CVX

Some problems where the region doesn't have to be a polyhedron, and the optimal solution doesn't have to be on an extreme point, simplex won't work.

$\min_x f(x)$	convex	Assume the d…
s.t. $g(x) \le 0$	convex	State the requ… form in order t… convex
$\max_x f(x)$	concave	
s.t. $g(x) \le 0$	convex	The objective … a convex feasi…
$\min_x f(x)$	convex	
s.t. $h(x) = 0$	convex and concave: affine! to mimic inequality, can be	$Ax \sim$
$x \ge 0$		
$\min_x f(x)$	convex	
s.t. $g(x) \ge 0$	concave	
$h(x) = 0$	affine	

A function f is convex if, for any two points x and y, the line segment between these points lies above the function graph. Mathematically:

$$f(\theta x + (1-\theta)y) \le \theta f(x) + (1-\theta) f(y),$$

where $\theta \in [0,1]$. This definition is central for proving that the solution is globally optimal in convex problems.

Rates of Convergence

$k \le O\left(\frac{1}{\epsilon}\right)$	$\epsilon \le O\left(\frac{1}{k}\right)$	sublinear
$k \le O\left(\frac{1}{\sqrt{\epsilon}}\right)$	$\epsilon \le O\left(\frac{1}{k^2}\right)$	sublinear
$k \le O\left(\log \frac{1}{\epsilon}\right)$	$\epsilon \le O\left(e^{-k}\right)$	linear
$k \le O\left(\log_2 \log \frac{1}{\epsilon}\right)$	$\epsilon \le O\left(e^{-2^k}\right)$	quadratic

As a wild generalisation (depends on exact problem class and algorithm):

- *First-order methods* converge either sublinearly or linearly
- *Second-order methods* converge superlinearly or quadratically

Information-Based Complexity measures the number of iterations required to achieve a solution within a specified tolerance. This is a practical metric for understanding how efficiently an algorithm can approximate the optimal value.

The **Jacobian matrix** generalizes the gradient for vector-valued functions, containing partial derivatives of each output component with respect to each input variable. Every Row is the transpose of the derivative of that function f1 with respect to xi.

The **Hessian matrix** provides a second-order derivative for scalar functions, showing curvature. Convex iff function lies on or above all of its tangents, explained below. Single variate – only second derivative, multi variate – strictly convex, Hessian matrix is **positive definite** everywhere. Definiteness of Hessian at a stationary point indicates whether or not it is a minimum (pos definite), maximum (neg definite) or saddle point (indefinite).

Local Minimum Condition: In convex functions, if x* is a local minimum, then it is also a global minimum. If the function is convex, first order condition becomes a **sufficient** condition for global minimum, if not then hessian being positive semidefinite is necessary.

In multi-variable constraints, visualizing sublevel (f(x)≤c) and super-level (g(x)≥c) sets helps in understanding the feasible region. For convex functions, sublevel sets are convex, which supports the optimization process by maintaining a convex feasible region.

Equality and inequality constraints together define what is known as the feasible set. Latter divide the region as represented by a hyperplane, with a positive/negative half space or null region i.e. the hyperplane itself. Large family of optimization problems, mainly convex optimization problems can be solved efficiently in a unified manner.

Polyhedral, which is intersection of half spaces, is a convex set. Region above the graph is called epigraph, and epi of a convex function is convex (hypo for concave). Scalar multi or addition are convex. An optimization problem is convex if the objective function and constraints are convex or linear. (obj being lin and constraint being convex counts). If f is convex but g is concave, problem can still be convex.

Principle of duality – Take a hyperplane that supports my set in such a way that the set falls entirely on the positive side of this hyperplane. That means a set is convex when you recover the set in intersection of all the positive sides of all the hyperplanes.

Extra - While a twice-differentiable function with a positive definite Hessian everywhere is strictly convex (e) and will have at most one stationary point (a), which is a global minimum (b), it does not guarantee that Newton's method will converge in one step. Newton's method converges in one step only if the function is quadratic. For non-quadratic functions, it typically requires multiple iterations to converge.

Interior Point Method – Way to solve KKT conditions. Interior Point Methods are optimization techniques used to solve linear and nonlinear constrained problems. They work by keeping the solution iteratively "inside" the feasible region and gradually moving towards the boundary, as opposed to boundary methods like Simplex. Interior Point Methods avoid directly hitting constraint boundaries by using a "barrier" approach, which penalizes the solution as it nears the boundary, making it computationally efficient.

We are going to solve a modified perturbed version called KKT(t) (which are much easier to solve. where t is +ve parameter which controls the degree of perturbation. As this parameter goes to 0, x(t) converges to x*. u = -t/gx. Plugging into the gradient equation, we get -t log, which acts as a barrier function at the objective, approaching infinity as the variables approach the boundary of a constraint, discouraging any further movement in that direction, smoothing out the sharp transition of a constraint. The **barrier method** (interior point method) solves a sequence of barrier problems with t converging to zero.

$$\nabla(\,f(x) - t\log(\,-g(x))) = 0$$

We can now perform newton's method to perform this minimisation. Initially, don't t to be too small otherwise it will result in a discontinuous function with the log, and newton would be too slow to converge, so start with a big t, meaning f would be overshadowed, and only g would need to be minimised. X(t) would be the analytic center of the feasible region. Once we have a solution, try for a smaller t, i.e. t' = 0.9t, apply newton again, but x(t) would be the starting point, so computing x(t') would be faster. Working towards t = 0 formed by the path of xs of t, called the central path (trajectory that solutions follow in the interior point method as they approach the optimal solution), falling in the interior of the feasible region.

KKT x^*		KKT(t) $x(t)$
$\nabla f(x) + u\,\nabla g(x) = 0$		$\nabla f(x) + u\,\nabla g(x) = 0$
$g(x) \le 0$		$g(x) \le 0$
$u \ge 0$	$\leftarrow$ $t \longrightarrow 0$	$u \ge 0$
$u\,g(x) = 0$		$u\,g(x) = -t$

Second order methods can often achieve higher rates of convergence over first order methods. However, this comes at the cost of more computation per iteration, e.g., in calculating Hessians and solving a large linear system of equations. First order methods might be necessary for applications where the problem is very large, where the Hessian may struggle to fit into memory.

To determine if the function $f(x,y) = x^2 + xy - 2$ is convex, we use the **Hessian matrix** approach. We start by calculating the first partial derivatives of f:

$$f_x = 2x + y \quad \text{and} \quad f_y = x$$

Next, we compute the second partial derivatives to form the Hessian matrix H:

$$H = \begin{bmatrix} f_{xx} & f_{xy} \\ f_{yx} & f_{yy} \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 1 & 0 \end{bmatrix}$$

To assess the definiteness of H, we find its eigenvalues by solving the characteristic polynomial:

$$\det(H - \lambda I) = -\lambda^2 + 2\lambda - 1 = 0$$

For a function $f : \mathbb{R}^n \longrightarrow \mathbb{R}^m$, state a condition that f needs to satisfy in order to be a convex function if f is not differentiable. State a different condition for the case where f is differentiable.

For the first part you could use: the line segments between any two points on graph of f, must be above or equal to f. A second option would be to state that the epigraph of f must be a convex set.

When f is differentiable, different conditions from the above are: the Hessian of f must be positive semidefinite, or the graph of f must lie on or above all its tangent planes.

Using μ_1, μ_2, μ_3 for the inequality constraints above respectively. The stationarity condition consists of the following two equations:

$$-\mu_1 \frac{1}{x^2} + 4\mu_2 x - \mu_3 = 0$$
$$2(y+2) - \mu_1 = 0$$

The constraints are:

$$\frac{1}{1.5811} - 0.6325 = 0.000$$
$$2 \cdot 1.5811^2 - 5 = 0.000$$
$$1 - 1.5811 = -0.5811$$

$$x^{(k)} = \arg\min_x f(x) + \eta_k \|\max(0, g(x))\|_2^2 + \eta_k \|h(x)\|_2^2$$

$$\min_{x \in \mathbb{R}^n} f(x) \quad \text{s.t. } g(x) \le 0$$

The Penalty Method turns a constrained optimisation problem into an unconstrained one, for a penalty parameter η_k, by applying a quadratic penalty to constraint violations:

$$\min_{x \in \mathbb{R}^n} f(x) + \eta_k \sum_i^m \max(0, g_i(x))^2$$

A sequence of the above penalty problems are solved for increasing η_k, until the constraint violations are within some tolerance.

All constraints are satisfied (**primal feasibility**). The last constraint is not active, so due to the complementary slackness condition, its dual variable must be zero $\mu_3 = 0$ (**complementary slackness**). Plugging in the candidate point to the complementary slackness terms we get the following expressions for the other two duals:

$$\mu_2 = 2(y+2)\frac{1}{4x^3} = 0.333$$
$$\mu_1 = 2(y+2) = 5.265$$

All duals are positive (**dual feasibility**) and the stationarity condition is satisfied with these dual values (**stationarity**).

The problem is convex and all functions are smooth over the domain of interest, so the KKT conditions are sufficient for a global optimal solution.

Battery efficiency, denoted by **η** (eta), measures the fraction of energy effectively stored or discharged. For charging, the internal power **b̃** is less than or equal to the external power multiplied by the efficiency: **b̃ ≤ η * b** For discharging, the reverse applies (due to efficiency losses): **b̃ ≤ b / η. Roundtrip Efficiency**: the efficiency over a full cycle of charging and discharging is calculated as the square of η: **Roundtrip efficiency = η^2** For example, if η (efficiency) is 90%, then the roundtrip efficiency is 0.9 * 0.9 = 0.81, or 81%. **Non-Const Efficiency:** By adding constraints at multiple points along the efficiency curve, the model can approximate the true behaviour of the battery's efficiency more accurately, without resorting to complex nonlinear equations. **The optimality gap** is the difference between the upper and lower bounds of the solution range in a minimization problem. It indicates higher confidence i.e. how close the solution is to the true optimum. The optimality gap for a maximisation is the difference between the smallest upper bound and the largest found feasible solution, inverse for minimisation.

Dual: Scaling factor for each primal constraint. The dual problem aims to find the largest lower bound for the primal objective, constrained by inequalities that preserve the primal-dual relationship.

Rules for Getting the Dual

Dual of the dual of an LP is the primal.

The dual of a **min**imisation problem:		The dual of a **max**imisation problem:	
$\min_x c^\intercal x$	$\max_y b^\intercal y$	$\max_x c^\intercal x$	$\min_y b^\intercal y$
$x \ge 0$	$A^\intercal y \le c$ (swap ineq.)	$x \ge 0$	$A^\intercal y \ge c$ (same ineq.)
$x \le 0$	$A^\intercal y \ge c$	$x \le 0$	$A^\intercal y \le c$
$x \in \mathbb{R}^n$	$A^\intercal y = c$	$x \in \mathbb{R}^n$	$A^\intercal y = c$
$Ax \le b$	$y \le 0$ (same ineq.)	$Ax \le b$	$y \ge 0$ (swap ineq.)
$Ax \ge b$	$y \ge 0$	$Ax \ge b$	$y \le 0$
$Ax = b$	$y \in \mathbb{R}^m$	$Ax = b$	$y \in \mathbb{R}^m$

Or remember one side, and convert max to min or min to max before taking dual

When a constraint is non-binding, its dual variable is zero. Constraints with higher dual variable values have a stronger impact on the objective, indicating where resources are most valuable.

$$\min_{x,y,z} x + 2y - 5z \quad \text{s.t. } x + 2y + 3z \ge 2,\; y \ge x + 2z,\; z \le 6$$

Convert the above problem to an equality and non-negative variable form (only equality constraints and non-negative variables). Implement the transformed form in PuLP and verify you get the same solution.

See transform.py.

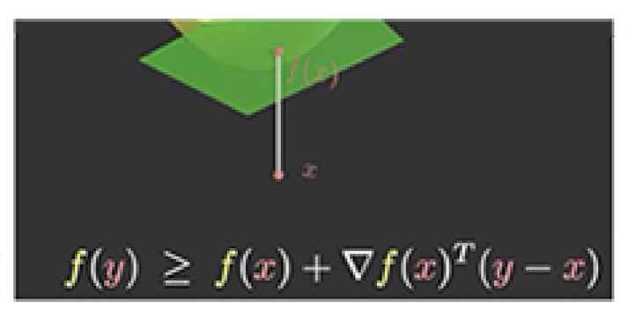
$$\min\; x^+ - x^- + 2y^+ - 2y^- - 5z^+ + 5z^-$$
s.t. $x^+ - x^- + 2y^+ - 2y^- + 3z^+ - 3z^- - s_1 = 2$
$y^+ - y^- - s_2 = x^+ - x^- + 2z^+ - 2z^-$
$z^+ - z^- + s_3 = 6$
$x^+, x^-, y^+, y^-, z^+, z^-, s_1, s_2, s_3 \ge 0$

How a starting feasible solution is obtained for Simplex: *Alternative simplex*: **Create a Feasibility Problem**: This involves modifying the original problem by adding slack variables to relax the constraints. **Objective**: Minimize these slack variables to bring the system into feasibility. If the optimal solution to this feasibility problem has any non-zero slack variables, it indicates that the original problem is infeasible. **Finding sv:** The feasibility problem allows for a straightforward starting point by solving for a trivial starting vertex. *Standard/Revised Simplex*: **Convert to Standard Form**: Transform the problem into equality constraints with non-negative variables. **Introduce Slack Variables**: Add non-negative slack variables to handle equality csrts. The goal is to minimize the sum of the slack variables. Start with a BFS by setting all original variables to zero and using the slack variables in the basis. If all slack variables are zero at the optimal solution, the original problem is feasible, and this solution serves as the starting BFS.

Pivoting in alternative simplex method, when a constraint is selected to leave the active set, the algorithm aims to move along the edge defined by the intersection of the remaining n−1 active constraints. As we progress along this edge, the constraint that will enter the active set is the first one encountered. Determining this requires factoring in the alignment between the edge direction and the new constraint, as well as considering the slack available for the movement along this edge. There are two special cases to consider. First, if it's possible to move indefinitely along the edge without encountering a new constraint, the problem is deemed unbounded. Second, if multiple cstr are reached simultaneously, a pivot selection rule is employed to decide which constraint should enter the active set.

The Simplex algorithm typically requires a **basic feasible solution** as a starting point. A basic feasible solution is an extreme point of the feasible region. Since **Alpha-Feas** only guarantees that xα is feasible (it satisfies Ax≤b) but doesn't guarantee it's a basic feasible solution, xα may not be suitable as a starting point for the Simplex algorithm, and it's not guaranteed it would be an extreme point either.

To solve this optimization problem, we begin with the conditions that x−7y≥0, x≥40, and y≥10, aiming to minimize x+5y. Dual objective function is max 40y1 + 10y2 + 0y3 with constraints y1 + y3 <= 1 and y2 - 7y3 <= 5. By simplex, substituting into the objective function, it simplifies to 120 - 70s1 - 10s2 - 30y1, so y1=0, maximize the other two.



$$f(y) \ge f(x) + \nabla f(x)^T (y - x)$$

Tangent Hyperplane is a good approximation of the graph of f at point x. Dual def of convex fn, f is convex if and only if it's graph is above its tangent hyperplane. So, for gradient 0, the hyperplane is horizontal – so global minimizer. So gradient 0 and solve for x. So non constant linear functions are unbounded. There can be more complex problems, to even solve the gradient, so there are other methods.

Gradient descent – we can evaluate the cost function at an arbitrary point, and with a process called back propagation, we evaluate the negative gradient and take a small step in that direction, doing it iteratively (in stochastic, we can take small random subset at each iteration, in adaptive, picks a different step size for each component).

$$\theta^{k+1} = \theta^k - \eta \nabla f(\theta^k)$$

When you minimize a function, what you are looking for critical points of that function, where derivate equals zero.

How step size is selected is called stepped, like fixed or rule based(might use info from previous iterations). In line based, updates based on gathering information by looking at previously un-explored points.

Line Search

Solve problem:

$$\alpha_k := \arg\min_\alpha f(x_k + \alpha d_k)$$

optimising a single variable (moving in a straight line from our current iterate)

where for gradient descent: $d_k = -\nabla f(x_k)$ (our search direction)

Either exactly, which might take time, or inexactly using an approach such as backtracking line search.

We might not want to spend too much time doing this exactly because our current search direction might not be that great to start with!

Backtracking line search:

1. Start with a large step estimate.
2. Iteratively shrink the step size until some conditions are met: i.e. the objective value has "adequately" improved.
3. Continue onto calculating a new search direction.

Newton's method-Hard to plot functions that have more than 2 variables(maybe even million), iterative optimisation algo, start with intial guess x0(unknown, draw from some random distraubution), and the closer the intial guess is to the true minimiser the better. We pick a direction d0(descent direction) and we follow it to get x1, track f(x1) for progress in decline. How to pick good descent direction – gradient gives us direction of biggest increase. $X_{k+1} = x_k + d_k$.

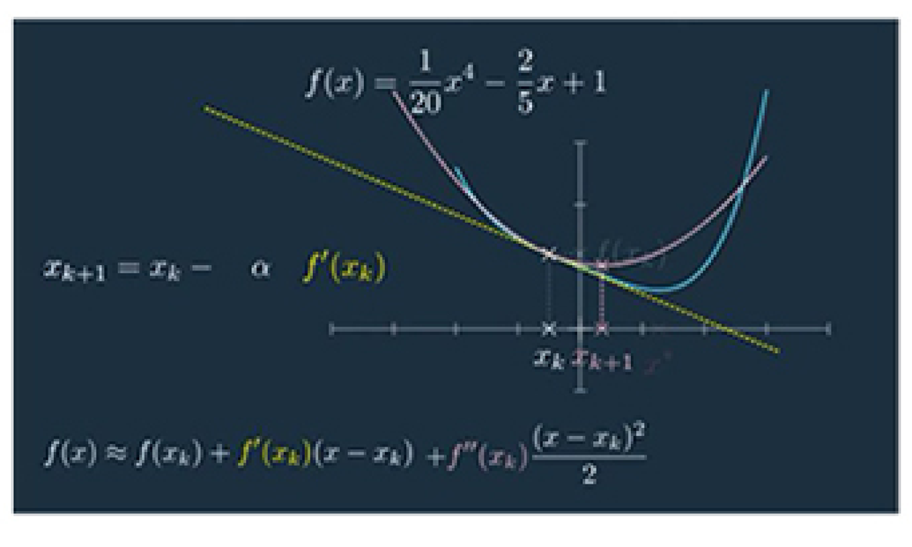
Newton's method is commonly used for **finding the zeros** of non-linear functions. Start with Taylor series of a **multivariate scalar function.** At some iteration x_k, when x tends towards x_k, the rate of change of the function x around the point x_k.

$$f'(x_k) \approx \frac{f(x) - f(x_k)}{x - x_k}$$

Rearranging terms, we see that the derivative is the slope of the tangent line to the graph of f at x_k. If the slope is positive, go as much as left in the tangent line, vice versa for negative. Both cases, $X_{k+1} = x_k$ - alpha(step size) * f'(x_k), subtracting a positive multiple of the derivative slope from the current iterate. We can't make alpha too large, we might overshoot and miss the minimizer, too small and it might take a lot more iterations, trying to tune it in a optimal way is what leads us to newton's method. From taylor series, we can get an even better approximation by including the second derivative(which is quadratic), and take the minimizer of this quadratic function as our next iterate. So alpha is now inverse of second derivative by algebra, and we don't have to pick alpha manually anymore, making it second order method from using second derivative information. For multivariate, replace the first derivative with the gradient and the second derivative with the inverse of the hessian. A hessian is only invertable if it is positive definite, i.e. all it's eigen values are positive HV = lamda v.

$$f(x) \approx f(x_k) + f'(x_k)(x - x_k)$$

3. **When the Function Is Exactly Quadratic:**
   - If $f(x)$ is quadratic, such as $f(x) = \frac{1}{2}x^TQx + c^Tx$, where Q is **positive definite**, then Newton's Method will converge to the minimum in **one step**.
   - In this case:
     - The gradient $\nabla f(x) = Qx + c$.
     - The Hessian $H_f(x) = Q$ (a constant matrix).
4. **Direct Solution for x When $f(x)$ Is Quadratic:**
   - Applying Newton's formula: $H_f(x_k)^{-1}\nabla f(x_k) = Q^{-1}(Qx_k + c) = x_k + Q^{-1}c$
     - Solving this directly moves x_{k+1} to the exact minimum point: $x_{k+1} = -Q^{-1}c$
   - Therefore, if $f(x)$ is truly quadratic, **Newton's Method reaches the minimum in a single step.**



LP

Decision Vars: These are variables we control, and they directly influence the solution, **Parameters***:* Constants in the problem that affect outcomes but aren't controlled directly. **Constraints:** Define the limits on decision variables, relationships between variables, and relationships between variables and parameters. **Objective:** Defines what makes a solution "good" by establishing a goal—either maximizing or minimizing a criterion. Model Based Adv: Flexibility: Formulate once and reuse, Automate, easy to update, Disadv: slower than tailored algo, limited control over soln procedures. Algo - problem-specific and often rely on heuristics, which are simplified rules to guide solutions – heuristics and meta heuristics, limited adaptability (new cstr often require rewriting or adjustm).

Leonid Kantorovich39, Defines an LP as a constrained optimization problem with a linear objective function and linear constraints with real valued continuous variables. Can be solved in polynomial time, handling millions of variables and constraints, cannot represent combinatorial choices directly, but often provide proximations.

PuLP is a Python library for LP and MILP, linking with solvers like Clp and Cbc. **LpVariable**: Creates decision variables. **LpProblem**: Sets up the optimization problem i.e. model m. Constraints are added to the model, and then the objective. Status (infeasible as -1, often due to conflicting requirements or overly restrictive constraints), var values, objective value can be printed out. PULP_CBC_CMD, msg = 1 to make the solver more verbose.

Active Constraints (or binding constraints) are those that are satisfied as equalities in the optimal solution, meaning they limit the feasible solution directly. **Contract Modification –** Removing a pesky binding constraint by adding a penalty for breaking the contract. If yields higher profit, financially favourable to break the contract.

Negative of obj turns into to min/max problem. Standard form of LP requires **equality constraints** and **non-negative variables,** former is done so using slack variables. Sl vars are non-negative basic variables to emulate an equality and is free to vary. They are used as loose variables which are tightened to loosen a decision var in the obj. **Feasible Region**: The total set of all feasible solutions, also known as the solution space.

The definition of convexity:

$$\forall x, y \in A: \quad \{x + \alpha(y - x) | \alpha \in [0,1]\} \subseteq A$$
$$\forall x, y \in B: \quad \{x + \alpha(y - x) | \alpha \in [0,1]\} \subseteq B$$

Any points in both A and B will also have their line segment in A and B, and so it will also be in the intersection of A and B:

$$\forall x, y \in A \cap B: \quad \{x + \alpha(y - x) | \alpha \in [0,1]\} \subseteq A \cap B$$

The **convex hull** of a set of points is the smallest convex region containing all those points, often represents the feasible boundary in LP, helping to define the area within which all potential solutions exist. An **extreme point** of a convex set S does not belong to an open line segment for **any** two points in S.

Unbounded Solution: When the feasible region extends indefinitely, leading to potentially infinite values for the objective. **Single Optimal Solution**: Occurs when the objective function touches the feasible region at a single point, like a vertex. **Multiple Optimal Solutions**: Found along an edge or face where the objective function aligns (slope of obj = slope of the edge). **Infeasible Solution**: If the feasible region is empty, no solution exists. For obj c^Tx, for minimization, we move in the opposite direction of c.

Since the objective function is linear, it defines a direction in which its value increases/decreases. To solve the problem graphically, all we must do in this direction and record the last intersection, which is the optimum, which will be achieved in at least one vertex. We can move from vertex to vertex, which is called pivoting, picking one that brings us closer to the goal.

The **Simplex algorithm** is a key method for efficiently solving LP problems by systematically navigating the vertices of the feasible region to locate the optimal solution (not polynomial time). The crucial idea about simplex algorithm: **Choose the Entering Variable**: Look at the objective function row and pick the variable with the largest positive coefficient. This choice maximizes the increase in the objective. **Determine How Far to Move**: For each constraint involving this variable, calculate how far you can go before hitting a limit. **Select the Exiting Variable**: The slack variable to tighten, based on the largest non-positive ratio between the entering variable and the constants. **Keep doing this all the coefficients in the objective fn are negative**. Opposite for minimisation problem.

Standard Simplex: Uses a tableau form, which is accessible but best suited for smaller examples.

MIP

MIP combines the flexibility of linear programming (LP) with integer constraints, allowing us to solve more complex problems that require some decisions to be whole numbers (like assigning people or resources in fixed units). The Knapsack Problem is a MILP prob involving selecting items with given values and weights to maximize total value within a weight limit, with a binary variable attached representing whether an item is chosen (1) or left out (0).

Linear Relaxation: Eases integer constraints to create a simpler LP model, since ILP is NP-Hard while LP is P. **Convex Hull**: The convex hull of a MILP is the smallest convex set that includes all integer feasible points, forming a convex polyhedron where all vertices correspond to integer feasible points. The strength of a formulation depends on how closely it approximates the **convex hull** of feasible integer solutions. If the linear relaxation perfectly matches the convex hull, any LP solution would also solve the ILP optimally. In other words, a strong convex hull formulation provides a tight relaxation, meaning the feasible region of the relaxed LP is close to the integer solution space. This alignment reduces the search area for integer solutions, improving efficiency. Formulations will typically have to weaken the more general they become. Unfort, the **convex hull changes** not just as the class of problem changes, but also **as parameter values change.**

Types of Cstrs**: Choice constraints** often require binary decisions, where a variable is set to either 1 (choice made) or 0 (choice not made) eg, In the knapsack problem, each item has a binary choice constraint. **Logical constraints** are used to express dependencies or conditions between variables eg: z implies x v y. **Disjunctive constraints** allow one of multiple constraints to hold, which is valuable for modelling choices where only one of several conditions can apply. **Piecewise Linear Constraints:** Useful for approximating nonlinear relationships by breaking them into linear segments. **On-off constraints** control whether certain activities or resources are active (on) or inactive (off) based on decision variables i.e. for u=1, x=0 as in low(x)(1-u) ≤ x ≤high(x)(1-u). The **Big-M** method provides a large constant (M) to enforce conditions indirectly by linking variables. If M is too large, it can worsen the relaxation quality, making solutions less efficient. In cases where we can't easily derive bounds, M is used i.e. x E (-M, M).

Nonlinear relationships (like squared or mod terms) are harder to model directly, so using auxiliary variables to approximate these functions, transforming nonlinear constraints into piecewise linear or other ILP-friendly forms.

Branch & Bound – B&B is a method for systematically exploring the solution space. It divides (or "branches") the problem into subproblems and calculates bounds to eliminate (or "prune") sections that cannot contain the optimal solution. For Milp where dec var are integers, simplex won't help to find a soln. Let's say for a linear relaxation, we calculate the obj value, and the exact values, but they are no longer feasible if they are fractional. Starting the branch and bound with the soln (i.e. the upper bound) with first subprob with the linear soln, picking the var with the largest decimal portion e.g. x2 = 2.67, branch out with <= 2 or >=3 constraints, since x2 cannot be between 2 & 3 since that's not a feasible answer. Taking the linear relaxation again for every node, repeat until all branches are pruned. For max prob, the best current feasible answer is the lower bound (because we assume there might be another solution that beats this solution), and the upper bound of a node is the maximum objective value over its entire region. If the feasible region of a node is empty, it is pruned. A branch is fathomed if the lb of a node is lower than the current LB. If the UB of a node is lower than the current LB, that branch is pruned. For min, for each partition, a lower bound is calculated by solving the linear relaxation of the partition, if an integer feasible solution is found within a partition, it becomes an upper bound for the overall problem. Because each node presents a restricted feasible region therefore its lower bound is not a lower bound of the original problem. Since the open nodes represent a partition of the original feasible region, the lowest lower bound is a valid lower bound for the original problem

The optimality gap is the difference between the best feasible integer solution found so far and the lowest bound among the unexplored nodes. If the gap is within an acceptable range, the algorithm may stop, considering the solution sufficiently close to optimal. Strategy: Feasibility heuristics quickly find feasible integer solutions to provide a good upper bound, reducing the search tree size by pruning nodes that cannot yield better solutions. Solve the LP relaxation to get a fractional solution. Round this solution to the nearest integer. Project back to the feasible region to reduce constraint violations. Repeat until a feasible integer solution is found or a set number of iterations is reached. Feasibility pump play is used to find feasible solutions that may not necessarily be optimal.

Newton's Method **approximates** the function locally as quadratic to get the direction and step size. However, this assumption may not always be accurate, which is why techniques like **line search** are used to adjust the step size as needed.

For non differentable functions like mod x, it can be dealt with a change in formulation when we allow constraints.

Advantage – Quadratic convergence, which means at every iteration we double the number of exact digits in our approximation. Disadvantage – Scalablity, n^2 number of memory units for n variables for hessian, and for inverse near cube of that.

In order to have a **unique** solution for the Newton step, we need the Hessian to be invertible:

$$x_{k+1} - x_k = -H_f(x_k)^{-1}\nabla f(x_k)$$

Penalty – For converting constrained function to an unconstrained one by adding the constraint to the obj with a penalty that penalises for not satisfying that constraint. i.e if y is the constraint, then if y<=0, then penalty is 0 for a minimising problem, otherwise infinity. The worst-case penalty would make obj infinity, and it would be impossible to solve. Approximating the penalty function with another function that is smooth and does not take infinite values like P(y) = u * y, where u slope is nonnegative. So now penalty will be proportional to how much you violated the constraint. But we are rewarding the obj when P(y)<0, which we don't want, because it would affect the optimal value. If we consider all the slopes of this linear function, and take the max instead, we recover the original zero infinity function. After rearranging max and min, we do the minimisation of the obj function by setting the gradient to 0 and solve for x. Maximizing with respect to u(making it the dual problem)

$$\max_{u \ge 0} \min_x \; x_1 + x_2 + u\,(x_1^2 + x_2^2 - 1)$$

Infimum of a set: largest lower bound on values in set, doesn't have to be part of the set. **Supremum** of a set: smallest upper bound on values in set

Dual Optimisation problem – Constrained to unconstrained, multiplying with scalar slopes, taking the max(we want the inequality scalar to be positive and the equality scalar to be either positive or negative, since we want to penalise both). This is the lagrangian of the convex problem. This dual problem gives us a lower bound on the optimal value of the primal problem (because we are minimising first), which is the weak duality theorem. For strong duality, slater's condition. Lagrangian Relaxation is like duality in LP, where we relax constraints to create a problem that is easier to solve. The dual function inf L(x, v, u) = d(v,u) is always concave. The **dual problem** maximizes d(v, u) subject to u>=0

$$\max_{v_i, u_j \ge 0} \min_{x \in \mathbb{R}^n} f(x) + \sum_i v_i\, h_i(x) + \sum_j u_j\, g_j(x)$$

1. **Purpose**: Slater's Condition helps ensure **strong duality** in convex optimization problems with inequality constraints, meaning that the optimal values of the primal and dual problems are equal.
2. **Condition**: Slater's Condition states that for a convex optimization problem:
   - If all inequality constraints $g_i(x) \le 0$ are **convex and nonlinear** (they aren't just affine/linear constraints), then **strong duality holds** if there exists a point x^* strictly within the feasible region.
   - Specifically, this point x^* must satisfy: $g_i(x^*) < 0$ for all i, and $h_j(x^*) = 0$ for all equality constraints $h_j(x)$.
3. **Implication**:
   - If such a point x^* exists (one that strictly satisfies the inequality constraints), then **strong duality** is guaranteed.
   - In other words, the optimal values of the primal and dual problems will be equal, and duality theory can be applied effectively.

So, **finding an x^* that satisfies all constraints strictly does not automatically mean strong duality holds**, but if such an x^* exists and the problem is convex, Slater's Condition guarantees strong duality. This is a sufficient condition (not necessar…

Dual Gradient Ascent – Dual Gradient Ascent is a method for solving the dual problem by iteratively adjusting the dual variable u to maximize the dual function. The dual gradient ascent method iterates between solving the inner level problem for the primal variables (provided the current iterate for the dual variables), and performing a gradient ascent step on the dual variables.

$$x_{k+1} := \arg\min_x \mathcal{L}(x, \mu_k)$$
$$\mu_{k+1} := \mu_k + \alpha_k g(x_{k+1})$$

KKT Conditions – The **KKT Conditions** represent a set of first-order necessary conditions for finding local optima in constrained problems, applicable when certain regularity conditions are met, like slater's. The gradient wrt x needs to be zero. Complementary slackness ensures that if a constraint $g(x)$ is inactive (i.e., it does not bind the solution), its corresponding Lagrange multiplier must be zero.If only one constraint, then the derivate of f and g are inversely proportional i.e. d(f) = -ud(g). So, x should be feasible, so g(x)>=0, u>=0, penalty terms ug(x) = 0, so it doesn't affect the optimal value. Useful for designing general purpose optimisation solvers.

x^* to be a local minimum, there exist some KKT multipliers μ λ

$\nabla_x \mathcal{L}(x^*, \mu, \lambda) = 0$	(stationarity)
$g(x^*) \le 0 \quad h(x^*) = 0$	(primal feasibility)
$\mu \ge 0$	(dual feasibility)
$\mu_j g_j(x^*) = 0$	(complementary slackness)

Revised Simplex: Operates with matrix manipulation, reducing memory and computational load, making it ideal for larger LP problems. **Dual Simplex**: Allows for adjustments in feasibility while maintaining optimality, which is useful for dynamically changing constraints.

A vertex in R^n is defined by the intersection of **n** linearly independent hyperplanes (associated with n equality or active inequality constraints). **Step 1**: Select n linearly independent constraints from the m constraints available. **Step 2**: Solve the resulting n×n system to locate an intersection point, a candidate for a vertex. **Step 3**: Confirm that this point satisfies all constraints to ensure its within the feasible region. The number of potential vertices grows exponentially with problem size, making exhaustive checking impractical. Simplex optimizes by focusing only on feasible vertices that lead toward the optimal solution and if **there are no neighbouring vertices that improve the objective, then our current vertex is optimal.**

Degeneracy occurs when multiple vertices share the same coordinates due to overlapping constraints, potentially causing Simplex to revisit the same solution. Degeneracy can lead to **cycling**, where the algorithm repeatedly encounters the same points, slowing or even preventing convergence. **Pivot Rules** like **Bland's Rule** ensure convergence by choosing pivots that prevent revisiting points, helping the algorithm move past degenerate vertices.

Pivoting

The act of picking a **constraint to remove from the active set** (which edge to move along), and then picking a **constraint to replace it**.

$\bar{A}^\intercal \mu = c$ — $\min_{x \in \mathbb{R}^n} c^\intercal x$ s.t. $Ax \le b$

For a problem in form, any elements of μ that are positive represent constraints we could kick out of S_k. What if max? (conventions may vary)

What if none are positive?

- We have **the optimal value**, so can stop.

When there are multiple, then **pivot selection rules** are utilised, which are important for guaranteeing convergence when there is **degeneracy**.

Efficiently, Improving the obj value.

Alternative Simplex Summary

Each ☐ is a system of linear equations to be solved

Current active constraint set:	S_k
Vertex position:	$\bar{A}x = \bar{b}$
Pick leaving constraint: (otherwise optimal)	$\bar{A}^\intercal \mu = c \rightarrow u \in \{u \mid \mu_u > 0\}$
Pick joining constraint: (otherwise unbounded)	$\bar{A}d_u = 1e_u \rightarrow s = \tilde{b} - \tilde{A}x,\; r = -\tilde{A}d_u \rightarrow v \in \arg\min_{v \in S_k \mid r_v > 0} \frac{s_v}{r_v}$
Next active constraint set:	$S_{k+1} := (S_k \setminus \{u\}) \cup \{v\}$

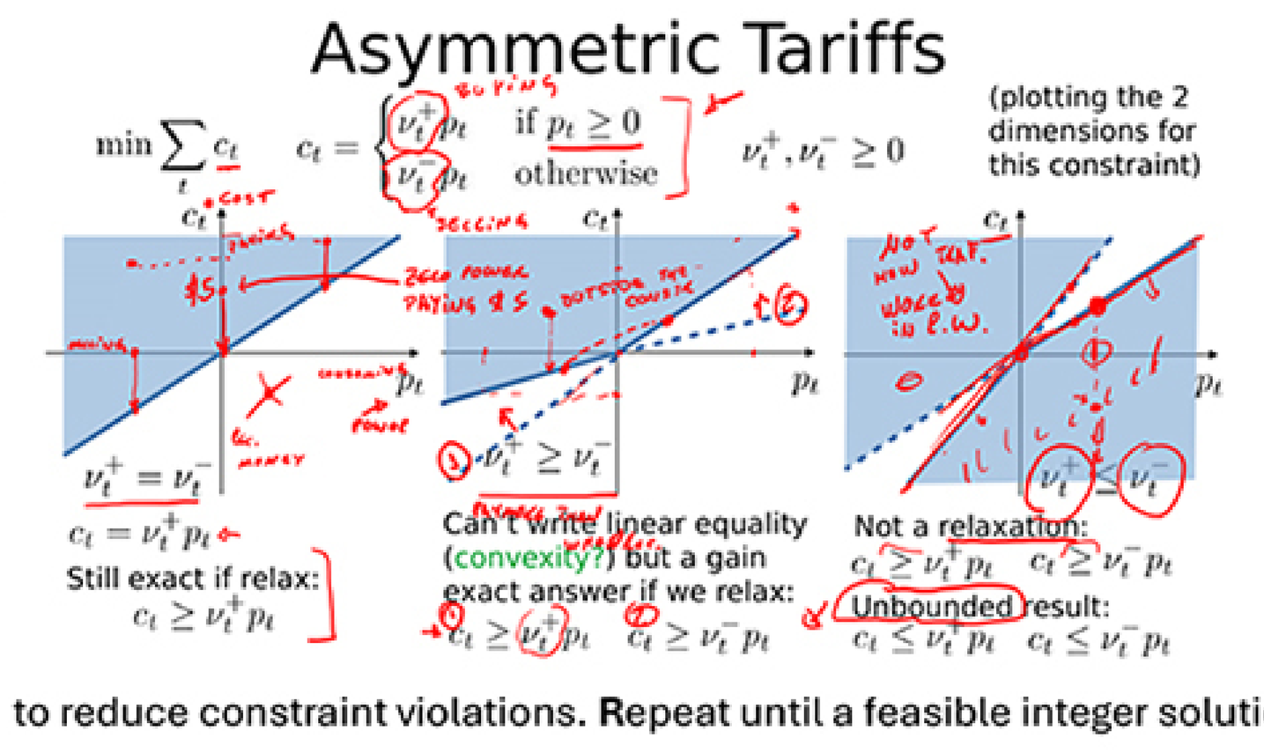
A **basis** is a set of linearly independent columns from the LP constraints matrix, representing the current active constraints in a BFS. **Variables in Basis**: Basic variables correspond to the basis, while non-basic variables are set to zero in the current solution. Start at an initial bfs, entering, leaving and then **pivot**: Update the solution based on the new basis.

A relaxation is an approximation where we **increase the feasible set.** Can be done cstr to cstr, like x>=10 to x>=9. **Relaxation** involves loosening some constraints of a problem to make it more tractable, usually by converting non-linear or integer constraints into linear, continuous forms. **Convex Relaxation**: A specific form of relaxation where non-convex constraints are transformed into a convex set, allowing for LP or convex optimization techniques. Conversely, we can restrict the feasible set by creating a subset with tightened points or tightening constraints, i.e. cutting off chunks of feasible region like x+y<=10 to x+y<=8. **Exact Relaxation:** The modified problem can have same objective value and exact solution (same optimal var values). i.e. min fx or arg min fx is equal to modified set X'. Inactive constraints can be removed or tightened to become active i.e. removing the slack.

Asymmetric Tariffs

$$\min \sum_t c_t \qquad c_t = \begin{cases} \nu_t^+ p_t & \text{if } p_t \ge 0 \\ \nu_t^- p_t & \text{otherwise} \end{cases} \qquad \nu_t^+, \nu_t^- \ge 0$$

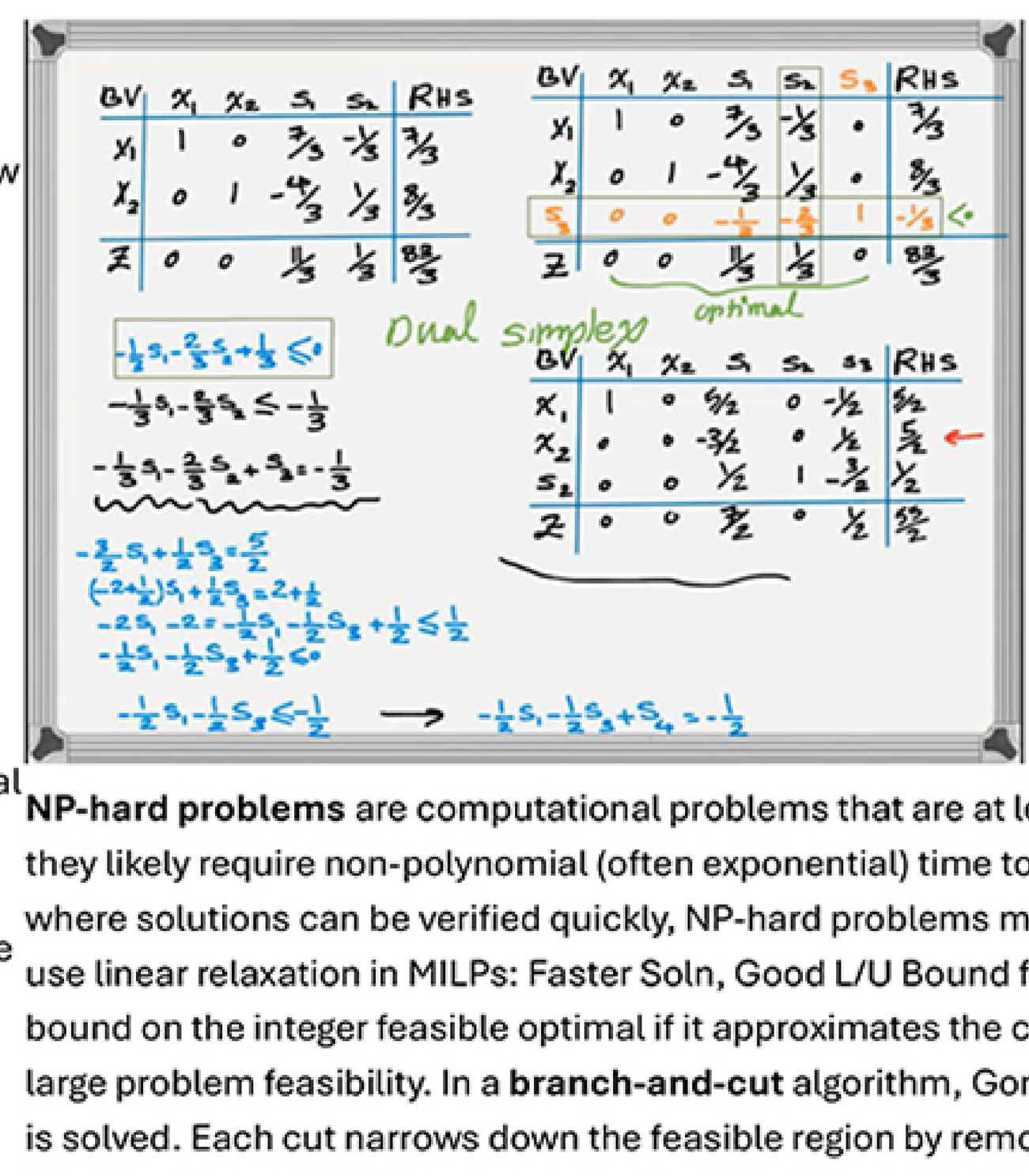
(plotting the 2 dimensions for this constraint)



P is the total power, when positive charging, as in buying power from the grid. Second case is when it costs us more to buy than to sell. We must minimise C i.e. y axis in the feasible region. It's also nonconvex, due to not being affine. So we relax the constraints to be intersection of regions above the lines. Third region is not a relaxation because I can't recover the optimal soln. Second is unbounded since we are trying minimize c.

Scheduling Constraints - Start Time Variables: s_j represents the start time of job j (Suitable for continuous time representation and precedence constraints but can complicate resource modeling). Indicator Variables: x_jt = 1 if job j starts at time t, and 0 otherwise, with time discretized into intervals (Suitable for modelling resource constraints by representing each period but requires time discretization and more variables). s_j >= s_i + d_i, where d_i, can be used to set deadlines. X4 <= x5 – d4 + 12u, x4>=x5+d5-12(1-u), where u dictates whether job 4 comes before or after. For indicator, like y3 <= sum of t for y2, upto t-d2-d3, i.e. to ensure no otr job.

Gomory Cut – 1Step: Linear Relaxation, **2Step:** Add new constraints called the cutting cstrs, to cut the feasible region and get best f integer soln. For that, solve simplex, pick one of the basic variables with fractional values and write eqn from the optimal table e.g. x1 + 7/3s1 = 1/3. Rewriting the equation so the fractional values are written as the summation of the integer value and a decimal portion e.g. x1 + (2+1/3)s1 = 1/3 -> x1 + 2s1= -1/3s1+1/3. For RHS, since slack are non-negative, that means -1/3s1 +1/3<=1/3, rounding rhs to integer, -1/3s1 + 1/3<=0 – cutting eqn.



Adding slack for the cut eqn, adding it to the optimality table. Since Z row is optimal, and RHS contains a negative value, it fulfills the condition of being solved by dual simplex, where we get the pivot coln using min test. Until the RHS consists of only integer values, continue adding cutting constraints until feasible soln is found. **Limitations** – fractional cuts work with the assumption that all variables are integers, so RHS for slack vars must also be integers. And if x1 and x2 are fractional, slack must be fractional too, otherwise turn them all into integers.

Gomory cuts are valid when the linear relaxation is bounded and produces a fractional solution

NP-hard problems are computational problems that are at least as difficult as the hardest problems in NP, meaning they likely require non-polynomial (often exponential) time to solve as the input size grows. Unlike NP problems, where solutions can be verified quickly, NP-hard problems may not even have a quick verification method. Why we use linear relaxation in MILPs: Faster Soln, Good L/U Bound for optimisation (LP solution will offer a stronger lower bound on the integer feasible optimal if it approximates the convex hull closely), approaching integer solns via cut, large problem feasibility. In a **branch-and-cut** algorithm, Gomory cuts are added iteratively to the LP relaxation as it is solved. Each cut narrows down the feasible region by removing fractional solutions, gradually steering the solution towards ILP convex hull. By combining branching and cutting, the branch-and-cut algorithm efficiently tightens the feasible region, improving the chances of finding the optimal integer solution faster.

$$ay + bz \ge |x|$$
$$x \le cy$$
$$x \ge -dz$$
$$x \le -e \vee x \ge f$$

Introducing a binary variable $u \in \{0,1\}$, the constraints become:

$$ay + bz \ge x,\quad ay + bz \ge -x,\quad x \le cy,\quad x \ge -dz$$
$$x \ge f + (\underline{x} - f)u$$
$$x \le \overline{x} + (-e - \overline{x})u$$

where the bounds are given by:

$$\underline{x} = \max(-a\overline{y} - b\overline{z}, -d\overline{z})$$
$$\overline{x} = \min(a\overline{y} + b\overline{z}, c\overline{y})$$

Branching divides the parent node's feasible region into smaller parts, including its linear relaxation. The union of the child nodes' feasible regions is generally smaller, making their optimal solutions at least as large as the parent's optimal solution.

We have a ILP with the following constraints:

$$x + 3y \le 4.5$$
$$8x + 2y \le 23$$

where $x, y \in \mathbb{Z}_{\ge 0}$. Tighten these two constraints without eliminating any integer feasible solutions.

If we just consider each constraint separately, we can tighten them to:

$$x + 3y \le 4$$
$$4x + y \le 11$$

However, we can go further by consider both constraints in conjunction:

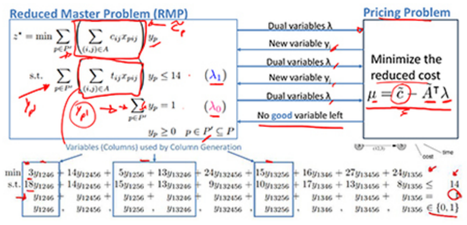
$$y \le 1$$
$$x + y \le 2$$

An example of these getting progressively tighter, for the respective linear relaxations:

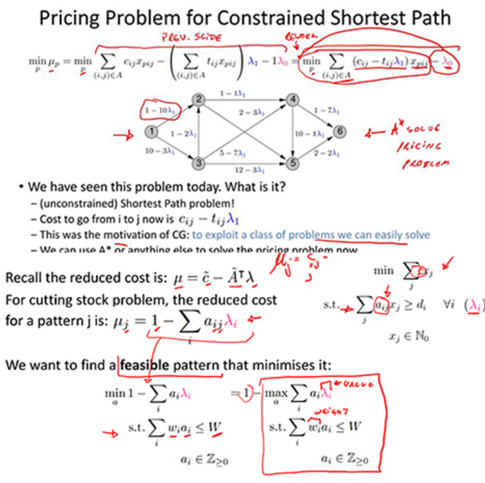
- The original allow $y = 4.5/3$ as a solution
- The independent tightening would allow $y = 4/3$ as a solution
- The combined tightening would allow at most $y = 1$ as a solution

DECOMP

Column Generation: This technique addresses complicating constraints by reformulating the problem into a "simpler" problem with many variables, called the Master Problem -> iteratively builds a (RMP) by selecting which variables or columns to add. Applications: **Constrained Shortest Path Problem:** Find the minimum-cost path from a start node to a goal node in a network, with each arc representing a cost, solvable by Dijkstra's. Addition of complexity like a time limit, A* would require modifications to handle time constraints i.e. arrival times. **Flow Model**: Formulate the problem to push one unit of flow from the start node and extract it at the goal node, each arc (i,j) has a binary variable xij to denote if it is used in the path, converted into ILP and solvable by branch and bound. Complicating constraint that makes the problem challenging to solve directly, like the **time constraint** (not exceeding 14 units). **Reformulation Strategy**: Reformulate the problem by treating **paths** as individual variables. This approach shifts the focus to finding an optimal path without needing to consider all paths simultaneously. This separation enables us to introduce **paths dynamically as needed** and on demand generation. The constraint matrix is large, this approach leads to many variables (paths), especially in dense networks. However, we will use Column Generation to introduce only the paths that improve the solution, avoiding the need to consider all paths initially.



Reduced Master Problem (RMP): **current version of problem** with only a subset(P') of paths (columns). We repeatedly solve the RMP as more paths are added. In Column Generation, the **Pricing Problem** identifies paths that can enhance the current RMP solution by calculating their reduced cost. **Reduced Cost**: represents the **marginal increase in the objective** when each non-basic variable is marginally increased. Here roof(A) is the Basic part of A in roof(A)roof(x) = b for basic variables, and A~ is the non-basic part of A. c~ is the obj coeff of the NB vars, and lambda is the dual variables for the basic variables. **Lambda 0** won't affect our reduced cost, because we already have chosen one path, no need to incorporate, already been satisfied, and it's 1 if no path is chosen. Since the pricing problem does not have a comp cstr, it's easy to solve, and only paths with **negative reduced cost** are added to the RMP, as these are paths that could improve the objective function. **Tldr:** Start with a set of initial columns and solve the RMP using these columns. Then, use the dual variables from this solution to define a new subproblem known as the Pricing Problem. Solve the Pricing Problem to find the column with the minimum reduced cost. If this new column has a negative reduced cost, add it to the RMP and repeat the process. If not negative, then the optimal solution to the linear relaxation of the problem has been found. **Branch & Price:** If a path variable xij has a fractional value in the RMP solution x12=0.6), we create branches to enforce integer solutions. **Branch 1**: x12=0, **Branch 2**: x12=1, for 0, remove all paths that use 1->2, inverse for 1**. Very similar** process for **cutting stock**, that identifies new cutting patterns that minimize waste while



changes can be made to the solution, allowing the algorithm to move from one solution to another in search of an improved outcome. In the **n-queens problem,** The **Random Walk** algo for n-queens selects random moves from the neighbourhood and executes them continuously, asymptotically complete – visits every state. **Hill Climbing** where the best move—or one of the best moves—is chosen at each step to reduce conflicts. This means evaluating all possible moves for each queen and selecting the one that most improves the solution (i.e., minimizes conflicts). This method requires evaluating the **entire neighborhood** to identify the optimal move. Unlike greedy search, which evaluates the entire neighborhood to select the best move, **first-found** randomizes the neighborhood evaluation. This approach picks the first improving move it encounters rather than the absolute best. **Randomized greedy** combines elements of randomness with greedy selection. With a probability p, the algorithm will perform a greedy or first-found move; otherwise, it will make a random move. This probabilistic element ensures that the algorithm is **asymptotically complete. Biased**, moves are selected with a probability inversely proportional to their score. For example, a move that reduces conflicts more significantly is more likely to be chosen.

VRP: The **1-Move** operator relocates a single customer from one route to another. **Swap 1-1** swaps two customers between different routes. **Swap 2-1** operator exchanges two customers in one route with a single customer from another. **Swap Tails** involves swapping the end segments, or "tails," of two routes. The 2-Opt operator removes two arcs within a route and reconnects them in a way that shortens the path, while 3-Opt removes three arcs and reconnects them in various possible configurations. These operators provide a way to systematically reduce travel distances, enhancing the efficiency of the overall routing solution. **Neighbourhoods - Knapsack problem**: Swap 2 items, Swap 1 item with multiple items of equal size, **Scheduling -** swapping jobs between machines or the order of jobs, **Map Coloring (k-coloring),** adjusting neighborhoods to allow single-color changes, local search can more easily explore feasible solutions and satisfy colouring constraints. **Soft constraints** allow some degree of flexibility. Instead of completely blocking certain solutions, they impose penalties for violations but allow the algorithm to explore those solutions, like allow adjacent regions to have the same colour but apply a penalty to the obj fn. Effectively defining a neighbourhood structure provides a balance between exploring new areas and intensifying the search around promising solutions.

Local minima are points where the solution is better than all neighboring solutions, making further improvement difficult. **Plateaus** are flat areas where many neighboring solutions have similar values, causing the search to stagnate. To address these issues, **random restarts** allow the search to start fresh from a new random solution whenever it gets stuck, helping explore different regions. **Sideways moves in 8-queens** permit non-improving moves across plateaus, with a set limit of 100 to avoid getting stuck indefinitely.

Simulated Annealing: Temperature influences the acceptance of new solutions. At higher temperatures, the algorithm allows greater diversification by accepting worse solutions, which broadens the search for new areas. As the temperature decreases, the algorithm shifts towards intensification, focusing on refining and exploiting the 'good' solutions already found. In Simulated Annealing, maintaining a constant low temperature does not guarantee that only improving solutions are accepted because there's always a small probability of accepting worse solutions (considering $P(\text{accept increase } \Delta) = e^{-\Delta/T}$). Additionally, using a constant low temperature throughout the search limits the algorithm's ability to explore the solution space effectively, increasing the risk of becoming trapped in local minima. The **cooling schedule** is crucial here; common approaches use a geometric cooling rate (e.g., Tk+1=α·Tk with α usually in [0.9, 0.999]) to gradually reduce the temperature. Don't want too many tempr changes. Steps involve updating T, re-boil(reinitialize t), handle randomly, and adapative parameters like maxChangesThisT.

Meta-heuristics are versatile optimization strategies that guide the search process in both discrete and continuous domains to find good feasible solutions without guaranteeing exact optimality. They differ from exact methods, as they are not based on algebraic models and often use **randomization** to explore the solution space. Typically **non-deterministic**, meta-heuristics may yield different results with each run and often stop based on time or iteration limits rather than achieving a guaranteed optimum. They are generally **problem-independent**, meaning they can be applied to a variety of optimization tasks, though they can incorporate **problem-specific heuristics** when beneficial. Many meta-heuristics also utilize **memory**, tracking past solutions to better direct the search process.

satisfying demand. Here a is the no of items of length i for pattern j. The max gives us a knapsack problem which is np easy. **Dantzig-Wolfe Decomposition** is a technique used to solve large-scale optimization problems with a **block structure** by breaking them into smaller, more manageable subproblems. Each subproblem, or "block," is optimized independently through **Column Generation**, where only the most promising solutions (columns) are generated. These subproblem solutions are then coordinated in a **Master Problem** that enforces global constraints, ensuring all blocks align to meet the overall objective. The process iterates, with new constraints added to the subproblems based on the Master Problem's solution, making it an efficient method for complex problems with strong interdependencies. **Complicating Constraints:** These constraints link variables in ways that make the feasible region difficult to define or compute efficiently, often adding complexity to the problem. Examples include the maximum time constraint in the constrained shortest path problem and the decision on cutting lengths in the cutting stock problem. **Complicating Variables**: These variables make solving the problem challenging because they are not continuous and/or link multiple constraints. Fixing their values simplifies the optimization problem. For instance, deciding whether to build facility k (yk∈{0,1}) in facility location problems.

Benders Decomposition: This technique handles complicating variables by decomposing the problem into an RMP and a subproblem. The RMP contains only the complicating variables and a new variable η. Its constraints involve only the complicating variables, optimality cuts, and feasibility cuts. Each iteration solves the subproblem by fixing the complicating variables to their RMP values, generating new cuts for the RMP. Bender's decomp, like colgen, works efficiently on block structures problems, where the subproblems (horizontal blocks) touch a little of the cvar. We don't know where the schools are, and the schools are the cvar i.e. y

X must be continuous; y can be continuous or integer. So, initialize y^, which is now a cons tant, so any constraint with y term is taken to the right side i.e. solve the prob in x for fixed y.

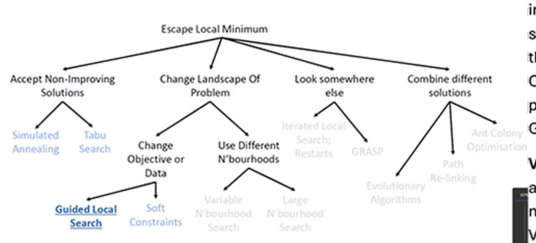
We like the DSP because y^ only appears in the obj, and cuts only apply on complicating constraints. Eita or η is the solution to the dual subproblem. If feasible, πe that achieves the maximum is a potential solution, and max of the dsp is atleast $\pi e\,(d - By)$. If DSP is unbounded, SP is infeasible.

η is going to be greater or equal than the objective fn of my dsp applied to this set of points -> this is going to be an optimality cut. If unbounded, rq(d-By)<=0 is the feasibility cut.

1: $\text{UB} = \infty, \text{LB} = -\infty$
2: $\bar{E} = \emptyset, \bar{Q} = \emptyset$
3: $\hat{y} = y_0$
4: **while** UB - LB $\geq \epsilon$ **do**
5: solve $\max_\pi \{\pi^\intercal(d - B\hat{y}) : \pi^\intercal D \leq c\}$
6: **if** Unbounded **then**
7: Get extreme ray r_q
8: $\bar{Q} = \bar{Q} \cup \{r_q\}$
9: **else** if FEAS
10: Get extreme point π_e
11: $\bar{E} = \bar{E} \cup \{\pi_e\}$
12: $\text{UB} = \min(\text{UB}, f^\intercal\hat{y} + \pi_e^\intercal(d - B\hat{y}))$
13: **end if**
14: solve $z = \min_{y \in Y}(f^\intercal y + \eta : \bar{E}$ and $\bar{Q}$ cuts)
15: $\hat{y} = \text{argmin}(y)$
16: $\text{LB} = z^*$
17: **end while**

If the solution is unbounded, it is no longer a polytope but a polyhedron. In such cases, vertices alone are insufficient for representation, so we use vectors, known as extreme rays rq, to indicate directions in which the solution is unbounded. Dual variables and extreme rays belong to exponentially large sets but are generated only as needed. In **column generation**, we solve the linear relaxation by iteratively adding variables and typically use **branch and bound** to handle integer constraints. In contrast, **Benders' decomposition** operates independently of branch and bound, separating the problem into a master problem and subproblem and iterating to optimality through cuts, without requiring branching. **In LPs**, a complicating variable is the dual of a complicating constraint and vice versa. A complicating constraint in the primal LP can be tackled with Column Generation or addressed via Benders Decomposition in the dual LP. Busplus (application) - problem, where is the hub? Any bus stop potentially can be a hub, how do we connect these hubs, how do we minimize the no of hubs? Min of summation of xkbk and cik;xik, for cstr summation xik = 1, xik<=yk, yk is integer, xik is continuous.
They provide no gaurantee of global or local optimality. A crucial feature of meta-heuristics is balancing **intensification** and **diversification**—intensification refines the search around promising areas, while diversification explores new areas to avoid getting stuck in local minima. **Tabu Search**: extends local search by tracking recently visited solutions to avoid cycling back into the same local minima. After finding a local minimum, tabu search selects a non-improving move and places it on a "tabu list," making it temporarily forbidden. This tabu list has a fixed length, meaning older moves eventually "fall off," allowing them to be revisited. Adjusting the list length is critical: a short list risks re-entering the same local minima, while an overly long list restricts exploration. Dynamic list length- see same soln, increase list length, decrease when incumbent found. An **aspiration criterion** allows tabu moves if they yield a new best solution, ensuring that promising paths are not blocked by the tabu list. Tabu search is widely used in problems like the VRP, where operators such as **1-move** adjust routes efficiently.

Guided Local Search (GLS) is a meta-heuristic that improves standard local search by adding penalties to specific features, guiding the search away from local minima. The objective function in GLS is modified to incorporate these penalties, allowing the algorithm to escape suboptimal areas by discouraging frequently encountered poor-quality features.



$$h(s) = g(s) + \lambda \cdot \sum_{i=1}^{M} p_i \cdot I_i(s)$$

λ: A parameter controlling the penalty strength. pi: Penalty for feature i, which increases as the feature is penalized. Ii(s): Indicator function that equals 1 if feature iii appears in solution s, and 0 otherwise.

Initialize: Begin with an initial solution s and set penalties pi=0 for all features. **Local Search**: Perform a local search to minimize the original objective g(s). **Select Features to Penalize**: Calculate the **utility** of each feature. Features with high utility are prioritized for penalization. **Update Penalties**: Increase the penalty piby 1 for all selected features. Perform local search on the augmented objective h(s) and continue the process.

$$util(s_*, f_i) = I_i(s_*) \cdot \frac{c_i}{1 + p_i}$$

For genetic, The algorithm starts with a **population** of potential solutions, each assessed for quality through a **fitness** score. The process involves several stages. First, **selection** pairs solutions based on their fitness, giving higher chances to better solutions. Then, during **crossover**, parts of each parent solution are combined to produce offspring, mixing traits from both parents. **Mutation** introduces random changes in the offspring to maintain diversity, which helps the algorithm avoid local minima and explore a broader search space. Finally, poorer solutions are replaced by the offspring, and the population evolves over multiple generations. Merging in vrp involves merging the first truck's route from the first grand tour(array), merging it with the second grand tour's second truck route, and removing all the locations the latter truck visited in the former one. There are many ways of mutating merging depending on the number of trucks and number of grand tours For 2 trucks in two gt, two ways of merging. The parents don't have to be always feasible, they can be close to feasible too.

Ant Colony Optimization (ACO): virtual ants explore paths and reinforce optimal routes through **pheromones**, which reflect path desirability. Ants probabilistically choose paths (partly at random) based on pheromone levels and **heuristic values** (like distance in the Traveling Salesman Problem). After each iteration, paths selected by successful ants receive more pheromone reinforcement, while others experience decay. This reinforcement encourages more ants to follow shorter paths over time, leading to convergence on near-optimal solutions.

Probability that ant chooses arc xy
τ : Pheromone strength
η : A–priori "attractiveness"

$$p_{xy}^k = \frac{(\tau_{xy}^\alpha)(\eta_{xy}^\beta)}{\sum_{z \in \text{allowed}_x} (\tau_{xz}^\alpha)(\eta_{xz}^\beta)}$$

METAHEURISTICS

Construction algorithms are essential in combinatorial optimization, where they create a foundational solution for problems like the Traveling Salesman Problem (TSP) or Knapsack. These algorithms construct solutions by adding one element at a time, setting up a structure that more advanced methods can later refine. A common approach in construction is the **"Nearest Neighbor"** method, often applied to TSP. Starting at a "home" city, it iteratively adds the nearest unvisited city to build a route. Greedy, performs locally optimal moves, often make "structural" errors, really good to ok for knapsack and scheduling. **"Minimum Insert Cost"** algorithm considers multiple potential positions for each addition and chooses the insertion point that minimally impacts the current solution's cost. By carefully balancing the cost of each insertion, this method helps avoid the structural flaws of simpler greedy approaches and performs well in more complex scenarios like the Vehicle Routing Problem (VRP), where routes and costs must be optimized across many stops. It allows insert between any pair of customers-not just at the end. **Randomization** in construction algorithms introduces variability by selecting a random element, such as a customer, and inserting it in the least costly position within the solution. This approach, unlike strict greedy methods, allows any element to be placed optimally rather than only at the end, which can prevent common structural errors and avoid overly deterministic solutions that lead to local minima. It's also to introduce unconventional solution paths that might not emerge in a deterministic approach and to mitigate deterministic biases. For Randomized greedy, choose the rth best of n choices i.e r = n * (rand.uniform(0,1)^y.

Vehicle Routing Problem (VRP): The VRP involves finding optimal routes for a fleet of vehicles to serve a set of customers at minimum cost. Solutions are built by adding customers in ways that minimize added cost to the route. This problem is foundational in logistics and transport optimization. **Regret Heuristic**: The regret heuristic adds foresight by considering the difference (or "regret") between the cost of the best route for a customer and the second-best alternative. The customer with the highest regret value is prioritized, aiming to minimize potential losses from future choices. The concept extends to k-regret, where choices are based on differences among multiple top routes, ensuring better overall optimization by accounting for future limitations. **Regret with Seeds**: To improve the initial structure in VRP, "seeds" are used to select initial customers based on distance criteria. For example, Seed 1 is the customer farthest from the depot, Seed 2 is the farthest from Seed 1, and so on. This method helps in laying out an initial skeleton for routes, enabling more balanced and structured solutions as regret-based choices build on this framework.

Can be extended to the k-th best routes:
k-regret $(i) = \sum_{j=2}^{k} (f(j,i) - f(1,i))$

Bespoke Method: Involve ordering customers in a systematic way, often based on geographic or other logical sequences, to simplify route construction. Like "Sweep": Sweeping through an area, clustering those customers until max capacity is reached, route is formed. Not good for constraints involving perishable goods.

Problem Definition: 3D packing addresses the challenge of fitting a collection of items into a set of trucks or containers efficiently. Each item has specific dimensions (width, height, depth), weight, and a due date, while each truck has compartments with set dimensions, weight limits, and departure times. The goal is to maximize the number of items packed while respecting all constraints. **Problem Statement**: Given a fleet of trucks and a set of items with various attributes, determine how to fit as many items as possible into the trucks. Constraints include observing weight limits, fitting items within truck dimensions without overhangs (bigger item on top of the smaller item), respecting load and departure dates, and adhering to placement rules (e.g., "top-load only" items must not be stacked under other items). **Solution**: Using a metaheuristic to solve this i.e. LNS, with 2 related subroutines – distribution of items among the trucks, fitting the items inside a truck. The solution involves packing items starting from corners to utilize boundaries effectively. Corners are easier to track and provide stability. Tops of items act as additional "containers" for further stacking, and items with the same top height can be joined to create flat surfaces for optimized stacking. This layered approach facilitates stable, efficient packing while following the constraints, reducing wasted space and maximizing load efficiency.

Local search is an optimization technique that improves an initial solution by exploring its "neighborhood"—a set of solutions reachable through small adjustments. A neighborhood is defined by an **operator** that specifies how

ILS improves solutions by performing repeated local searches combined with periodic **perturbations** to escape local minima. Once a local minimum is found, ILS applies a "kick" or "shake" with 'strength' of how many elements to change. This perturbation strength must be balanced—too weak, and the search risks returning to the same minimum; too strong, and it becomes a random restart. A common perturbation for the Traveling Salesman Problem (TSP) is the **Double Bridge** move, which replaces four arcs to generate a new, slightly altered solution.

GRASP (Greedy Randomised Ascent Procedure) - constructs solutions in a randomized, greedy fashion. It starts with an empty solution and gradually adds elements using a **Reduced Candidate List (RCL)** based on the cost of each addition (e.g., minimum insert cost in TSP). Each new element is chosen randomly from the RCL, allowing controlled randomness. The parameter alpha controls the balance: low α yields a greedy solution, while high α introduces more randomness. Like ILS, GRASP can partially destroy and reconstruct solutions, combining systematic construction with controlled randomness to balance exploration and exploitation. A key characteristic of the construction phase in GRASP is that it constructs a solution by selecting components from a Restricted Candidate List (RCL) based on certain strategies. The RCL is created using a greedy function to evaluate and rank potential components. Instead of always choosing the top-ranked component (as in a purely greedy algorithm), GRASP introduces randomness by selecting from the RCL, which includes several high-quality candidates.

VNS - VNS dynamically changes neighborhoods to escape local minima. Starting with a list of neighborhoods, the algorithm performs a local search in one neighborhood. If no improvement is found, it switches to a larger neighborhood. When an improvement is achieved, it returns to the smallest neighborhood to refine the solution. For VRP, neighborhood moves might include **1-move**, **1-1 swap**, **2-2 swap**, **Or-opt**, and **k-opt** moves. Define multiple neighborhoods, create arcs that jump things, visit some other states or solutions that our other neighborhood didn't allow us to do, changing how you get to the optimal min, but the local minima might change, you want the neighborhoods to get more complicated (from smaller to larger branch of factors, computationally cheap to expensive since it resets, computationally efficient) as k gets larger(increment count of neighbors), helps to escape the stuck local minima, exhaust the search space of the neighbor before moving onto the next one. No requirement there should be no intersection, i.e. neighbors can intersect, the conjunction doesn't have to be empty.

LNS: optimizes solutions by partially destroying and reconstructing them. This "destroy-and-recreate" approach removes parts of the solution (such as customers in VRP) and reinserts them using a preferred insertion method. The destruction phase can vary, for example, removing the longest arc, choosing a sequence of related customers, or dumping all customers from certain routes. The extent of destruction is typically random within set limits to ensure flexibility. For VRP, remove some visits, move them to assigned lists.

During reconstruction, LNS may use methods like **Minimum Insert Cost** or **k-Regret** to rebuild the solution systematically. This balance between structured removal and strategic reinsertion makes LNS effective in complex optimization problems, allowing significant changes without losing the overall solution quality.

Adaptive Large Neighborhood Search (ALNS) builds on LNS by adapting both the removal and insertion methods based on past performance, increasing the chances of finding optimal or near-optimal solutions.

To destroy, rand (min, max) where min is a number that is meaningful to destroy and max is not too large, otherwise it might be harder to reconstruct. For milp reconstruction, the original solution is a feasible solution to the milp, so you are either guaranteed to have either or better solution, which is a nice property to have. Validating against a milp is very effective, as the number of free variables is going to be very small and is guaranteed to get the global minimum given enough time, since the variables are fixed on the milp, so the model is going to be 1000 or million times bigger, but it's effective due to the low number of free variables.

Adaptive insert method - Method keeps accumulating that reward, starting with random uniform. Start with a small reward, and after 100 iterations, we use the rewards as a basis to rerate the probabilities, so the method that got more chosen is going to be chosen more often, keep doing that every 100 iterations. Same with **adaptive destruction method**, but you don't want any methods to have zero probability, so guard against zero should be there.

ADVANCED OPTIMISATION TOPICS

You have seen several heuristic search algorithms in the AI class to solve unconstrained combinatorial problems. You have seen that A* using an admissible heuristic is guaranteed to find the optimal solution to the problem.

Recall that A* works by minimising the cost of reaching the goal and an admissible heuristic for A* is a function $h(s)$ such that $h(s) \leq h^*(s)$ for all states s of the problem where $h^*(s)$ represents the optimal cost to reach the goal from s. In other words, an admissible heuristic $h(s)$ is a lower bound on the optimal solution for all s.

In the multi-objective version of A*, a multi-objective cost function $\vec{c}(x) = [c_1(x), c_2(x), \ldots, c_m(x)]$ is used for $m \geq 2$ and $h^*(s)$ now represents the **Pareto Front** for the state s.

Explain how to generalize the concept of an admissible heuristic for the multi-objective case and describe the property an admissible multi-objective heuristic should satisfy.

Furthermore, use your definition to provide an example of an admissible heuristic value for the Pareto front $\left\{\begin{pmatrix}162\\7.2\\34\end{pmatrix}, \begin{pmatrix}149\\7.0\\38\end{pmatrix}, \begin{pmatrix}140\\7.5\\45\end{pmatrix}, \begin{pmatrix}152\\8.2\\29\end{pmatrix}\right\}$

Multi-Objective Optimisation

The formal definition of an admissible heuristic for multi-objective A* is:

A multi-objective heuristic h is a set of vectors and it is admissible if and only if, for all states s, for all vector $\vec{u}^*$ in the Pareto Front of s there exists a vector $\vec{v} \in h(s)$ such that $\vec{v} \preceq \vec{u}^*$.

The key differences from the single objective case are:

- replace $\leq$ by $\preceq$
- make sure that every value in the Pareto Front has at least 1 vector in the heuristic that is "better than or equal to" it, i.e., $\vec{v} \preceq \vec{u}^*$.

Also, keep in mind that the best admissible heuristic possible is the optimal solution, i.e., a lower bound that is equal to the function being bounded. For multi-objective, this is the same thing, i.e., the Pareto-Front is the best possible admissible heuristic. The ideal-point or using an individual admissible for each direction is an **example** of admissible multi-objective heuristic and it can be greatly improved in practice.

Stochastic Programming

In this question, let's consider that our random variable ϵ represents all the uncertainty in our stochastic problem and is "discrete", meaning $P(\epsilon = i) > 0$ for $i \in \{o_1, \ldots, o_k\}$. This is exactly the case we discussed in the lecture.

An important property of the expected value operator is its "**linearity**": given a random variable ϵ, $E[a\epsilon + b] = aE[\epsilon] + b$ for $a \in \mathbb{R}$ and $b \in \mathbb{R}$. It is easy to see this when you expand the expected value operator, i.e.,

$E[a\epsilon + b] = \sum_i P(\epsilon = i)(a \times i + b) = a(\sum_i P(\epsilon = i)) + b = aE[\epsilon] + b$

In class, we learned that the optimal solution of the recourse-problem is less than or equal to the optimal solution of the expected-value problem. Formally, as we saw in class, let $g(x, \hat{\epsilon})$ represent the stochastic problem for a fixed of the value $\hat{\epsilon} \in \{o_1, \ldots, o_k\}$ for the random variable ϵ. Moreover, for this question, assume that this stochastic problem takes the form of a **Linear Program**, that is, the objective function is a linear function of x and $\hat{\epsilon}$ and the feasible region Ω is defined by a set of linear constraints. Then we have $\min_{x \in \Omega} E_\epsilon[g(x, \epsilon)] \leq \min_{x \in \Omega} g(x, E_\epsilon[\epsilon])$

Explain why the recourse-problem $\min_{x \in \Omega} E_\epsilon[g(x, \epsilon)]$ can potentially be lower than $\min_{x \in \Omega} g(x, E_\epsilon[\epsilon])$ for a **Linear Program**, despite the **linearity** of the expected value operator.

Using the same notation as the L-Shape Method slides, we have that $\min_{x \in \Omega} g(x, E_\epsilon[\epsilon])$ is:

$$\min_{x,z} f^\top x + (\sum_i P(\epsilon = i)c_i)^\top z$$
$$\text{s.t. } (\sum_i P(\epsilon = i)B_i)x + Dz = (\sum_i P(t = i)d_i)$$

where all the occurrences of "$(\sum_i P(\epsilon = i) \cdots)$" are computing the expectation regarding to the uncertainty. In other words, they will become the average vector or matrix depending on what the target of the expectation is.

Linearity of expectation still holds, e.g., we can move z in the objective function inside the summation computing the expectation and value is still the same: $(\sum_i P(\epsilon = i)c_i)^\top z = \sum_j (\sum_i P(\epsilon = i)c_i)_j * z_j = \sum_i (\sum_j P(\epsilon = i)c_{ij} * z_j)$. Using this, we can create an specific value of z (the recourse action) for each possible value of the uncertainty, i.e., change z to z_i. But this alone is not enough. We need to specialize the constraints, i.e., change $(\sum_i P(\epsilon = i)B_i)x + Dz = (\sum_i P(t = i)d_i)$ to $B_i x + Dz_i = d_i \quad \forall i$ and this change in the feasible space is what allows us to obtain a potential smaller objective function.