Heuristic Summary

Massimo Perego

Contents

TL;DR: Problems

Combinatorial Optimization: A problem is a CO problem if we need to optimize an objective function and the solution is a subset of a finite ground set, from which the search space is defined.

Knapsack Problem KP: From a set of object, select the subset with maximal value that stays within a weight constraint.

Maximum Diversity Problem MDP: Inside a metric space, select the k points with the maximum pairwise distance.

Bin Packing Problem BPP: Put a set of objects into the minimum number of containers, given the capacity of each container.

Parallel Machine Scheduling Problem PMSP: Divide a set of tasks, each with its own length, among a set of machines, minimizing the maximum length of a singular machine.

Max-Sat: Given a CNF, assign truth values to the logical variables in a way that satisfies the maximum weight subset of its logical clauses.

Set Covering Problem SCP: Given a binary matrix, select a subset of column of minimum cost covering all rows at least once.

Set Packing Problem: Given a binary matrix, select the subset of non-conflicting columns with maximum value (each row covered by at most one column).

Set Partitioning Problem SPP: Given a binary matrix, select a subset of non-conflicting column of minimum cost, covering each row (exactly one column for each row).

Vertex Cover Problem VCP: Given an undirected graph, select the subset of vertices of minimum weight/cardinality such that each edge is incident to at least one vertex in the solution.

Maximum Clique Problem: Given an undirected graph, find the subset of pairwise adjacent vertices of maximum weight (heaviest clique possible).

Maximum Independent Set Problem: Given an undirected graph, find the subset of pairwise non-adjacent vertices of maximum weight (heaviest set of independent vertices).

Traveling Salesman Problem TSP: Given a directed graph, find the Hamiltonian circuit of minimum weight, i.e., a circuit such that it visits each node exactly once (weight on the arcs, obv).

Capacitated Minimum Spanning Tree Problem CMSTP: Starting from a root, find the minimum spanning tree such that each branch respects a capacity (number/sum of weight of the arcs).

Vehicle Routing Problem VRP: Given a directed graph, with a starting node, select the set of circuits such that each node is visited exactly once and each circuit respects the capacity.

Algorithm Analysis

Approximation: To determine how "good" an algorithm is, we can look at how close its solution ($f_A(i)$) is to the optimal one ($f^*(i)$):

- **Absolute difference $\tilde{\delta}_A(I)$:** the simple difference between solution found and optimal solution
$$\tilde{\delta}_A(I) = f_A(I) - f^*(I)$$
- **Relative difference $\delta_A(I)$:** it's the ratio between the absolute difference and the optimum (often as a percentage)
$$\delta_A(I) = \frac{f_A(I) - f^*(I)}{f^*(I)}$$

- **Approximation ratio $\rho_A(I)$:** The ratio solution found and optimal solution
$$\rho_A(I) = \frac{f_A(I)}{f^*(I)}$$

These are all for minimization problems, everything is switched for maximization (keep stuff always positive/greater than 1).

Worst case analysis: A compact measure, independent of the instance, is the worst case; the approximation can be:

- **Absolute:** the result will never be worse than the optimum by more than a fixed amount;
- **Relative:** the ratio of the approximate result and optimal one will never exceed a certain constant α.

Beyond the worst case: Using only the worst case can be *rough*, some alternative approaches are:

- **Average-case:** assume a probability distribution on the instances and evaluate the expected value of the approximation factor;
- **Randomization:** the operations of the algorithm depend on some random values, the solution becomes a random variable that can be investigated;
- **Parametrisation:** prove an approximation guarantee that depends on other parameters of the instances besides the size n; we add parameter k and the time can be expressed as $T(n, k)$, dependent on both, the problem could be polynomial in n but exponential in k; k could be part of the input or the solution (in which case we'll know it only *a posteriori*, but an estimate could be available);
- **Kernelization:** transform all instances of the problem into simpler ones, instead of instances of another problem; it becomes a simpler problem to solve, allowing it to be solved faster (either exactly or heuristically) or to prove that there exists an optimal solution and maybe know which elements are certainly in said solution.

Run Time Distribution Diagram RTD: It shows the probability that the execution time is below a certain time t (axis: runtime-P of solving). To draw it: how many of the instances are solved in that amount of time or less?

Scaling Diagram: Describes the dependence of the time on the size (axis: instance size-execution time). Logarithmic scale for the time shows that the algorithm is exponential, both axis logarithmic can show that the algorithm is polynomial. To find the coefficient: choose two points and

$$\left(\frac{s_1}{s_2}\right)^{\alpha} \approx \frac{T_1}{T_2}$$

find α, with s_1, s_2 and T_1, T_2 instance sizes and times respectively.

Solution Quality Diagram SQD: It plots the probability that the relative difference δ is smaller than a value α, for each possible value of α (axis: relative solution quality-cumulative frequency, i.e., how many times do we get that solution quality or better). To draw it: sort the relative differences by non-decreasing values and determine how many of the instances are under each value.

Boxplot diagram: Show median ($n/2$th element), surrounded by the box formed by the lower and upper quartiles ($n/4$th element and $3n/4$th element, respectively). The whiskers go out to the end of the range in both directions.

Dominance: The dominance can be:

- **Strict:** a boxplot is fully below the other;
- **Probabilistic:** each quartile is not above the corresponding one of the other algorithm.

Wilcoxon's Test: It's a test, focused on effectiveness, used to determine if the empirical difference among two algorithms is significant. Steps:

1. Compute the difference of the algorithms results, instance by instance
2. Sort them by increasing absolute values and assign a rank (number) to each one
3. Give a sign to each rank, positive if the difference was positive, negative otherwise
4. Sum all positive ranks (W^+), as well as the negative (W^-)
5. Calculate the probability that $|W^+ - W^-|$ is equal or larger than the observed value (not required during exams)

If the probability is low enough the difference is significant and if $W^+ > W^-$ then the first algorithm is better than the second, or vice versa.

Constructive Heuristics

Construction Graph: Every construction heuristic A defines a construction graph, in which the nodes are all the possible subsets $x \subseteq B$ acceptable for A, and the arcs connect all solution pairs such as $(x, x \cup \{i\})$, with $x, x \cup \{i\} \in \mathcal{F}_A$, $i \in B \setminus x$ (same solution, cardinality of one more). The graph is directed and acyclic by definition.

Pure and Adaptive: A constructive algorithm is

- **Pure:** if the selection criterion depends only on the new element i;
- **Adaptive:** if φ_A depends both on i and the current solution.

Traveling Salesman Problem TSP: Types of Heuristic:

- **Nearest Neighbor:** from the last visited node to the nearest unvisited node;
- **Cheapest Insertion:** removes an arc and adds two, including a new node, such that the total change is minimum;
- **Nearest Insertion:** look for the node closest to the circuit (minimum distance from another node in the circuit) and find the best way to include it (which arc do I remove to include it? Use the one that minimizes total weight);
- **Farthest Insertion:** search for the node farthest from the circuit and connect it in the best way possible.

Greedoids and Matroids: Assuming that the objective function is additive, and the solutions are bases (maximal subsets, can't add any more stuff) of the search space. A greedy algorithm is possible for a problem (ground set B and search space $\mathcal{F} \subseteq 2^B$) which is a **greedoid**, i.e., for which the following hold:

- **Trivial axiom:** $\emptyset \in \mathcal{F}$, the empty set is an acceptable solution;
- **Accessibility axiom:** if $x \in \mathcal{F}$ and $x \neq \emptyset$ then $\exists i \in x : x \setminus \{i\} \in \mathcal{F}$. Any acceptable subset can be built adding its element in a suitable order, i.e., there is at least one path that leads to every solution in the search space;
- **Exchange axiom:** if $x, y \in \mathcal{F}$ with $|x| = |y| + 1$, then $\exists i \in x \setminus y$ such that $y \cup \{i\} \in \mathcal{F}$, any acceptable solution can be extended with a suitable element from a bigger solution.

Strengthening the accessibility axiom leads to a **matroid**:

- **Heredity axiom:** if $x \in \mathcal{F}$ and $y \subset x$ then $y \in \mathcal{F}$. Any acceptable subset can be built by adding elements in any order.

The constructive algorithm finds the optimal solution if $(B, \mathcal{F})$ is a matroid embedding.

The **optimality** of the greedy algorithm can be proven for a greedoid with strengthened exchange axiom:

- **Strong exchange axiom:**

$$\begin{cases} x \in \mathcal{F}, y \in B_{\mathcal{F}} & \text{s.t. } x \subset y \\ i \in B \setminus y & \text{s.t. } x \cup \{i\} \in \mathcal{F} \end{cases} \implies \exists j \in y \backslash x : \begin{cases} x \cup \{j\} \in \mathcal{F} \\ y \cup \{i\} \setminus \{j\} \in \mathcal{F} \end{cases}$$

Given a base y and a subset of the search space and base x, you can have an element i in the base but not in y that can be feasibly added to x. Then there is an element j in y but not x that can be added to x while replacing j with i in the base, all while remaining in the search space.

Regret-based constructive heuristics: What if something is good now but bad later? We want to measure the "bad later" with a regret function. Usually such a heuristic consists in:

- Partitioning the choices in classes;
- Compute, for each choice, the basic selection criteria;
- Compute, for each class, the regret, i.e., usually, the difference with the second-best choice or the average of all other choices;
- Choose the best choice for the class in which regret is maximum;

This works well when we know we'll have to pick from each class, and we fear being stuck with a much worse choice later.

Roll-out heuristics: Also known as single-step look-ahead constructive heuristics, at each step of the basic heuristic look ahead at the next step and compute the solution for each possibility. The complexity remains polynomial but much larger. It can be extended to multiple steps, worsening further the complexity.

Destructive Heuristics: Start with the full ground set and remove one element at a time, always remaining in the search space and while maximizing the selection criteria. Usually the solutions are much smaller than the ground set, so they take too long , but it might be useful to append a destructive heuristic to constructive solutions which might have redundant elements.

Constructive Metaheuristics

Adaptive Research Technique ART: Set a number of iterations ℓ for a basic constructive heuristic, at each iteration forbid elements inside the solution with probability π for a number of iteration L. When wanting to include an element, check that the current iteration is distant enough from the last ban of said element, i.e., if $t > T_i + L$ where t is the current iteration and T is the vector containing the tabu attribute for each element.

Greedy randomized Adaptive Search Procedure GRASP: Also called semi-greedy, instead of always choosing the best at each step, sometimes make a random step. How can we determine which step?

- **Uniform probability:** each possibility has the same probability;
- **HBSS:** sort the possibilities by non-increasing values of φ, assign a probability according to the position in the order;
- **Restricted Candidate List:** sort the possibilities, choose from a number of choices among the best. We can define the RCL through
  - **Cardinality:** take the μ best elements
  - **Value:** include all the elements
    - for minimization problems, with value below:

$$\varphi_\mu = \varphi_{\min} + \mu(\varphi_{\max} - \varphi_{\min}) = (1-\mu)\varphi_{\min} + \mu\varphi_{\max}$$

    - for maximization problems, with value above:

$$\varphi_\mu = \varphi_{\max} - \mu(\varphi_{\max} - \varphi_{\min}) = (1-\mu)\varphi_{\max} + \mu\varphi_{\min}$$

  We choose randomly among the *elite*.

Ant System: Similar to semi-greedy, but all choices are feasible and the probability of choosing an element is a function which depends on the selection criteria (φ, which is turned into visibility η) and some auxiliary information (trail τ), updated after each iteration. To update we have the oblivion parameter ρ and the conversion coefficient Q:

$$\tau_i = \begin{cases} (1-\rho)\tau_i & \text{for } i \notin x \\ (1-\rho)\tau_i + \rho \cdot Q \cdot f(x) & \text{for } i \in x \end{cases}$$

where $f(x)$ is the value of the final solution. We multiply all trails by $(1-\rho)$ and then add $\rho Q f(x)$ to the elements in the current solution, since they're "promising".

Exchange Heuristics

Neighborhood: $N : X \to 2^X$, a neighborhood is a function which associates each feasible solution to a subset of feasible solutions. We can define a neighborhood based on:

- **Distance:** how distant a solution is from another, all the solutions within a certain distance form a neighborhood. An example of distance is the Hamming distance: if we represent the solution as the incidence vector of its elements, the Hamming distance is the number of elements in which the solutions differ, e.g., $N_{\mathcal{H}_1}$ is the neighborhood of solutions one element away from each other;
- **Operations:** how many operations $\mathcal{O}$ are needed to get from a solution to another, e.g., $N_{\mathcal{S}_1}$ is the neighborhood with an element swapped.

Steepest Descent: Starting from a solution, move to the best solution in the neighborhood and repeat until there are only worsening moves. The best solution is always the last one and is a local optima. Possible strategies:

- **Global best:** among all neighboring solutions take the most improving;
- **First-best:** choose an order for the solutions in the neighborhood and select the first improving one found.

Landscape: It's the triplet (X, N, f), where X is the search space, N is the neighborhood function and f is the objective function. It is the search graph with node weights given by the objective. The effectiveness of Steepest Descent depends on the landscape, the smoother it is the better the results.

Very Large Scale Neighborhood (VLSN): Larger neighborhoods (generally) yield larger attraction basins, making Steepest Descent more effective, but they take longer to explore. With Very Large Scale Neighborhoods (exponential/high polynomial in $|B|$) we need to limit computational time: if the objective can be optimized without an exhaustive search we can explore the neighborhood heuristically. We want to parameterize the neighborhood, define a composite move as a set of elementary moves (compatible and commutable) and increase or decrease the number of operations when necessary/sufficient to improve the solution. Finding the optimal solution in such neighborhoods requires solving an auxiliary problem, typically on a matrix or graph.

Order-first Split-second: It's a method used for solving partition problems:

- Build a starting permutation of the elements to be partitioned
- Partition in an optimum way, under the additional constraint that elements of the same component be consecutive in the starting permutation (keep the ordering)

We can exploit an **auxiliary graph**:

- each node corresponds to an element, plus a fictitious node 0;
- starting from each node, there's an arc to the next if adding that component is feasible for the solution (consider all the elements from the node in which we started, do this starting from each node).

Variable Depth Search VDS: A composite move is a sequence of elementary moves. From each solution in the basic neighborhood, make a sequence of moves optimizing each elementary step, but allowing worsening moves and forbidding backwards moves. Terminate when the solution becomes worse than the starting one or all moves are forbidden, return the best solution found along the way. Kinda like a roll-out for exchange heuristics, try all moves (without getting *too bad*, bounded in the original solution) and find out what works best.

Destroy and Repair: Also called iterated greedy methods, when the *right* number of exchanges is unknown it's a possible idea to:

- delete from the solution a certain number of elements k;
- complete it with a constructive (repair) heuristic.

Exchange Metaheuristics

Iterated Local Search ILS: At each iteration, it perturbates the last accepted solution to create a new starting point, finds the local optima from there and then it decides whether to accept it or not. The perturbation procedure (ideally) allows to move out of the attraction basin of a local optima (not too strong or it's a random restart, not too weak or it risks remaining in the same basin) and the acceptance procedure evaluates whether the result found is a promising starting point for the following iterations (accepting only improving solution favors intensification, accepting any solution favors diversification, has to be tuned).

Variable Neighborhood Search VNS: Finds the local optima for a certain neighborhood using Steepest Descent, then there's a shaking procedure to obtain a new starting solution, randomly extracted from the current neighborhood considered, repeat for a number ℓ of iteration. There's a hierarchy of neighborhoods and, after each iteration, if a better solution was found a smaller neighborhood is selected (intensification), a larger one is used otherwise (diversification).

Variable Neighborhood Descent VND: If a solution is the local optima for a certain neighborhood, then changing the neighborhood might lead to a better solution. It uses a family of neighborhoods, explores each one until it finds the local optima, then goes to the next one until the solution is a local optima for all neighborhoods. The neighborhoods can be heterogeneous (topologically different, e.g., $N_{\mathcal{S}_1}$ and $N_{\mathcal{T}_1}$) or hierarchical ($N_1 \subset \cdots \subset N_k$). The last solution found is always the best one.

Dynamic Local Search DLS: It keeps the neighborhood but modifies the objective function during the search to escape the local optima; useful when the objective function has many plateaus. The basic idea is to:

- Define a penalty function $w : X \to \mathbb{N}$;
- Combine the penalty and objective functions into one auxiliary function $\tilde{f}$; it can be applied additively or multiplicatively to the values of the elements;
- Apply Steepest Descent, optimizing $\tilde{f}$;
- After each iteration update the penalty values based on the results, modifying the landscape of the search.

Simulated Annealing: Start from a solution in a neighborhood, extract another solution at random, compute its value, accept it if better, if it's worse accept it with a certain probability dependent on the temperature, usually $e^{-\delta f/T}$, where δf is the difference between current and new solution and T is the temperature. The temperature is gradually reduced after each iteration, lowering the probability of accepting worsening solution as time goes on.

Tabu Search: It keeps the basic selection criteria of Steepest Descent, removes the termination condition (beware of cycling) and imposes a tabu on already visited solutions. While exploring a neighborhood, at each step it selects the best feasible non-tabu solution. It is potentially inefficient: at step t it

needs to check that the solution is new (the evaluation must be efficient) and the number of solutions grows indefinitely, along with the memory occupation. Forbidding solutions can also lead to disconnection of the search graph and/or slow down the exit from attraction basins. Solutions to these problems: forbid "attributes" instead of solutions or give a duration to the prohibition (tabu tenure).

Recombination Algorithms

Scatter Search: It generates a starting population of solutions and improves all of them with an exchange procedure, then it builds a reference set $R = B \cup D$ where B includes the best solutions and D the farthest solution from B (by some definition of distance). For each pair of solutions $(x, y) \in B \times (B \cup D)$:

- Recombine x and y, generating z
- Improve z with an exchange procedure, obtaining z'
- Check if z' belongs to either B or D
- Terminate when R is unchanged

Recombining inside $(B \times B)$ intensifies the search, while recombinations inside $(B \times D)$ diversify.

Recombination: Usually the solutions x and y are manipulated as subsets; the recombination goes as follows:

1. Start with $z = x \cap y$, they concur, so they *must* be good elements (right?);
2. Augment z using elements from $x \setminus y$ or $y \setminus x$, using some criteria;
3. If necessary, add external elements from $B \setminus (x \cap y)$;
4. If necessary (z is unfeasible), apply an auxiliary exchange heuristic (repair procedure).

Path Relinking: Given a neighborhood N and an exchange heuristic based on it, collect in a reference set R the best solutions generated by the auxiliary heuristic (often this is just a final intensification, that one is the normal heuristic) and for each pair of solutions $x, y \in R$

- Build a path from x to y inside the neighborhood, choosing at each step the solution closest to y;
- Find the best solution along the path (and improve it);
- If said solution it's good enough for the reference set, add it to R.

Encoding-based Algorithms: Many recombination heuristics manipulate encodings of the solutions, rather than the solutions themselves, this aims to abstract and distinguish the method from the problem, while building an operator that can work with any problem that uses a certain family of encoding.

Features of a Good Encoding: The following properties should be satisfied by a good encoding (decreasing importance):

1. Each solution should have an encoding;
2. Different solutions should have different encodings;
3. Each encoding should correspond to a feasible solution;
4. Each solution should correspond to the same number of encodings;
5. The encoding and decoding operations should be efficient;
6. Locality: small changes to the encoding should induce small changes to the solution.

Usual encodings: Some common encodings are:

- **Incidence vector:** A binary vector where each element corresponds to an item in the ground set and the value determines whether the object is in the solution or not; e.g., KP, if the value is 1 the element is in the knapsack;
- **Symbol strings:** When the ground set is divided into disjoint components and the solution must take one element from each component, we can define an alphabet for each component, with symbols corresponding to each element that can be part of that component in the solution; e.g., in the PMSP, we define an alphabet for each machine, containing a symbol for all possible tasks, whose inclusion in the solution means that said task is assigned to that specific machine;
- **Permutations:** If the solution can be expressed as permutation of a set, this can serve as an encoding; e.g., in the TSP, the solution is a permutation of nodes.

Genetic Algorithm: The most famous encoding algorithm, it builds and encodes a population and repeatedly applies:

- **Selection:** generate a new population starting from the current one, extracting elements with probability proportional to the fitness (the selection can either be proportional, simple probability related to the fitness value,

rank, probability related to the "position in the fitness line", or tournament selection, extracting a number of subsets and choosing the best individual from each subset);

- **Crossover:** combines 2 or more individuals to generate other, usually by splitting and swapping parts of the encoding;
- **Mutation:** modifies an individual to generate a similar one, the modification usually depends on the encoding, e.g., binary vector: flip some bits, Symbol strings: substitute some symbols at random, Permutations: swap elements, in some way.

Feasibility within encodings: Sometimes, crossover and mutation can generate encodings of legal but unfeasible solutions, this can lead to inefficiency, ineffectiveness and design problems. Approaches to face this problem can be:

- **Special encodings and operators:** to avoid/limit unfeasibility; e.g., only encode feasible solutions (such as with symbol strings for partitioning problems, can't get unfeasible with that) or define crossover/mutation operators that maintain feasibility, such as ones which simulate moves on the solutions; we're kinda giving up on abstraction though, going back to heuristics based on neighborhoods;
- **Repair procedures:** turn unfeasibility into feasibility; refine the decoder function to repair unfeasible solutions; this allows a bias in favor of feasible encodings and in favor of solutions which are more easily obtained by the repair procedure;
- **Penalty functions:** allow unfeasibility but discourage it; add a measure of unfeasibility (which has to be defined) to the fitness function, penalizing unfeasible solutions (which often have a better value of the objective function); it's better to use the smallest effective penalty to allow reaching "hidden" feasible solutions while not allowing too many unfeasible solutions to be found, it has to be tuned.