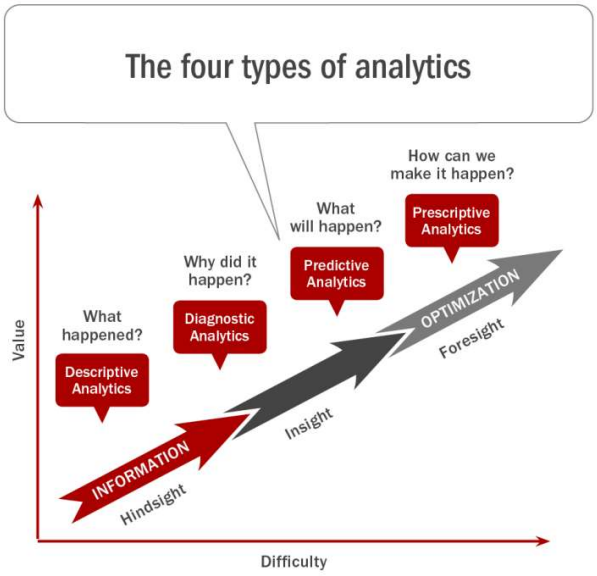
Combinatory decision making: many possibilities, different restrictions,

- choose any solution that meet all constraints → combinatory satisfaction
- choose the best solution according to an objective → combinatory optimization

Data analysis:



Starting from first 2 steps, based on analyzing past data, then we can make prediction on step 3, and with those predictions we can find optimal solutions in step 4

Starting from NP-hard (mostly) problems, we can find optimal solutions with intelligent search, this can be done in several ways, one of those is Constraint programming (CP)

Constraint Programming: A declarative programming paradigm for stating and solving combinatorial optimization problems.

User models a decision problem by formalizing:

- the unknowns of the decision → **decision variables** (Xi).
- possible values for unknowns → **domains** (D(Xi) = {vj }).
- relations between the unknowns → **constraints** (r(Xi, Xi')).

A constraint solver finds a solution to the model (or proves that no solution exists) by assigning a value to every variable (Xi ← vj) via a S**earch Algorithm**.

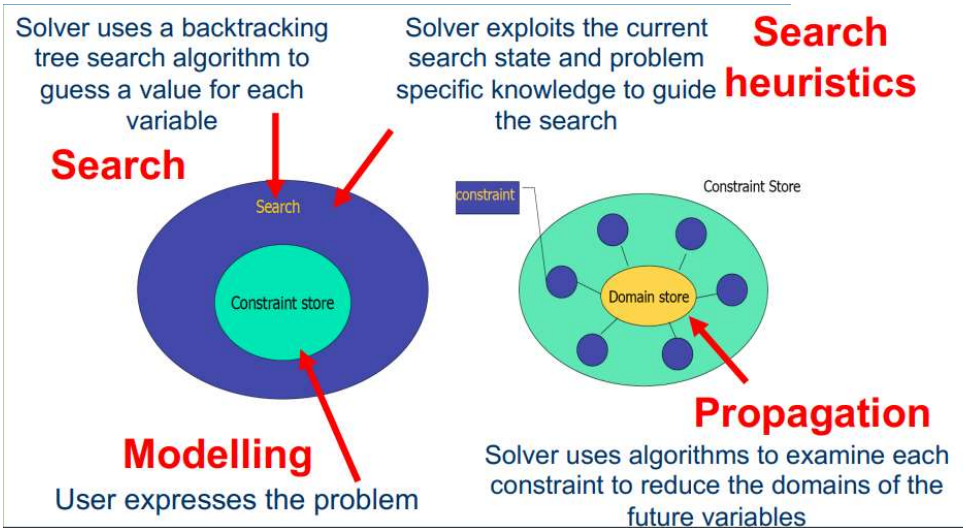
The big advantages of using CP are:

- it provides a rich language to define constraints and search procedures
- it focuses on constraints, so more constraints means more domain reduction, so problems that are easier to solve
- can contain messy constraints that are not linear in nature

CP doesn't actually have a special focus on optimality, even though it can be used also in that sense

Often the best choice for optimization are hybrid models (combining CP with ILP or HS)

Constraint solver: uses systematic backtracking tree search to guess value for every variable, and uses propagation to delete variables that are not compatible with the constrains, shrinking the domain



MODELING

Constraint Satisfaction Problem (CSP): CSP => triple <X, D, C>

- X: decision variables
- D: set of domains
- C: set of constraints

Solution to CSP = assignment of values to all variables, satisfying all constraints simultaneously

Constraint Optimization Problems (COP)
CSP enhanced with optimization criterion => <X, D, C, f>

- f: formalization of the optimization criterion

variable domains include: binary, integer, continuous
variables can also take a value from any finite set
also exist special “structured” variable types → set variables, activities or interval values

Constraints
Any constraint can be expressed by listing all allowed combination
It’s usually better to use relations between objects to make it more clear

Properties of constraints

- Order of imposition doesn’t matter
- Non-directional, a constraint between X and Y, also works between Y and X
- Rarely independent, usually variables are shared between constraints

Constraints can be:

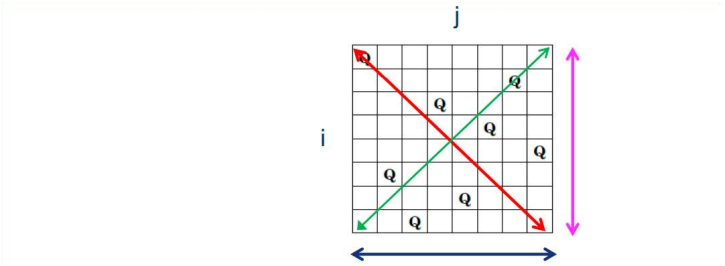
- Algebraic expressions → x1 > x2
- Extensional constraints (table constraints) → (X, Y, Z) in {(a, a, a), (b, b, b)}
- Variables as subscripts (element constraints) → Y = cost[X] (where cost is an array of parameters)
- Logical relations → (X<Y) v (Y<Z) → C
- Global constraints → alldifferent([X1, X2, X3])
- Meta-constraints → $\sum i$ (Xi > ti) ≤ 5 (means at most 5 variables should satisfy this constraint)

Permutation: function p: S → S, intuitively re-arrangement of a set of elements
Variable Symmetry: permuting variable assignments
Value Symmetry: permuting values

We need to reduce variable and value symmetry to reduce the set of solutions and the search space. Another way to reduce the set of solutions, and in turn, the search space, is by using lexicographic constraints.

Lexicographic constraint → A lexicographic constraint, compares two sets of expressions (conventionally expressed in arrays), index by index, to see which is greater in lexicographic terms.

Example for the n-queens problem:



- $lex\leq(B, [B_{ji} \mid i, j \in [1..n]])$
- $lex\leq(B, [B_{ij} \mid i \in [n..1], j \in [1..n]])$
- $lex\leq(B, [B_{ji} \mid i, j \in [n..1]])$
- $lex\leq(B, [B_{ij} \mid i \in [1..n], j \in [n..1]])$

- i, j → j,i
- i,j → reverse i, j
- i,j → reverse j, reverse i
- i,j → i, reverse j

Sometimes, we could need to combine multiple models to get to the better one. We can do that by adding channeling constraints.

For example:
In the n-queens problem, we could use a model in which we use n variables Y for columns, assigning values for the row position.
In the same way we can use a model with n variable X for rows in which we assign a value for column position.
By combining those two models, we could get rid of the alldifferent constraints used in each model, and use a single constraint for diagonal control. To do so in a way that, X and Y variables can coexist we can use the channeling constraint:

- **forall i, j $X_i = j \leftrightarrow Y_j = i$**

CONSTRAINT PROPAGATION & GLOBAL CONSTRAINTS

Constraint solver: Enumerates all possible values for variables via systematical backtracking tree search, during search it examines constraints to remove inconsistent values from the domains of the future variables, via propagation

Local consistency: A form of inference which detects inconsistent partial assignments (local because we examine individual constraints)

Popular local consistencies are domain based (they can change the domain values):

- **Generalized Arc Consistency** (GAC)
  - A constraint C on k variables C(X1, X2… Xk), gives the set of allowed combinations (called allowed tuples or supports), so if there's a value in any domain that doesn't appear in any support, is inconsistent
- **Bounds Consistency** (BC)
  - Defined for totally ordered domains, so we identify a domain as a range between min and max, a bound support is a tuple (d1,…,dk) where di is between min and max. Is weaker than GAC, cause in fact it considers only min and max, so not every value, but it might be easier to look for a support in a range than in a domain, achieving BC is often cheaper than achieving GAC

A local consistency notion defines properties that a constraint must satisfy after constraint propagation

Propagation algorithm: achieves a certain level of consistency on a constraint C by removing the inconsistent values from the domains of the variables in C, the level of consistency depends on C, sometimes we may have to re propagate a constraint to make it GAC

Global Constraints: Capture complex, non-binary and recurring combinatorial structures arising in a variety of applications

- Modeling benefits
  - reduce gap between problem statement and model
  - allows constraints not possible otherwise (semantic)
- Solving benefits
  - Strong inference in propagation (operational)
  - Efficient propagation (algorithmic)

Counting Constraints: restrict either number of variables or the number of times a value can be taken,

- alldifferent constraint is in fact a counting constraint, cause it counts that each value appears at most once
- Nvalue constraint, sets the number of possible different values, if the N is equal to the set dimension, it has the same effect as alldifferent
  - Nvalue([1, 2, 2, 1, 3], 3) → (s, N)
- Global cardinality constraint, sets the number of times each value from a set of values can appear in the set of constraints
  - gcc([1, 1, 3, 2, 3], [1, 2, 3, 4], [2, 1, 2, 0]) → (sX, sV, sO) V = value, O = occurrences
- Among Constraint, constrains the number of variables taken from a given set
  - among([1, 5, 3, 2, 5, 4], {1,2,3,4}, 4) → (sX, sV, N) N can also be an interval l, u

Sequencing Constraints: ensure a sequence of variables obey certain patterns

- Sequence/AmongSeq Constraint, applies among to a sequence in a sliding window pattern, so applying it on different sub-sequences
  - sequence(1,2,3,[1,0,2,0,3],{0,1}) → (l.u.q. sX, sV)
    - q = dimension of sub-sequence, l = lower bound, u = upper bound

Scheduling Constraints: Help schedule tasks with respective release times, duration, and deadlines, using limited resources in a time interval.

- **Disjunctive Resource Constraint**: constraint on the usage of a shared resource, making it that it doesn't overlap, the resource can execute at most one task at a time.

- **Cumulative constraint**: Constraint the usage of a shared resource which has a given capacity, that can do multiple tasks at the same time, each task with a different resource requirement, and that can never exceed the resource capacity.

Ordering Constraint: Enforce an ordering between the variables or the values
- **Lexicographic ordering constraint**: requires a sequence of variables to be lexicographically less than or equal to another sequence of variables.
- **Value Precedence Constraint**: requires a value to precede another one in a sequence of variables.
  - value_precede(5, 4, [2, 5, 3, 5, 4]) → (v1, v2, s) v1's first occurrence must precede v2's first occurrence.

Specialized Propagation for Global constraints, revolves around two main different approaches
- **Constraint Decomposition**: A constraint is decomposed into smaller and simpler constraints, each of which has a known propagation algorithm, so by propagating each of those we give a propagation algorithm to the original global constraint, this may not always provide an effective propagation.
- **Dedicated Ad hoc Algorithm**: often propagating a constraint via an ad hoc algorithm is faster than propagating the constraints in the decomposition.

A GAC Propagation Algorithm maintains GAC on alldifferent and runs in polynomial time, establish a relation between the solutions of the constraint and the properties of a graph → maximal matching in a bipartite graph, a graph where vertices are divided into 2 disjoint sets U and V, such that every edge connects a vertex from U to one of V.

Matching: in a graph is a subset of edges such that no 2 edges have a node in common, maximal matching is the largest matching possible.
If we assign variables on a partition and values of the domain on the other, finding the maximal matching and applying that to an algorithm corresponds to alldifferent constraint.

The algorithm:
- Construct variable-value graph
- Find a maximal matching M, otherwise fail
- Orient graph
- Mark edges starting from free value nodes using graph search
- Compute SCCs and mark joining edges
- Remove not marked and free edges

When re-executed:
- remove marks on edges
- remove edges not in the domains of the respective variables
- if a matching edge is removed, compute a new maximal matching
- otherwise just repeat marking and removal

This algorithm results in a runtime complexity equal to O(m), where m = sum of the dimensions of every variable's domain.

Table constraint: is like explicitly writing the possible values for given variables, requires knowing results before computation but is actually more effective than a decomposition
es: C(X1, X2) = {(0,0), (0,2), (1,3), (2,1)}

In a product configuration problem we use compatibility constraints on product components to rule combinations

Formal language-based constraints: we can use a DFSA (deterministic finite-state automaton) to define the solutions
regular constraint is a DFSA defined by 5 tuples:

- q: finite set of states
- ς: Set of symbols
- t: partial transition function q x ς → q
- q0: initial state
- f: final states

many constraints are instances of regular: lex, among, precedence, ecc… but we shouldn't use regular when we can use one of those

SEARCH

Backtracking tree search with no propagation: enumerates every possible variable-value combination via a systematic backtracking tree search, by default it uses depth-first traversal, when all the variables of a constraint is instantiated, the validity of the constraint is checked, systematic search eventually finds a solution or proves unsatisfiability, but its complexity is O(d^n), (exponential)

Propagation removes inconsistent values directly from the domains, avoiding the exploration of inconsistent assignments. When doing propagation, the solver doesn't only propagates on the single affected variable, but also on the constraints over variable. Which are affected by the domain changes on their variables.

Depth-First Search (DFS)
Branching Decisions: Usually consists of posting a unary constraint on a chosen variable Xi.

- Enumerating (or labeling): assign single values from D(Xi).
- Domain partitioning: Sometimes on models with large domains, it would be useful to partition the variable domains, creating a branch for each of the subdomains.

Branching/Search Heuristics: They guide the search on its decisions

- **Branching**
  - **Static Heuristics**: made in advance, like giving an order for the branch generation in the tree, pre-decided but really cheap.
  - **Dynamic heuristics**: at any node, any variable & branch can be considered and it's decided dynamically during search, hence costly, taking into count the current state of the search tree.
- **Search**
  - **Fail-first principle**: Try first where we are most likely to fail, aiming at proving as soon as possible that the search is in a sub tree with no feasible solutions, used in all the heuristics.
  - **Minimum domain** (dom): Choose next variable with minimum domain size, to minimize search tree size.
  - **Most constrained** (or maximum degree) (deg): Choose next the variable involved in the most number of constraints.
  - **Combination dom/deg**: Since both heuristics are usually good we can use a combination of both, it choose next the variable with minimum domain / degree.
  - **Weighted degree** (Wdeg): constraints are weighted starting from 1, during the propagation of a constraint, its weight is incremented by 1 if the constraint fails.
  - **Combination domWdeg**: combining dom and weighted degree.

Heavy tail behavior: Given a collection of instances of a problem, often we can observe some exceptionally hard instances that take exceptionally longer time to solve, that's not a characteristic of the instance specifically. Some times it can even change based on variable order or something this little. Therefor, for some combination of instance + solver parameters we get trapped into an exponential sub tree, such mistakes are seemingly random.

To overcome this problem:

- **Randomization** can really help, es: pick variables/values at random or randomly breaking ties
- **Restarting**: restarting the search after a certain amount of resources are consumed (usually search steps) and in the subsequent runs using a different search es: adding randomization or different heuristics

Different restarting strategies:

- **Constant restart**: restarts after using L resources
- **Geometric restart**: restart after a^nL resources where n = 0, 1, 2,…
- **Luby restart**: restart after s[i]L where s[i] is the i^{th} number of Luby sequence (112112411121124 …), proven to often be the most effective way

Limited Discrepancy Search: a discrepancy is any decision in a search tree that doesn't follow the heuristic. LDS trust heuristics and gives priority to the left branch, it iteratively search the tree by increasing number of discrepancies. But all discrepancies are alike; irrespective of depth and heuristics tend to make more mistakes at top of search tree, so therefor it is worth exploring top discrepancies first.

Depth-bounded Discrepancy Search: Biases search to discrepancies high in the tree via an iterative increasing depth bound, and discrepancies below this depth are prohibited. So, on the i^{th} iteration, DDS explores those branches on which discrepancies occur at a depth of i or less.

Constraint Optimization Problems (COP): CSP enhanced with optimization criterion es: min cost, shortest distance, max profit,…

There are different techniques that can be used in COPs solving:

- Search over the domain: es: to minimize f, if D(f) = 1..n, we try from the lowest values until we find a solution, that technique is called **Destructive lower bound** (cause it throws away all the intermediate computations):
  - we can also do **Destructive upper bound** where we start from the highest value in D(f)
  - Both techniques have fundamental good qualities, so, to have both we can use **Binary Search**, the main idea is to keep both a (feasible) upper bound ub and an (infeasible) lower bound lb:
    - solve by posting lb < f < (lb + ub)/2
    - if feasible, update ub; if infeasible, update lb
    - stop when a solution with f = lb+1 is found.

    On the other hand, almost all information is discarded between each attempt, which in turn results in a lot of repeated work

  (those algorithms operate the other way around for maximization problems)
- **Branch & Bound algorithm,** solves a sequence of CSPs via a single search tree and incorporates bounding in the search, to do so, each time a feasible solution is found, it posts a new bounding constraint which ensures that the future solution must be better than it and repeats until infeasible.

CONSTRAINT-BASED SCHEDULING

Scheduling: Ordering resource-requiring tasks over time.

Tasks: activity variables that we use for scheduling, those can be constrained on resources and time

Activities can be: (S start time, d duration, E end time)
- **Preemptive**: Can be interrupted at any time S + d <= E
- **Non-Preemptive**: cannot be interrupted at any time S + D = E

Resource: the available asset to execute the operations (es: volume of a truck, seats in a classroom,...)

Cumulative/ parallel resource: can execute multiple activities in parallel, the amount of resources requested by the parallel activities cannot exceed the maximum capacity of the resource.
Any 2 activities requiring the same resource are related by a cumulative constraint

Unary/Disjunctive resource: activities cannot overlap in time, independently of the resource capacity.
Any 2 activities are related by a disjunctive (noOverlap) constraint

Temporal constraints:
- **Precedence constraints**: forces one activity to end before another starts
- **Time-legs & Time windows**: Bounds the difference between end time of an activity and start time of another
- **Sequence-dependent set up times**: a separation constraint for unary resources

Cost function: a common one is the make-span, minimizing the time for all activities completion
minimize max([S1+d1,…, Sn+dn])

HEURISTIC SEARCH

Complete methods: guarantee to find for every finite size instance a (optimal) solution in bounded time. Might need exponential computation time

Approximate methods: cannot guarantee optimality, nor termination in case of infeasibility. Good quality solutions in a reduced amount of time
- **Constructive heuristics**:
  - Those are the fastest approximation methods. They generate solutions from scratch by repeatedly extending the current partial assignments until a solution is found, or stopping criteria are satisfied. Therefor it uses problem specific knowledge to construct a solution.
- **Local search**:
  - Often returns better solutions than constructive heuristics. Starts from an initial solution, iteratively tries to replace the current solution with a better one in an appropriately defined neighborhood by applying small (local) modifications. Can also start from an unfeasible assignment of all variables
    - **Neighborhood structure**: A function N: $S \rightarrow 2^S$ hat assigns to every s a set of neighbors N(s). Often implicitly defined by specifying the modifications that must be applied to s. The application of this operator is usually called a move.

        - **Local minimum**: locally minimum solution with respect to a neighborhood structure.
- **Meta heuristics**
    - High level strategies to increase performance, with the aim to not get trapped in local minima and search for better local minima. Do so by exploiting search history, heuristics, randomness and probabilistic choices and general strategies to balance intensification and diversification.
        - **Intensification**: exploitation of accumulated search experience (concentrate the search in small areas)
        - **Diversification**: exploration “in the large” of the search space
        - **Contrary and complementary**: we need dynamical balance to reach good effectiveness.

LS-based Methods: differently from LS, adds a diversification component to iterative improvement to escape local minima, like allowing worsening moves (S**imulated Annealing**), changing neighborhood structure (**Variable neighborhood search**, or taboo search, which doesn't directly changes neighborhood definition, but changes its structure. By exploiting the search history, therefor eliminating already explored solutions or moves) or changing the objective function (**Guided local search**, which modifies the search landscape with the aim of making it less desirable) during search, we combine that with a **termination criteria**, like maximum CPU time or maximum iterations.

Population-based Methods: At each iteration a population of solutions are used. The basic principle, is to learn correlations between good solution components.
Candidate solutions are generated using a parameterized probabilistic model, which is updated using the previously seen solutions in a way that the search will concentrate in the regions containing high quality solutions.

Ant colony Optimization (ACO): we simulate the pheromone trails used by ants, by a parameterized probabilistic model – **Pheromone Model** – which uses a set of parameters, called pheromone values, which act as the memory to keep track of the search process, to intensify search around best solution components,
ES. pheromone value t(X, v) for every variable and value in X domain, can represent the desirability of assigning v to X.
ACO employs constructive heuristics for probabilistacally constructing solutions using the pheromone values, all pheromone values are decreased by an evaporation factor, which allows ants to progressively forget older solutions and emphasize the most recent ones.

LS better when: neighborhood structures create a correlated search graph, inventing moves is easy, computational cost of moves is low, they got high intensification ability

Population-based methods better when: can be encoded as composition of good building blocks, computational cost of moves in Ls is high, difficult to design neighborhood structures, they got high Diversification ability.

Hybrid Meta heuristics: the idea is to use LS-based methods inside population-based ones.
- **Complete methods**
    - guarantee to find for every finite size instance an (optimal) solution in bounded time, but might need exponential computation time
- **Approximate methods**
    - cannot guarantee optimality, nor termination in case of infeasibility, but obtains good-quality solutions in a significantly reduced amount of time

- **Meta heuristics + complete methods**: Meta heuristics use a complete method to efficiently explore neighborhood
  - **Large neighborhood search**: even if finding an improving neighbor might be difficult, the average quality of local minima is high. The key idea is to use a generic large neighborhood and explore it with a complete method like CP. Therefor we view the neighborhood as the solution of a sub problem, and use tree search to exhaustively but quickly explore it,
    - digging deeper, we fix part of the variables to the value we have in a solution, and relax the remaining ones. We use different design decisions to choose complete vs incomplete exploration, how many variables to fix or which variables to fix
  - **ACO + CP**: Instead of using constructive heuristics, which are very problem specific, we can use CP for probabilistically constructing solutions using the pheromone values
    - after ACO+CP is performed we get a pheromone matrix
    - the resulting solution provides an upper bound
    - CP performs a complete search and uses the pheromone matrix as a heuristic information for value selection.