost for basis B is
   $cx = c_B x_B + c_N x_N = c_B(A^{-1}Bb - A^{-1}BA_N x_N) + c_N x_N$
   $= c_B A^{-1}Bb + (cN - c_B A^{-1}BA_N)x_N + (c_B - c_B A^{-1}BA_N)x_B$
   $= c_B A^{-1}Bb + (c - cBA_B^{-1}A)x$
4. We define $\prod = c_B A_B^{-1}$ and call it the simplex multiplier.
   Therefore, $cx = \prod b + (c - \prod A)x$

5. $\prod b$ is the optimal solution when $c \geq \prod A$.

Proof:

$cx = \prod b + (c - \prod A)x$

We have already seen that the coefficients of all the variables being non-negative guarantees that we have reached the optimal solution. Hence in the case of the optimal solution,

$(c - \prod A) \geq 0 \implies c \geq \prod A$

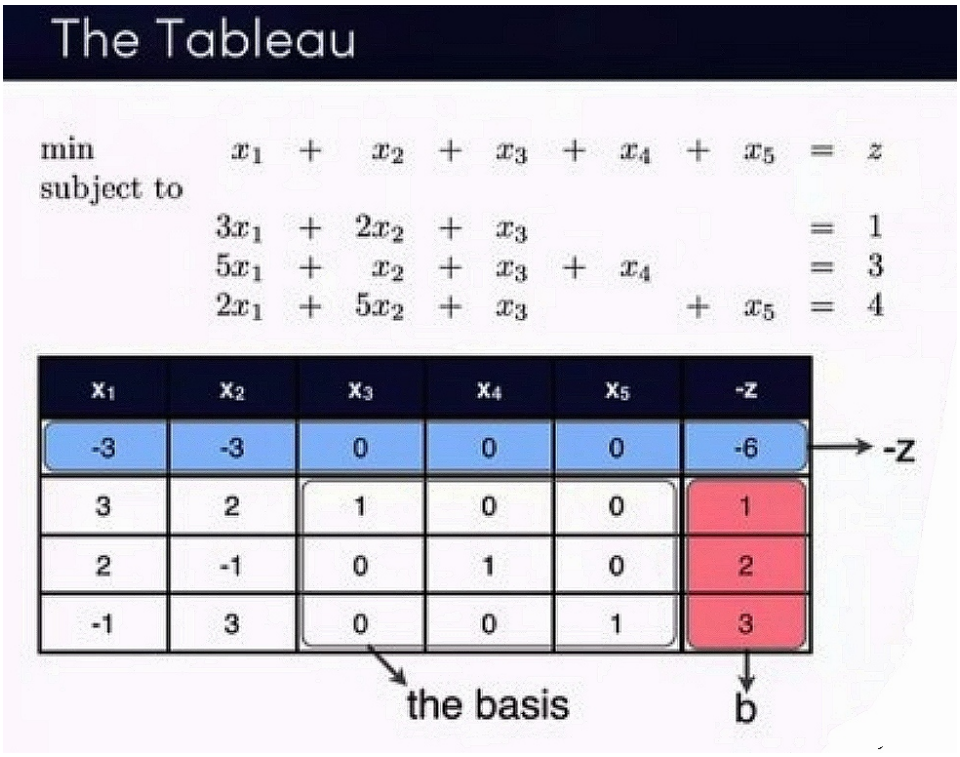
Now, let $\prod b = c_0$

Let y be a feasible solution. Then, we have $Ay = b$.

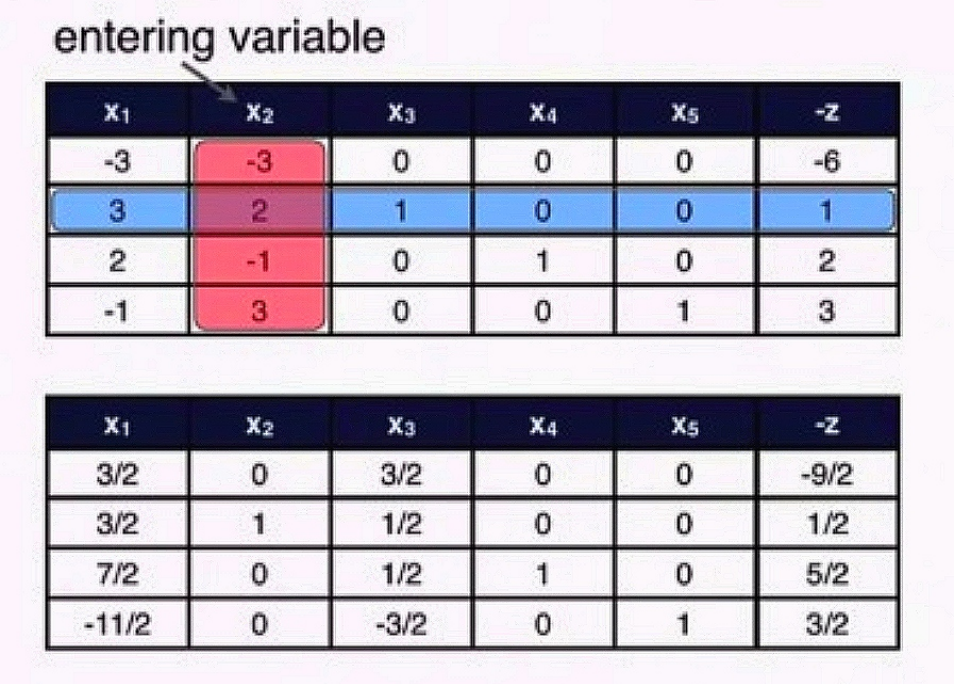
Hence, $cy \geq \prod Ay = \prod b = c_0$.

Converting to tableau

Basically, **in comparison to what we did earlier, we bring the non-basic variables too on the LHS and send constant c_0 in z to RHS whose value will ultimately give the optimal value for z.** Then we make a table out of it to ease our calculations.



Simplex algorithm using tableau



When we see that all coefficients in the first row are non-negative, we have reached the optimal solution. In the given case, it is $\frac{9}{2}$.

Duality

Primal and dual

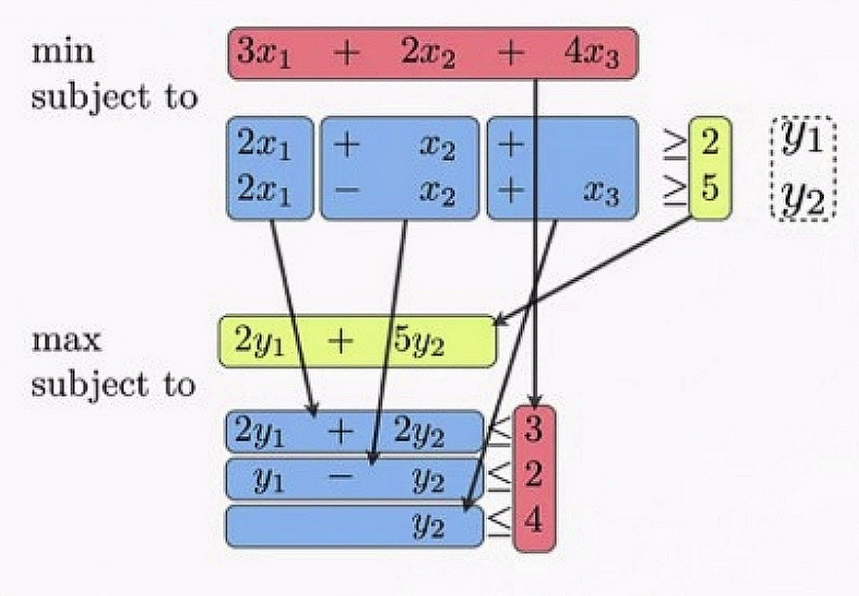
We define the primal and dual as follows:

Primal

Minimize cx subject to $Ax \geq b$ and $x_j \geq 0$ where $1 \leq j \leq n$ on the m constraints.

Dual

The dual of the primal mentioned above is as follows:
Maximize yb subject to $yA \leq c$ and $y_i \geq 0$ where $1 \leq i \leq m$ on the n constraints.



Weak Duality Theorem

For every minimization linear program, the value of the objective function of primal is always greater than that of the dual.
For every maximization linear program, the value of the objective function of primal is always less than that of the dual.

Proof:

Let x and $\prod_0$ be feasible solutions to the primal and dual respectively where the primal is a minimization linear program. We have,

$c \geq \prod_0 A \implies cx \geq \prod_0 Ax \geq \prod_0 b.$

Similarly, we can prove the result of the maximization linear program.

Strong Duality Theorem

If the primal has an optimal solution, then the dual has an optimal solution with the same cost.

Proof:

- By weak duality theorem, we can say that since the primal has a feasible solution, dual cannot be unbounded.
- Now, we already know that for the primal, in the optimal condition, $c \geq \prod A$ where $\prod$ is the simplex multiplier of the primal, which is hence a feasible solution of the dual.

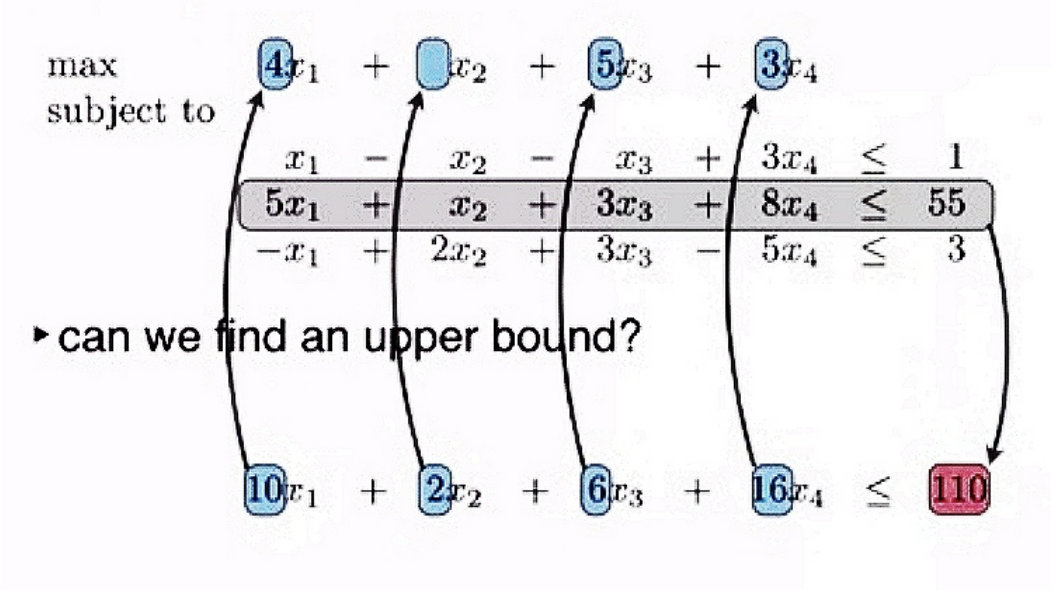
- Now, from what we have seen for the primal, let x^* *be the optimal solution of primal. Then we have the associated* $x^*B = AB^{-1}b$.
  The dual has a feasible solution, $y^* = \prod = c_B A_B^{-1}$ *from previous result.*
  Hence, $y^*b = c_B A^{-1} Bb = cBx^*$ Now, we can say that a feasible solution of the dual has the same cost as that of the optimal solution of the primal and since the cost the dual is always $\leq$ cost of the primal, this is indeed the optimal value of the dual too.

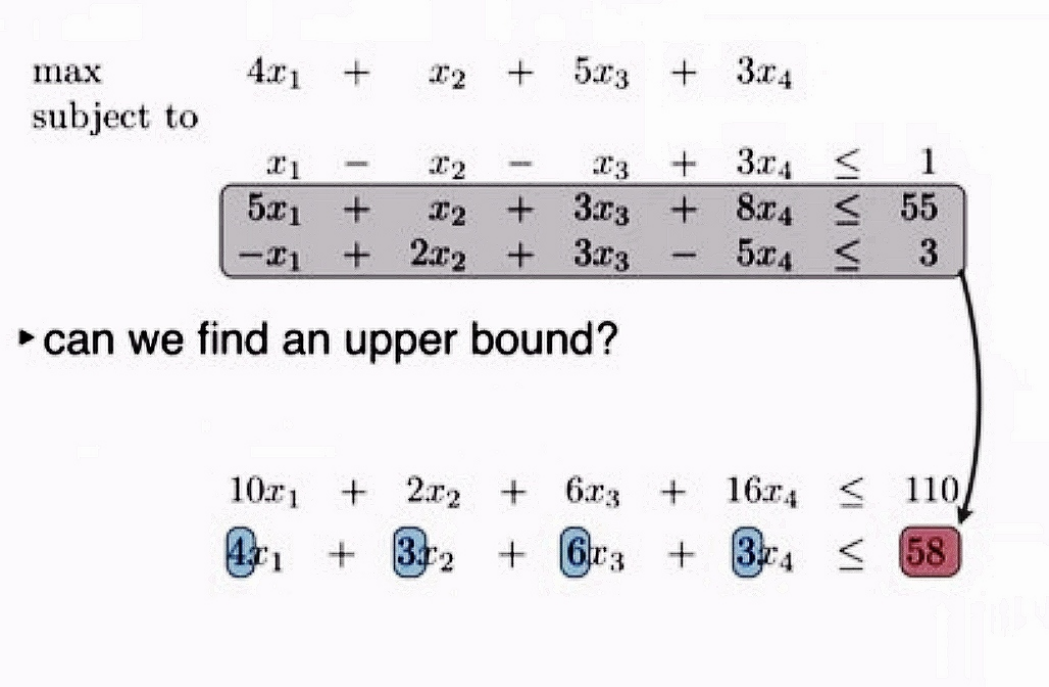
Interpretation and use

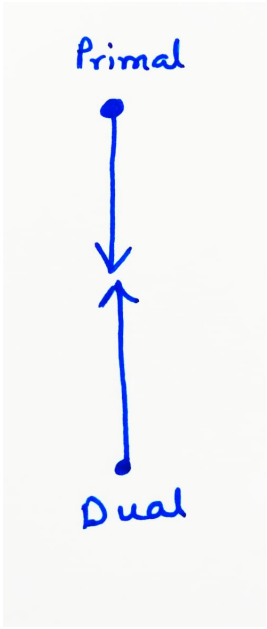
Bounding

Can we find an upper bound to the objective function we want to maximise (maximal case)?

- Yes, we can try to make a linear combination of the constraints so that each coefficient in this combination is greater than the coefficient of the corresponding variable in the objective function. The value of this combination will be an upper bound to the objective function as $\forall i \; x_i \geq 0$ (non-negativity constraint).







- We then try to minimise the value of this combination which can be treated as an objective function with the value of constraints of the original LP as the decision variables. (Does this ring any bells?)
- Yes, it is basically minimising the dual's objective function.

Complementary Slackness

The primal being: minimise cx subject to $Ax \geq b$ and $x_j \geq 0$ where $1 \leq j \leq n$ on the m constraints,
and the dual being: maximise yb subject to $A^T y \leq c$ and $y_i \geq 0$ where $1 \leq i \leq m$ on the n constraints.
Let x^* *and* y^* be the optimal solutions to the primal and the dual respectively. The following conditions are necessary and sufficient for the optimality of x^* *and* y^*:

- $\sum_{j=1}^{n} a_{ij}\ x_j^* = b_i$ *or* $y_i^* = 0$ for $1 \leq i \leq m$, and
- $\sum_{i=1}^{m} a_{ji}\ y_i^* = c_i$ *or* $x_j^* = 0$ for $1 \leq j \leq n$

Proof:
Using the fact that $x^{*T} A^T y^*$ is a 1×1 matrix, $x^{*T} A^T y^* = (x^{*T} A^T y^*)^T = y^{*T} A x^*$. Now by Strong Duality, as x and y are both optimal to their respective LPs,

$cx^* = by^*$ and $cx^* = x^{*T} c \geq x^{*T} A^T y^* = y^{*T} Ax^* \geq y^{*T} b = by^*$ *(from weak duality)*
$\iff cx^* = x^{*T} c = x^{*T} A^T y^* = y^{*T} Ax^* = y^{*T} b = by^*$
$\iff x^*(A^T y^* - c) = 0$ and $y^*(b - Ax^*) = 0$.

We note that $0 = x^*(A^T y^* - c) = \sum_{j=1}^{n} (x_j^* (\sum_{i=1}^{m} a_{ji} y_i^* - c_j))$

But, $x_j^* \geq 0$ *and* $(\sum_{i=1}^{m} a_{ji} y_i^* - c_j) \geq 0 \implies x_j^* (\sum_{i=1}^{m} a_{ji} y_i^* - c_j) = 0$ for each $j = 1, 2, ..., n$

Similarly, we deduce that $y^*(b - Ax^*) = 0$ iff $y_i^* (\sum_{j=1}^{n} a_{ij} x_j^* - b_i) = 0$ for each $i = 1, 2, ..., m$

A sample demonstration:

Let the primal be to maximise $4x_1 + x_2 + 5x_3 + 3x_4$ with the following constraints:

$$\begin{bmatrix} 1 \\ 5 \\ -1 \end{bmatrix} x_1 + \begin{bmatrix} -1 \\ 1 \\ 2 \end{bmatrix} x_2 + \begin{bmatrix} -1 \\ 3 \\ 3 \end{bmatrix} x_3 + \begin{bmatrix} 3 \\ 8 \\ -5 \end{bmatrix} x_4 \leq \begin{bmatrix} 1 \\ 55 \\ 3 \end{bmatrix}$$

After including the dual variables y_i 's,

$$\begin{matrix}\begin{bmatrix}\boxed{1y_1}\\5y_2\\-1y_3\end{bmatrix}\boxed{x_1}+\begin{bmatrix}\boxed{-1y_1}\\1y_2\\2y_3\end{bmatrix}\boxed{x_2}+\begin{bmatrix}\boxed{-1y_1}\\3y_2\\3y_3\end{bmatrix}\boxed{x_3}+\begin{bmatrix}\boxed{3y_1}\\8y_2\\-5y_3\end{bmatrix}\boxed{x_4}\leq\begin{bmatrix}\boxed{1y_1}\\55y_2\\3y_3\end{bmatrix}\\ \geq 4 \qquad\qquad \geq 1 \qquad\qquad \geq 5 \qquad\qquad \geq 3\end{matrix}$$

which at optimality,

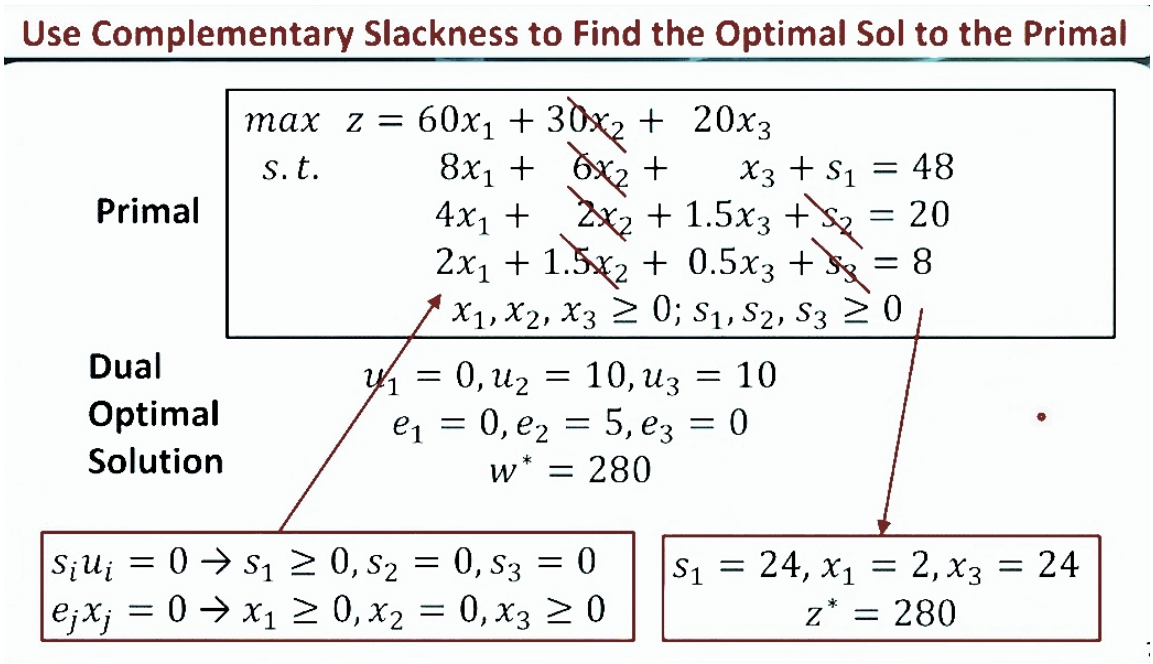
$$\begin{matrix}=\begin{bmatrix}4x_1\\1x_2\\5x_3\\3x_4\end{bmatrix}\leq\begin{bmatrix}\boxed{1x_1}\\\boxed{-1x_2}\\\boxed{-1x_3}\\\boxed{3x_4}\end{bmatrix}\boxed{y_1}+\begin{bmatrix}5x_1\\1x_2\\3x_3\\8x_4\end{bmatrix}y_2+\begin{bmatrix}-1x_1\\2x_2\\3x_3\\-5x_4\end{bmatrix}y_3\\ \leq 1 \qquad\qquad \leq 55 \qquad\quad \leq 3\end{matrix}$$

However, the extreme l.h.s and the extreme r.h.s are the same thing which forces equality in the intermediate inequalities. Thus,

$$\begin{bmatrix}\boxed{1y_1}\\5y_2\\-1y_3\end{bmatrix}\boxed{x_1}+\begin{bmatrix}\boxed{-1y_1}\\1y_2\\2y_3\end{bmatrix}\boxed{x_2}+\begin{bmatrix}\boxed{-1y_1}\\3y_2\\3y_3\end{bmatrix}\boxed{x_3}+\begin{bmatrix}\boxed{3y_1}\\8y_2\\-5y_3\end{bmatrix}\boxed{x_4}=\begin{bmatrix}\boxed{1y_1}\\55y_2\\3y_3\end{bmatrix}$$

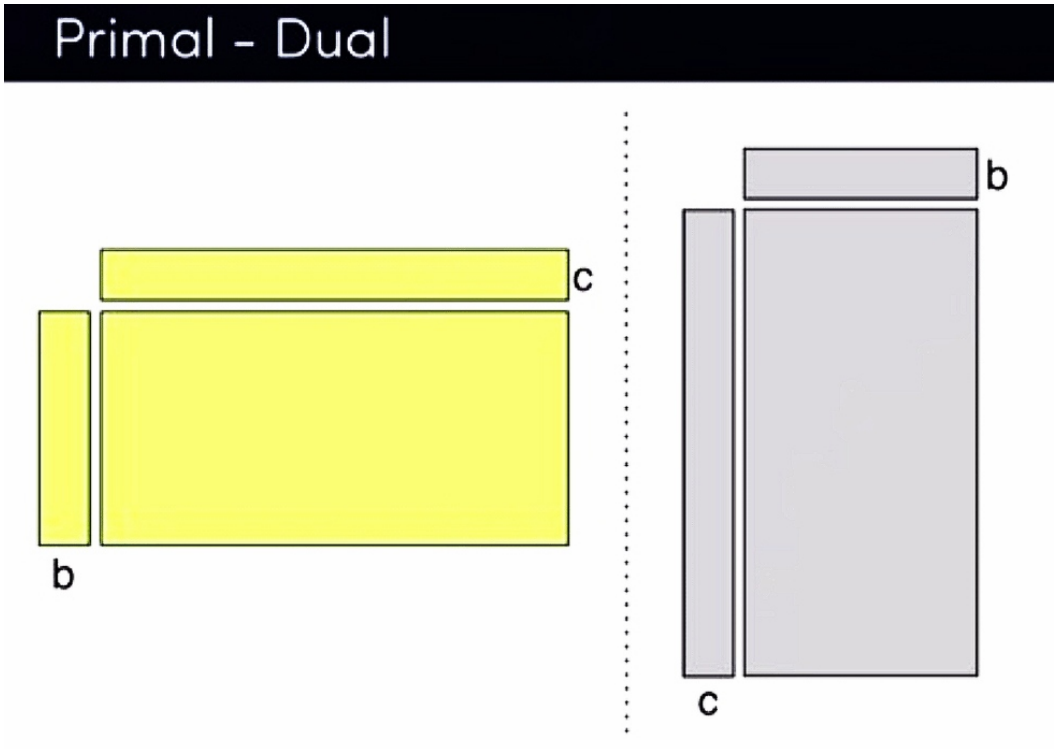
which makes it clear that $1y_1x_1 - 1y_1x_2 - 1y_1x_3 + 3y_1x_4 = 1y_1 \implies y_1(x_1 - x_2 - x_3 + 3x_4) = 0$ and similar deductions for other rows.

- This basically means that for **the optimal case, either the value of a decision variable of primal is 0 or the corresponding slack in the dual is 0.**
- We can use this property of complementary slackness to get optimal solution to the primal in cases like when we have the optimal solution and the values of dual variables. An example is given below where x_j's and u_i's are the primal's and dual's decision variables respectively and s_i's and e_j's are slack variables in primal and dual respectively.



- We can also use it to check the optimality of a solution.

Duality in the Tableau



> How to take dual? (shown for maximal primal)

Primal	Dual
variables x_1, x_2, ... x_n	n constraints
m constraints	variables $y_1, y_2, ...y_m$
objective function c	r.h.s of constraints
r.h.s b of constraints	objective function b
max c^Tx	min $b^T y$
constraint matrix A	constraint matrix A^T
i^{th} constraint \leq	$y_i \geq 0$
i^{th} constraint \geq	$y_i \leq 0$
i^{th} constraint =	$y_i \in R$
x_j \geq 0	j^{th} constraint $\geq$
x_j \leq 0	j^{th} constraint $\leq$
x_j \in R	j^{th} constraint $=$

Some Applications

Max-flow / Min-cut Problem

What is the max-flow problem?

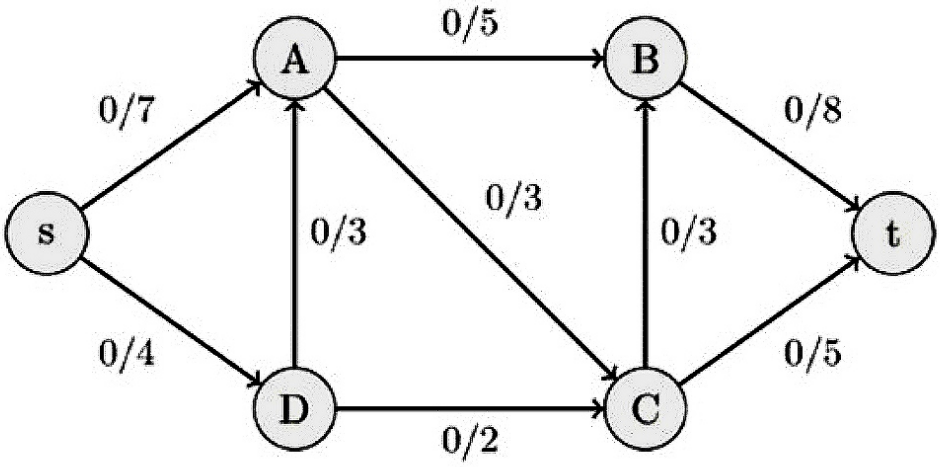
A **flow network** is a directed graph G with vertices V and edges E each of which is associated with a capacity c_e, which assigns each edge $e \in E$ a non-negative integer value. One of its vertices is a source and one is sink.

A **flow** in a flow network can be denoted by a function f, that assigns each edge e a non-negative integer value, namely the flow. The function has to satisfy the following conditions:

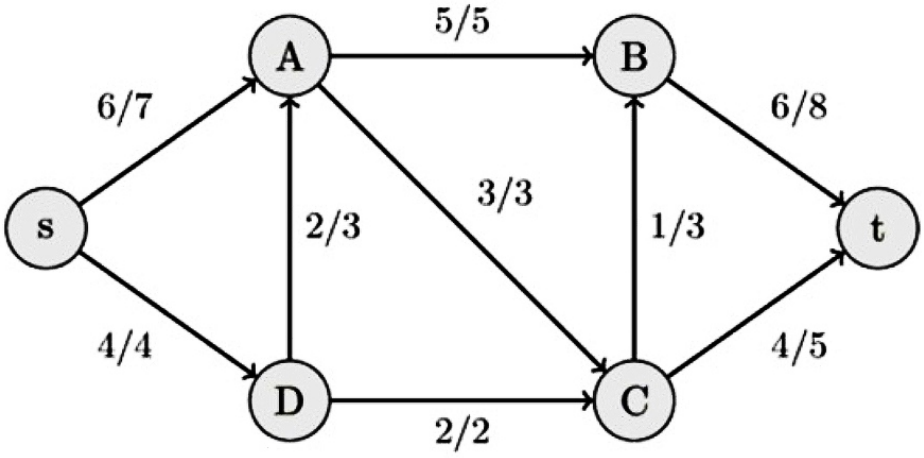
- **Capacity constraint:** The flow of an edge cannot exceed its capacity. $f(e) \leq c_e$

- **Conservation Constraint:** The sum of the incoming flow of a vertex u has to be equal to the sum of the outgoing flow of u except for the source and sink vertices. Hence,
  $\sum_{(v,u)\in E} f((v,u)) = \sum_{(u,v)\in E} f((u,v))$
- The source vertex s has only an outgoing flow, and the sink vertex t has only incoming flow.
- Also for s being the source and t being the sink, the outgoing flow of the source should be equal to the incoming flow of the sink which is basically the flow of the network.
  $\sum_{(s,u)\in E} f((s,u)) = \sum_{(u,t)\in E} f((u,t))$
- Without loss of generality, we take that s has no incoming edges and t has no outgoing edges.

A maximal flow is a flow with the maximal possible value. Our goal is to find this maximal flow of a flow network.



The maximal flow of 10 in the above flow network is as follows:



Why care about it?

The maximum flow problem can model the routing of traffic through a transportation network, data packets through a network, or oil through a pipeline network. Problems like bipartite matching and image segmentation reduce to the maximum flow problem.

Algorithms

- Naive Greedy Algorithm
    - The greedy approach to the max-flow problem can be to start with zero flow for all edges and greedily proceed from one flow to the next to send more flow on some path from s to t. However, this might not give the max-flow all the time.

> Concept of Residual Graph: Residual edges are edges in the opposite direction of flow of original edge from one vertex to other with initial flow/capacity of 0/0. Residual graph contains residual edges along with the original edges.An augmenting path is a path of edges in the residual graph with unused capacity greater than zero from the source s to the sink t.Every augmenting path has a bottleneck value and that can be used

> to augment the flow.However, till now we could only increase flow in an edge using the bottleneck but we might need to undo the update by decreasing the flow for augmenting paths that don't give max flow. We do this by using residual edges in a way that increasing the negative flow in an residual edge decreases flow in original edge.

- Ford-Fulkerson Algorithm
  - In this algorithm, we find augmenting paths through the residual graph repeatedly and augment the flow till no more augmenting paths can be found.
  - The time complexity of this algorithm is $O(maxflow * E)$ since we run a loop while there is an augmenting path and in the worst case, we may add 1 unit flow in every iteration.

```
initialize f_e = 0 for all e ∈ E
repeat
search for an s-t path P in the current residual graph G_f such that
every edge of P has residual capacity > 0
// takes O(|E|) time using BFS or DFS
if no such path then
    halt with current flow {f_e} e ∈ E
else
    diff = min e ∈ P (e’s residual capacity in G_f )
    // augment the flow f using the path P
    for all edges e of G whose corresponding forward edge is in P
    increase f_e by diff
    for all edges e of G whose corresponding reverse edge is in P
    decrease f_e by diff
```

Max-flow as a Linear Program

1. **Decision variables:** We trying to solve for a flow, specifically, the amount f_e of flow on each edge e. So, our variables are just f_e where $e \in E$.
2. **Constraints:** Taking $\delta^-(v)$ as the set of all incoming edges into vertex v and $\delta^+(v)$ as a set of outgoing edges from vertex v, conservation constraints and capacity constraints which can be written as

- $\sum_{e \in \delta^-(v)} f_e - \sum_{e \in \delta^+(v)} f_e = 0$ for every vertex $v \neq s, t$.
- $f_e \leq u_e$ for every edge $e \in E$.
- We also need to add non-negativity constraints: $f_e \geq 0$ for every edge $e \in E$.
- It can be seen that every one of these $2m + n - 2$ constraints (where $m = |E|$ and $n = |V|$) is linear.

3. **Objective function:** max $\sum_{e \in \delta^+(s)} f_e$ which is also a linear function.

Max-flow min-cut theorem

Max-flow min-cut theorem states that in a flow network, the maximum amount of flow passing from the source s to the sink t is equal to the total weight of the edges in a minimum cut.

Max-flow/min-cut theorem via duality

> Concept of an s-t cut
> An s-t cut $C = (S, T)$ is a partition of vertex set V of the flow network such that $s \in S$ and $t \in T$. Basically, an s-t cut is a division of the vertices of a network into two parts, with the source in one part and the sink in the other.
> The capacity of an s-t cut: The capacity of an s–t cut is defined as the sum of the capacity of each edge in the set containing edges going from the source's side to the sink's side.

Minimum s-t Cut Problem

Finding an s-t cut whose the capacity of is minimal. It can also be thought of as finding the the smallest total weight of the edges which if removed, would disconnect the source from the sink. The s-t cut found is called the minimum cut.

Duality view of the theorem

- **The Primal**
  To model the primal, we take path decompositions this time rather than flow. So, the decision variables, denoting the flow through the path p, have the form f_p , where p is a path from source s to sink t. Let P denote the set of all such paths. Now, there is no need to explicitly state the conservation constraints. Thus our problem now is to maximize $\sum_{p \in P} f_p$ subject to:
  - Total flow on $e \leq$ capacity of e, i.e., $\sum_{p \in P : e \in p} f_p \leq c_e \ \forall e \in E$
  - $f_p \geq 0 \ \forall \ p \in P$
  - In matrix form, it is maximising $1^T f$ where 1 is the p dimensional vector
    $$\begin{bmatrix} 1 \\ 1 \\ . \\ . \\ . \\ 1 \end{bmatrix}$$
    subject to $Af \leq c$ and $f \geq 0$.
- **The Dual**
  The dual can be written as follows in matrix form taking l_e as the decision variables:
  - Minimise $c^T l$ subject to $\sum_{e \in p} l_e \geq 1 \ \forall \ p \in P$ and $l_e \geq 0 \ \forall e \in E$
  - The key observation is that every s-t cut corresponds to a feasible solution to this dual linear program. To see this, we fix a cut (S, T), with $s \in S$ and $t \in T$, and set
    $l_e = 1$ if $e \in \delta^+(S)$ where $\delta^+(S)$ is the set of outgoing edges from S and 0 otherwise.
  - We can see that the constraints are satisfied as l_e being either 0 or 1 satisfies the non-negativity constraint and as every s-t path must cross the cut (S, T) at some point since it starts in S and ends in T. Thus, every s-t path has at least one edge e with $l_e = 1$.
  - The objective function is hence $\sum_{e \in p} c_e l_e = \sum_{e \in \delta^+(S)} c_e =$ capacity of (S, T).

- Hence, the dual is basically minimising the capacity of s-t cuts in the flow network. This is the minimal-cut problem.
- However, we have considered only the 0-1 solutions of the dual. So, by weak duality, we can say that,
  $max - flow = optimal\ value\ of\ primal \leq$
  $optimal\ value\ of\ dual \leq min - cut.$
- By strong duality, if a max-flow value exists, then it must be equal to the min-cut value.

Minimum-Cost flow

What is the mininum cost flow problem?

In the minimum cost flow problem, we have:

- a directed graph $G = (V, E)$
- a source $s \in V$ and sink $t \in V$
- a target flow value d
- a non-negative capacity u_e for each edge $e \in E$
- a real-valued cost per flow unit c_e for each edge $e \in E$.

The goal is to compute a flow f with value d i.e., pushing d units of flow from s to t, subject to the usual conservation and capacity constraints so that the cost is minimized, i.e., $min \sum_{e \in E} c_e f_e$ where $e \in E$.

It is to be noted that for each edge e that since c_e is “per-flow unit” cost, the contribution of edge e to the overall cost with f_e units of flow is $c_e f_e$.

Minimum Cost Flow as Linear Program

Taking the minimum cost flow problem as defined earlier, the linear objective function is simply

$min \sum_{e \in E} c_e f_e$

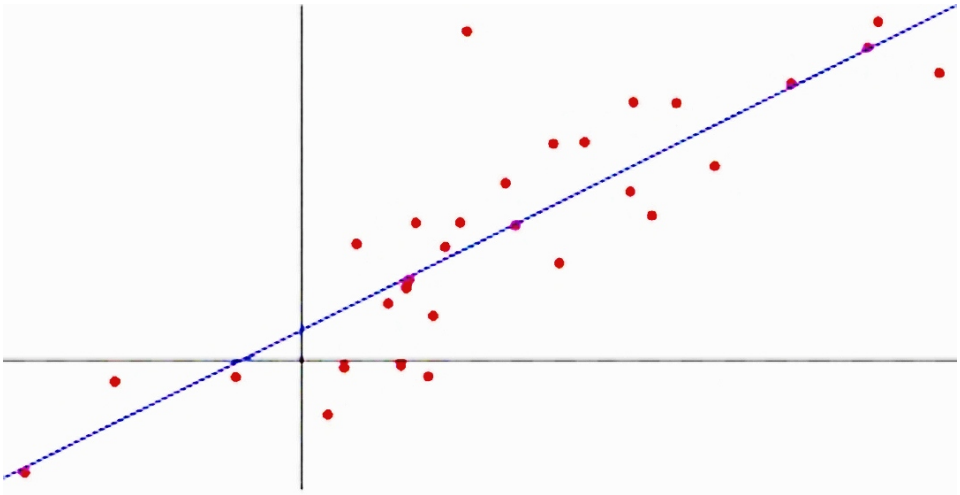
where $e \in E$ as defined earlier.
We need to add the following constraint to the usual capacity and conservation

constraints:

$\sum_{e \in \delta^+(s)} f_e = d$ where d is the target flow value.

The flexibility of linear programs can be seen from the fact that if we want to impose a lower bound, say l_e on the amount of flow on each edge e, in addition to the usual upper bound u_e, we can just replace "$f_e \geq 0$" by $f_e \geq l_e$.

Linear Regression



The problem

We want to fit a line to data points. It is frequently used in basic problems in machine learning. Formally, the input consists of m data points $p_1, ..., p_m \in R^d$, each with d real-valued "features" (coordinates). These coordinates can be thought of as properties of your data, say for $d = 3$, it can be the profession, income, and the number of years of experience of an employee.

The input also consists of a "label": $l_i \in R$ for each point p_i which for example, can be the expenditure of the employee. Thus, l_i can be thought of as the output we get from our data model for a particular point p_i.
The p_i's and l_i's are fixed and are parts of the input. They are not decision variables.

Informally, the goal is to express the l_i's as effectively as possible as a linear function of the p_i's.

Thus, the goal is to compute a linear function $h : R^d \to R$ such that $h(p_i) \approx l_i$ for every data point p_i.

Computing a “best-fit” linear function finds common use in prediction and data analysis.

- In the case of prediction, we use labeled data to identify a linear function h that, at least for data points whose labels are known, does a good job of predicting the label l_i from the feature values p_i.
  Then we hope that this linear function also makes accurate predictions for other data points for which the label is not already known, i.e., assuming it "generalizes".
- In the case of data analysis, the goal is to understand the relationship between each feature of the data points and the labels, and also the relationships between the different features. As a simple example, it might be interesting to know when one of the d features is much more strongly correlated with the label l_i than any of the others.

Linear Regression as LP

We now show that computing the best line, for one definition of “best-fit” reduces to linear programming.

Every linear function $h : R^d \to R$ has the form:

$h(z) = \sum_{j=1}^{d} a_j z_j + b$

for coefficients $a_1, ..., a_d$ and intercept b.

Hence, the coefficients a_j and b can be taken as the decision variables.

But, what’s our objective function?

Clearly if the data points are colinear we would want to compute the line that passes through all of them. But that is an ideal case, so we must compromise between how well we approximate different points.

- For a given choice of $a_1, ..., a_d$, b, we define the error on point i as:

  $E_{p_i}(a, b) = |(\sum_{j=1}^{d} a_j p_{ij} - b) - l_i|$

where p_{ik} is the k^{th} feature in point p_i.
Geometrically, when $d = 1$, we can think of each (p_{i1}, p_i) as a point on the plane and $E_{p_i}(a, b)$ being the vertical distance between this point and the computed line.

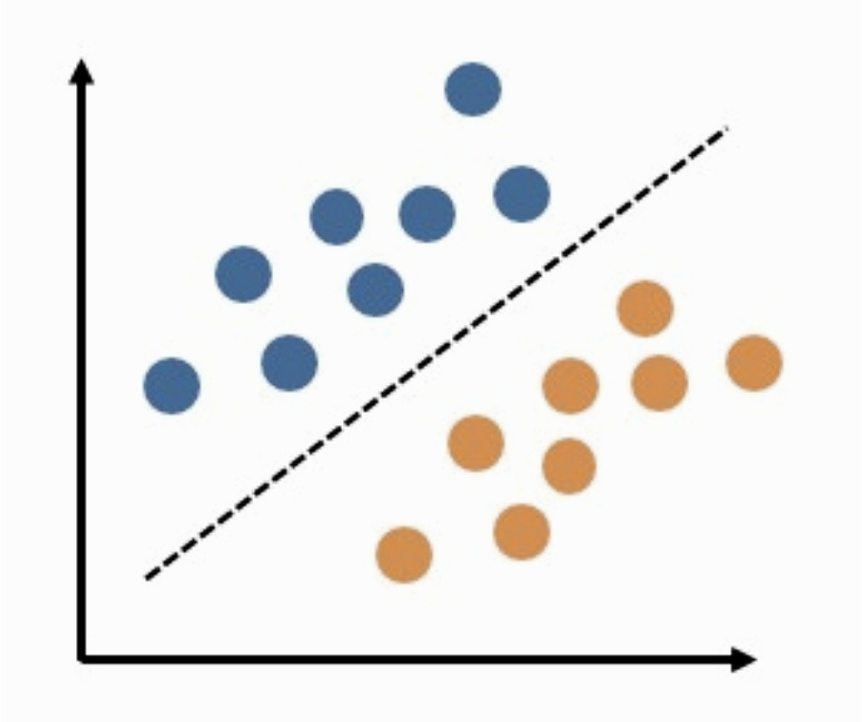
- Here, we consider the objective function of minimizing the sum of errors: $min_{(a,b)} \sum_{i=1}^{m} E_{p_i}(a, b)$
- Consider the problem of choosing a, b to minimize the objective function defined earlier. The problem is that this is not a linear program. The source of nonlinearity is the absolute value sign $||$. However, these absolute values can be made linear with a simple trick.

  The trick is to introduce extra variables $e_1, ..., e_m$, one data point. The intent is for e_i to take on the value $E_{p_i}(a, b)$ using the result $|x| = max(x, -x)$.
- Hence, we add two constraints for each data point:
  $e_i \geq [(\sum_{j=1}^{d} a_j p_{ij} - b) - l_i]$ and $e_i \geq -[(\sum_{j=1}^{d} a_j p_{ij} - b) - l_i]$

  We change the objective function to $min \sum_{i=1}^{m} e_i$ with decision variables $e_1, ..., e_m, a_1, ..., a_d, b$.
- The key point is: at an optimal solution to this linear program, e_i must be equal to $E_{p_i}(a, b)$ for every data point p_i.

  Feasibility of the solution already implies that $e_i \geq E_{p_i}(a, b)$ for every p_i. And if $e_i > E_{p_i}(a, b)$ for some p_i, then we can decrease e_i slightly so that the constraints still hold, to obtain a superior feasible solution.
- Hence, we conclude that an optimal solution to this linear program, we get the line minimizing the sum of errors.

Linear Classifiers



The problem

We are looking for a binary function (from $R^d \rightarrow 0, 1$). For example, data points could represent images, and we want to know which ones have a dog and which ones don't.
Formally, the input consists of m data points $p^1, ..., p^m \in R^d$ labeled 1 and m' data points $q^1, ..., q^m \in R^d$ labeled 0.

The goal is to compute a linear function $h(z) = \sum_{j=1} a_j z_j + b$ (from R^d to R), where z_k represents the k^{th} feature of z, such that $h(p^i) > 0$ and $h(q^i) < 0$.

Geometrically, we are looking for a hyperplane in R^d such that all positive points are on one side and all negative points on the other. Such a hyperplane can be used for predicting the labels of other unlabeled points by checking which side of the hyperplane it is on. If there is no such hyperplane, an algorithm should correctly report this fact.

Linear classifier as LP

The only issue with this problem is that the constraints are strict inequalities, which is not allowed in linear programs.

However, we can easily add an extra decision variable.

- The new decision variable δ represents the “margin” by which the hyperplane satisfies the constraints. So, we have a linear program:

  $max\ \delta$ subject to

  $\sum_{j=1}^{d} a_j p_j^i + b - \delta \geq 0$ for points $p^1, p^2, ...p^m$

  $\sum_{j=1}^{d} a_j p_j^i + b + \delta \leq 0$ for points $q^1, q^2, ...q^{m'}$

  which is a linear program with decision variables $\delta, a_1, ..., a_d, b$.

- If the optimal solution to this linear program has a strictly positive objective function value, then the values of the variables $a_1, ..., a_d, b$ define the desired separating hyperplane.
  not, then there is no such hyperplane.

- We conclude that computing a linear classifier reduces to linear programming.

The Minimax theorem for two-player zero-sum games

Some useful definitions

- **Game**: A game is defined as a conflict involving gains and losses between two or more opponents who need to follow some formal rules.

- **Strategy**: A strategy of a player is a complete plan determining which action he/she will take at each stage he is to move.
  The strategy is not the same as a move. A move is an action taken by a player at some point during the play of a game where strategy is a complete algorithm for playing the game, directing a player what to do for every possible situation throughout the game.
  Let us denote the set of strategies for a player i as $S_i = \{s_{i1}, s_{i2}, ..., s_{ik}\}$.
  For instance, in a rock-paper-scissors game, the set of strategies is {Rock, Paper, Scissors}.

- **Pure Strategy**: A pure strategy provides a complete definition of how a player will play a game. A player's strategy set is the set of pure strategies available to that player. We can hence say that a player plays with certainty using a pure strategy.
  Considering the definition of S_i above, the pure strategies are $s_{i1}, s_{i2}, ..., s_{ik}$.

- **Mixed Strategy**: A mixed strategy of a player is a probability distribution over the set of his/her strategies. This allows for a player to randomly select a pure strategy. We can hence say that a player randomizes his/her choices using a mixed strategy.
  Considering the set of strategies mentioned above,
  A mixed strategy σ_i for player i is a function on S_i such that $0 \leq \sigma_i(s_{ij}) \leq 1$ and $\sigma_i(s_{i1}) + \sigma_i(s_{i2}) + \cdots + \sigma_i(s_{ik}) = 1$.
  Here σ_i represents other players' views about which strategy i would play.
  A pure strategy can be thought of as a specific case of mixed strategy where one of the strategy s_{ij} has $\sigma_i(s_{ij}) = 1$ and $\sigma_i(s_{iw}) = 0$ for $w \neq j$.
- **Payoff Matrix**: An $m \times n$ matrix that gives the possible outcome of a two-person zero-sum game when player A has m possible moves and player B has n possible moves.

Zero-Sum Games

A **zero-sum game** is a game in which players make payments only to each other. In such a game, one player's loss is the other player's gain.

- A zero-sum game is specified by a real-valued matrix $m \times n$ payoff matrix A. Here, we consider the **two-player zero-sum game**.
- One player, the row player, picks a row. The other (column) player picks a column.
- Each row corresponds to a choice available for the row player and each columnn corresponds to a choice available for the column player.
- By definition, the entry a_{ij} of the matrix A is the row player's payoff when the person chooses row i and the column player chooses column j. The column player's payoff, in this case, is defined as $-a_{ij}$, hence the term "zero-sum".
- Hence, a_{ij} is the amount that the column player pays to the row player in the outcome (i, j).
- Thus, the row and column players prefer bigger and smaller numbers in the payoff matrix respectively.
- A **two-player game** is defined by four sets (X, Y, A, B) where

1. X and Y are the set of strategies of the first and second player, respectively.
2. A, B are real-valued functions defined on $X \times Y$.
   The game is played as follows:

- Player 1 chooses $x \in X$ and Player 2 chooses $y \in Y$ simultaneously, each unaware of the choice of the other.
- Then their choices are made known and Player 1 wins $A(x, y)$, and Player 2 wins $B(x, y)$. A and B are called utility functions for Players 1 and 2 respectively.
- A is the payoff matrix with Player 1 as the row player and B is the payoff matrix with Player 2 as the row player where the payoff matrix is as defined earlier.
- The goal of both players is to maximize their utility.

- In a two-player zero-sum game, $A = -B$.
  - Hence, we describe a **two-player zero-sum game** by three sets (X, Y, A).
- We can write the expected payoff of the row player when payoffs are given by an $m \times n$ matrix A, the row strategy is x (a distribution over rows), and the column strategy is y (a distribution over columns), as
  $\sum_{i=1}^{m} \sum_{j=1}^{n} P[outcome(i, j)] a_{ij} \quad (\text{definition of expectation})$
  $= \sum_{i=1}^{m} \sum_{j=1}^{n} P[\text{row i is chosen}].P[\text{column j is chosen}] a_{ij}$
  $(\text{since the row and column players randomize independently})$
  $= x^T A y$
  $(since\ P[\text{row i is chosen}]\ is\ x_i\ and\ P[\text{column j is chosen}]\ is\ y_j)$

In a two-player zero-sum game, would one prefer to commit to a mixed strategy before or after the other player commits to his/hers? Intuitively, there is only a first-mover disadvantage, since the second player can adapt to the first player's strategy. The minimax theorem, however, implies that it doesn't matter.

For example, the matrix A for rock-paper-scissors game is as follows:

$$\begin{bmatrix} & \text{Rock} & \text{Paper} & \text{Scissors} \\ \text{Rock} & 0 & 1 & -1 \\ \text{Paper} & -1 & 0 & 1 \\ \text{Scissors} & 1 & -1 & 0 \end{bmatrix}$$

The Minimax Theorem

Let A be an $m \times n$ matrix representing the payoff matrix for a two-person zero-sum game. Then the game has a value and there exists a pair of mixed strategies that are optimal for the two players.

In other words, for every two-person zero-sum game (X, Y, A) there is a mixed strategy x^* for player 1 and a mixed strategy y^* for player 2 such that,

$max_x min_y x^T Ay = min_y max_x x^T Ay = x^{*T} Ay^*$

LP Duality and Minimax

We give a **proof of the minimax theorem** as follows:
We take the matrix A the payoff matrix as described above. Let $r_1, r_2, ...r_m$ and $c_1, c_2, ...c_n$ be the rows and columns of the matrix A respectively.
Firstly, we observe that for a vector x,
$min_y\ x^T Ay = min_j\ x^T A1_j = min_j < x, c_j >$ where $j \in \{1, 2, ...n\}$ and 1_j is where 1_j is the j^{th} standard basis vector, corresponding to the column player choosing column j.
because Ay is a distribution over $r_1, ..., r_m$. Taking the maximum over all distributions x, we have
$max_x min_y x^T Ay = max_x min_j < x, c_j >$ where $j \in \{1, 2, ...n\}$.
Similarly,
$min_y max_x x^T Ay = min_y max_i < r_i, y >$ where $i \in \{1, 2, ...m\}$.

Now, we see that $max_x min_y\ x^T Ay$ is equivalent to the following LP program taking $v = min_y x^T Ay = min_j < x, c_j >$ where $j \in \{1, 2, ...n\}$:
$max\ v$
s.t.
$< x, c_j > \ \geq v \quad \forall\ 1 \leq j \leq n$
$\sum_{i=1}^{m} x_i = 1$

$x_i \geq 0 \ \ \forall \ 1 \leq i \leq m$

Rewriting the above, we get
$v - < x, c_j > \ \leq 0 \quad \forall \ 1 \leq j \leq n$
$\sum_{i=1}^{m} x_i = 1$
$x_i \geq 0 \quad \forall \ 1 \leq i \leq m$

We see that the first constraint forces v to be at most $min \ < x, c_j >$.
We claim that if (v^*, x^*) is an optimal solution, $v^* = min \ < x^*, c_j >$.

- This can be seen from the fact that as noted earlier, v^* *can be at most* $min \ < x, c_j >$. *And, if* v^* is less than $min \ < x, c_j >$, by keeping x_i's constant, it is possible to slightly increase v^* *without contradicting feasibility which would hence contradict optimality.*
  Therefore, $v^* = min \ < x^*, c_j >$.

Since v^* maximizes v over all distributions x, $v^* = max_x min_j < x, c_j > = max_x min_y \ x^T A y$

Similarly, we can write the linear program for the column player as follows,

$min \ w$
s.t.
$w - < r_i, y > \ \geq 0 \ \ \forall \ 1 \leq i \leq m$
$\sum_{j=1}^{n} y_j = 1$
$y_j \geq 0 \ \ \forall \ 1 \leq j \leq n$ and the optimal
$w^* = min_y max_i < r_i, y > = min_y max_x \ x^T A y$
Notice that these two linear programs are duals of each other which can be seen easily from the way we take dual of an LP. By strong duality, the optimal values $v^* = w^*$ which proves the minimax theorem.

Linear Programming Algorithms

Simplex Method Revisited

- **Worst-Case Running Time**
  The simplex method is **very fast in practice**, and routinely solves linear programs with hundreds of thousands of variables and constraints. However, the **worst-case running time** of the simplex method is **exponential in the input size**.

  We see that the number of vertices of a feasible region can be exponential in the dimension (e.g., 2^n vertices of the n-dimensional hypercube). However, it is very hard to construct a linear program where the simplex method actually visits all of the vertices of the feasible region.

- **Average-Case and Smoothed Running Time**
  It has been shown that the simplex method (with a suitable pivot rule) runs in polynomial time "on average" with respect to various distributions over linear programs.

  The simplex method has been proved to have polynomial "smoothed complexity". Smoothed analysis is a hybrid of worst-case and average-case analyses. It measures the expected performance of algorithms under slight random perturbations of worst-case inputs.

  The main result is that, for every initial linear program, in expectation over the perturbed version of the linear program, the running time of simplex is polynomial in the input size.

  On the whole, bad examples for the simplex method are rare and hence simplex method remains the most commonly used linear programming algorithm in practice.

Ellipsoid Method

- **Worst-Case Running Time**
  The ellipsoid method was originally proposed as an algorithm for non-linear programming. Later it was proved that for linear programs, the algorithm is actually guaranteed to run in polynomial time. This was the **first polynomial-time algorithm for linear programming**.

  The ellipsoid method is **very slow in practice** — usually multiple orders of magnitude slower than the fastest methods. But, how can a polynomial-time

algorithm be so much worse than the exponential-time simplex method? There are two reasons behind that.

- The **degree in the polynomial** bounding the ellipsoid method's running time is **huge** (like 4 or 5, depending on the implementation).
- The **performance** of the ellipsoid method **on "typical cases"** is generally **close to its worst-case performance**. This is in sharp contrast to the simplex method, which almost always solves linear programs in time far less than its worst-case (exponential) running time.

- **Separation Oracles**

  The ellipsoid method is useful for proving theorems, especially for establishing that other problems are worst-case polynomial-time solvable, and thus are at least efficiently solvable in principle.

  The ellipsoid method can solve some linear programs with n variables and an exponential (in n) number of constraints in time polynomial in n.

  - But, doesn't it take exponential time just to read all of the constraints? For other linear programming algorithms, yes but not for the ellipsoid method as it doesn't need an explicit description of the linear program. **It just needs is a helper subroutine** known as a **separation oracle**.

  The responsibility of a separation oracle is to take as input an allegedly feasible solution x to a linear program, and to either verify feasibility (if feasible) or produce a constraint violated by x (otherwise). Of course, the separation oracle should also run in polynomial time.

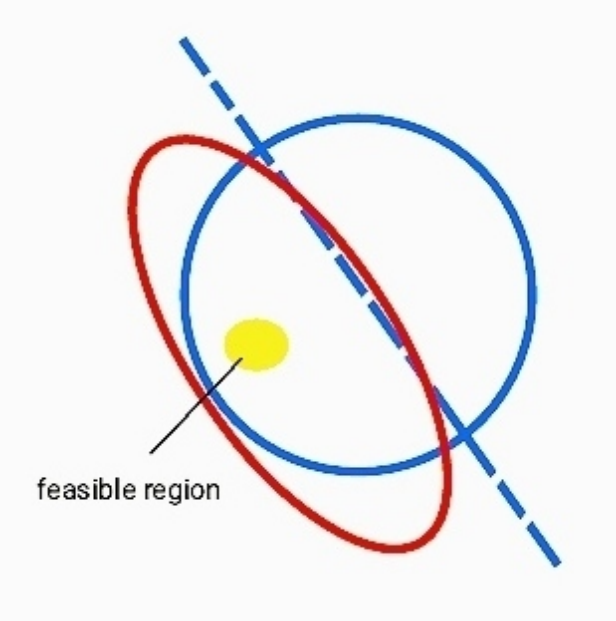
- **An Example:** How could one possibly check an exponential number of constraints in polynomial time?
  We have actually already seen some examples of this.

  - For example, we have seen **the dual of the path-based linear programming formulation of the max-flow problem**:
  - Minimise $c^T l$ subject to $\sum_{e \in p} l_e \geq 1 \; \forall p \in P$ and $l_e \geq 0 \; \forall e \in E$
    Here, P denotes the set of s-t flow paths in the flow network (with edge capacities c_e).

- Since a graph can have an exponential number of s-t paths, this linear program has a potentially exponential number of constraints. But, it has a polynomial-time separation oracle. The key observation is: at least one constraint is violated iff
  $min_{p \in P} \sum_{e \in p} l_e < 1$
- Thus, the separation oracle is just Dijkstra's algorithm in this case. Notice that the above condition is equivalent to the following:
- Given an allegedly feasible solution l_e where $e \in E$ to the linear program, the separation oracle checks if each l_e is non-negative (if $l_e < 0$, it returns the violated constraint $l_e \geq 0$).
- If the solution passes this test, then the separation oracle runs Dijkstra's algorithm to compute the shortest s-t path, using the l_e 's as (non-negative) edge lengths.
- If the shortest path has a length of at least 1, then all of the constraints are satisfied and the oracle reports "feasible".
  If the shortest path p^* has a length less than 1, then it returns the violated constraint $\sum_{e \in p^*} l_e \geq 1$.
- Thus, we can solve the above linear program in polynomial time using the ellipsoid method.

- **How the Ellipsoid Method Works**
  The first step in the ellipsoid method is to reduce the optimization problem to a feasibility problem. Basically, if the objective is max $c^T x$, we replace the objective function by the constraint $c^T x \geq k$ for some target objective function value k.
  If we can solve this feasibility problem in polynomial time, then we solve the original optimization problem using binary search on the target objective function value k.

  The algorithm:



- The ellipsoid method maintains at all times an ellipsoid which is guaranteed to contain the entire feasible region.
- It is initialised with a huge sphere to ensure the invariant at initialization.
- It then invokes the separation oracle on the center of the current ellipsoid.
- If the ellipsoid center is feasible, then the problem is solved.
- If not, the separation oracle produces a constraint satisfied by all feasible points that is violated by the ellipsoid center.
- Geometrically, the feasible region and the ellipsoid center fall on opposite sides of the corresponding halfspace boundary.
- Thus we know we can recurse on the appropriate half-ellipsoid.
- Before recursing, however, the ellipsoid method recreates a new ellipsoid that contains this half-ellipsoid which is now guaranteed to contain the entire feasible region.
- It can be shown that the volume of the current ellipsoid is guaranteed to reduce at a certain rate after each iteration, and this yields a polynomial bound on the number of iterations required.
- The algorithm stops when the current ellipsoid is so small that it cannot possibly contain a feasible point (given the precision of the input data).

- Thus we see how this method solves linear programs even with an exponential number of constraints. It never works with an explicit description of the constraints. It just generates constraints with every iteration. Since it terminates in a polynomial number of iterations, it generates only a polynomial number of constraints.

Interior Point Methods

Till now we have seen the simplex and the ellipsoid methods. The simplex method works “along the boundary” of the feasible region, and the ellipsoid method works “outside in”. Interior point methods work “inside out”.
There are many genres of interior-point methods, beginning with Karmarkar’s algorithm.

- The main idea is, instead of maximizing the given objective $c^T x$, we maximize $c^T x - \lambda \cdot f(\text{distance between x and boundary})$ where $\lambda \geq 0$ is a parameter and f is a “barrier function”.
- In general, a barrier function is a continuous function whose value on a point increases to infinity as the point approaches the boundary of the feasible region.
  Here, the barrier function f goes to $+\infty$ as the distance between x and boundary goes to 0 (e.g., $log(\frac{1}{z})$).
- We initialize with a large value of λ so as to simplify the problem. For example, when $f(z) = log(\frac{1}{z})$, the solution is the center of the feasible region, and can be computed using methods like the Newton’s method.
- Then we gradually decrease the value of λ and track the corresponding optimal point along the way. When $\lambda = 0$, the optimal point is an optimal solution to the linear program, as required.

Interior-point methods consist of many algorithms that run in polynomial time in the worst case many interior-point methods also compete with the simplex method in practice. For example, one of Matlab’s LP solvers uses an interior-point algorithm

---

Bibliography

https://www.coursera.org/learn/discrete-optimization/home/welcome

http://timroughgarden.org/w16/w16.html

https://www.whitman.edu/Documents/Academics/Mathematics/lewis.pdf

http://algorithmics.lsi.upc.edu/docs/Dasgupta-Papadimitriou-Vazirani.pdf

http://www.jcreview.com/fulltext/197-1592807730.pdf