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:

$$\text{k-regret}(i) = \sum_{j=2}^{k} \left(f(j,i) - f(1,i) \right)$$

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

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.

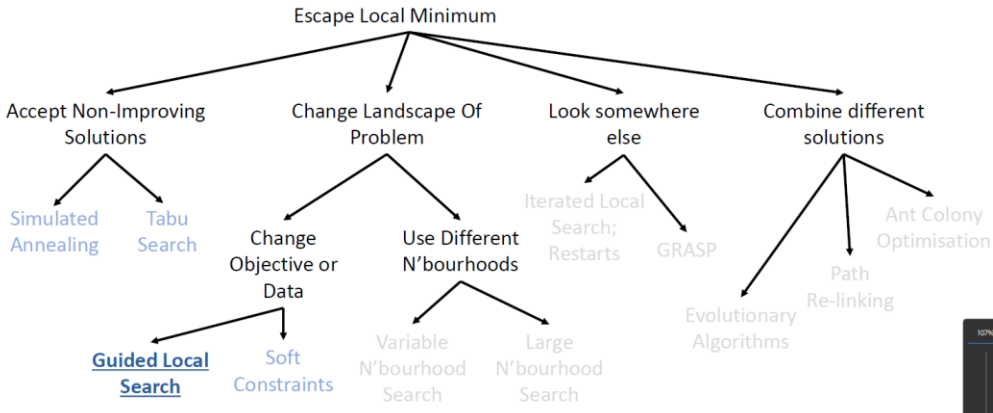
Local minima are points where the solution is better than all neighboring solutions, making further improvement difficult. **Plateaus** are flat areas where many neighboring solutions have similar values, causing the search to stagnate. To address these issues, **random restarts** allow the search to 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., $T_{k+1}=\alpha \cdot T_k$ 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.

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.

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

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.

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"

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

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.