1 - Constraint Programming

Introduction

@February 23, 2023

A declarative programming (you state your decision, along with the constraints) paradigm for stating and solving combinatorial optimization problems. You model a decision problem:

- unknowns of the decision → decision variables (X_i)
- domains of decision variables $D(X_i) = \{v_j\}$
- relations between decision variables, which are the constraints $r(X_i, X_{i'})$
    - Availability Constraints
    - Frequency Constraints
    - Operational Constraints

We want to either find a solution or know that we will not find one. A **Solver** searches all the possible combinations and finds a solution (or proves that no solution exists) by assigning a value to every variable from its domain ($X_i \leftarrow v_j$) via a search algorithm.

Why? It provides a rich language for expressing constraints and defining search procedures, with easy modeling (I have fast prototyping with a variety of constraints, programs are easy to maintain and it is extensible) and easy control of search (I can experiment with advanced search strategies)

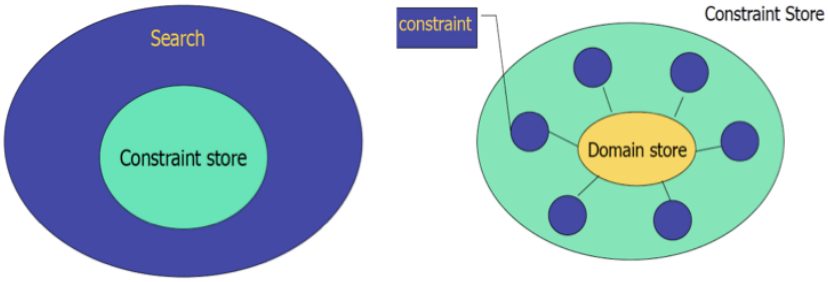
Two approaches in Combinatorial Decision Making and Optimization:

- **Integer Linear Programming** (Operational Research) → it is modeling with linear inequalities and computed through numerical calculations. Its focus is on objective functions and optimality and works through bounding and subsequent elimination of suboptimal values from domains. To achieve this, it exploits global structure, through relaxation, cutting planes and duality theory
- **Constraint Programming** (Artificial Intelligence) → I have a rich language for modeling and search procedures. Works with logical processing and focuses on constraints and feasibility. It works through propagation, which is the elimination of unfeasible values from the domains. It exploits local structure, with the domain reductions being based on the individual constraints.

The strength of CP lies in irregular problems, that contain messy constraints non-linear in nature and multiple disjunctions which result in poor information returned by a linear relaxation of the problem. As for optimality, CP has no special focus on this, while ILP scales up on loosely constrained optimization problems and Heuristic Search (HS) is effective in finding quickly good-quality solutions. The best optimality approaches are often hybrids of these three methods, with CP suitable framework for hybridization.

Overview of CP

The Constraint Solver finds a solution through a search algorithm that enumerates all possible variable-value combinations via a systematic backtracking tree search. You could use simply this to solve the problem, though it could be dimensionally unfeasible (see the video example). During search, examines the constraints to remove inconsistent values from the domains of the future (unassigned) variables → Constraint Propagation. Shrinks the domain of the future variables.



1. Modeling → the user expresses the problem, in the form of the Constraint Store, which contains all the constraints related to the problem. These constraints are usually dependent from each other, via common variables (some variables are involved in more than one constraint).
2. Search → the solver uses a backtracking tree search algorithm to guess a value for each variable
3. Propagation → the solver uses algorithms to examine each constraint to reduce the domains of the future variables.
4. Search Heuristics → the solver exploits the current search state and some problem specific knowledge to guide the search and supposedly quicken the process.

The model has therefore a dual role: it is a way to express our problem (captures the combinatorial substructures), but also influences the search space, by initiating the

propagation (enables the solver to reduce the search space). Constraints act as propagation algorithms and variables' domains as communication mechanisms.

Search Decision and Propagation are interleaved and one influences the other.

User declaratively models the problem and an underlying solver returns a solution through his own default search. Modeling is still critical (to achieve a strong propagation, one must use advanced modeling techniques). Plus, it's necessary to program the search strategy to obtain results.

Example: Eight Number Puzzle

See the material on Virtuale for the solution and information

As we can see from the example, from a bad choice of variables comes a bad assignment of values; hence, a good heuristic choice is very important. But sometimes, good heuristics are not possible; therefore, we should apply stronger forms of propagation. And this can come from better modeling.

Modeling in Constraint Programming

@February 24, 2023

Modeling directly affects what's going on in Propagation (see previous lecture).

A Constraint Satisfaction Problem (CSP) is a triple <X, D, C> where:

- X is the set of decision variables $\{X_1, ..., X_n\}$
- D is a non-binary and finite (not always) set of domains $\{D_1, ..., D_n\}$ for X:
  - D_i is a set of possible values for the variable X_i
- C is the set of constraints $\{C_1, ..., C_m\}$ → every constraint is a relation over a subset of the variables $X_j, ..., X_k$, denoted as $C_i(X_j, ..., X_k)$. This relation is a subset of the cartesian product of the domains (contains only the allowed tuples): $C_i \subseteq D(X_j) \times ... \times D(X_k)$

A solution is an assignment of values to all the variables which satisfies all the constraints simultaneously

Optimization problems are CSP enhanced with an optimization criterion to judge which is the better solution (e.g. minimum cost, shortest distance…). So formally it is a quadruple <X,D,C,f>:

- f is the formalization of the optimization criterion as an objective variable. The goal is to minimize f (or maximize -f, which is exactly the same)

Some Examples

The N-Queens Problem

We have to place N queens on an NxN board so that the queens do not attack each other.

We can choose to have N variable, which represent the placement of the queen in every row (in every row I know I need to have a queen). The domains are the columns (which column in the row is the queen's chosen place).
Constraint formalization: "They do not attack each other":

- We don't need a constraint for row attack → separate variables for each row. Our domain and variables choices can already address some constraints
- Column attack → global constraint alldifferent
- Diagonal attack → we look at the difference in column and row assignment and verify they're not equal (otherwise it's a violation).

It is not a good model (but the reasons, we will see them later).

Sudoku

Have a variable for each cell, 9x9, each with values 1…9. We should have a permutation of 1..9 on every row, every column (all different constraints) and 3x3 boxes (keep in mind that all these constraints influence each other). Plus we have given numbers.

Task Scheduling

An example can be timetables.
Schedule n tasks on a machine, in time D, by obeying the temporal and precedence constraints. Every task has its processing time, cannot start before its release date and must end before the deadline. Cannot overlap and must respect precedence

We want to decide the Starting Time (we have fixed processing times - which means we can infer the ending time).

- Respect of release date and deadline: $\forall i \in \{1, ..., n\}, r_i \leq S_i \leq d_i - p_i$
- No overlap in time: global function noOverlap (the tools give us some functions to help with modeling)
- Precedence: $S_i + p_i \leq S_j$ for each pair of tasks $t_i \rightarrow t_j$

Optimal Map Coloring

This is an Optimization Problem. We're given a map/graph and we want to color the regions, ensuring we never assign the same color to neighboring regions.

Ensure that the code is general enough to model all instances (the instances are not necessarily the objective).

Variables and Domains

The classical: one simple unique variable, that can be binary, integer, continuous. In some modeling language variables may take a value from any finite set you specify. We can also have Set Variables (the Domain is a set of sets) or Activity Variables (scheduling applications).

Constraints

Declarative (invariant) relations among objects. Intensional representation (mathematical relation). We have also an extensional representation that follows directly from the definition (it is the collection of all the allowed combinations). This is useful if the mathematical representation is not possible/difficult, but generally can be inconvenient and inefficient with large domains.

As a little side-note: the disjunction does not propagate well (see future lectures). In general, using global constraints instead of formalizing them yourself gives a usually better search performance (you don't work pairwise but with all the variables at once). Plus, use SAT for logical relations.

Meta-Constraints are constraints inside constraints (e.g. $\sum_i (X_i > t_i) \leq 5$ - I want at most 5 variables that satisfy the constraints). The internal constraint is treated as boolean (1 or 0). Used when there is no solution that satisfies all the constraints and so we need to relax some constraints to solve the problem.

CP is a rich language.

The order of imposition doesn't matter (due to the relation properties). It is non-directional: a constraint between X and Y can be used to infer domain information on Y given domain information on X and viceversa. Rarely independent, due to shared variables which serve as communication mechanism between constraints.

Be careful with modeling

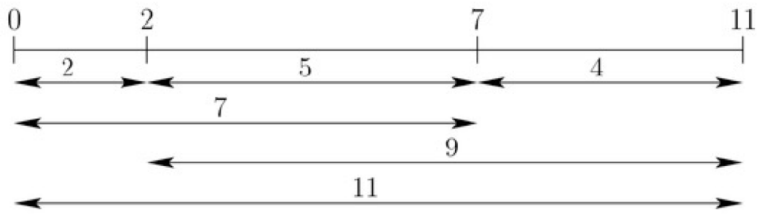
Choosing the variables and domains is crucial, because the number of variables and the size of domains decide the search space size (exponential growth). Also the choice of constraints

is important, because it defines how the search space can be reduced (propagation) and also how search can be guided (heuristics).

which variables shall I choose?
which constraints shall I enforce?
can I exploit any global constraints?
do I need any auxiliary variables?
are some constraints redundant, therefore can be avoided?
are there any implied constraints?
can symmetry be eliminated?
are there any dual viewpoints?
among alternative models, which one shall I prefer?

Example: Golomb Ruler

Optimization problem about placing m marks on a ruler, such that the distance between each pair of marks is different and the length of the ruler is minimum (largest order: 27). Used in radio astronomy and information theory.



Non-optimal solution to the Golomb ruler Problem

Naive Model

- There's no given this time. An example for the domain: $\{0, 1, ..., 2^{(m-1)}\}$ → ensures we can find at least one solution. The variable X_i represents the position of the i^{th} mark.
- Constraints → iterate over all variables and check pairwise
- Objective function is to minimize(max([$X_1, ..., X_m$])) - we need the max because we don't know which is the highest mark.

The model is problematic, with many domains and Quartic $O(m^4)$ Quaternary constraints. It is already evident that it won't propagate well (from the presence of the absolute values)

Better Model

Use of auxiliary variables: new variables I introduce into the model as a helping hand (impact on complexity of the problem) to model some constraints e/o express some constraints so that they lead to significantly better propagations.

In this case, I introduce for all $i < j, D_{ij} = |X_i - X_j|$

As the constraint on distances becomes global (now I can use allDifferent), I reduce complexity and I reach a Quadratic $O(m^2)$ ternary constraints.

I can also add Implied Constraints: constraints you have in the model that can be logically implied by the constraints but the solver cannot use it (they cannot see it by themselves). So by showing them, you can significantly improve the propagation (computationally significant), without changing the set of solutions (semantically redundant)

The implied constraint is that the variables are all different (descends from the difference between distances)

There can also be redundant constraints

Symmetry in CSPs

Creates many symmetrically equivalent search states → other search states that will be equal to a given one. It is very bad, because we have a huge search state and it is useless to explore symmetrical branches, there's no information. It is necessary to break the symmetry to ensure that search gives informative results.

- **Variable Symmetry** (the one in the Golomb ruler) → permutation π of the variable indices s.t. for each feasible assignment, we can rearrange the variables according to π and obtain another feasible assignment (same for unfeasible). The operation is a permutation of the variable assignments. I essentially identify the symmetries and introduce new constraints to address the breaking of the symmetry.
- **Value Symmetry** → permutation π of the values s.t. for each feasible assignment, we can rearrange the values according to π and obtain another feasible assignment (same for unfeasible).
  The two solutions I find are one the reverse of the other in terms of distances in the Golomb ruler. More complex to identify and same issues as the other symmetry.

I can also have compositions of variable and value symmetries (something that happens in the Golomb Ruler).

Symmetry Breaking Constraints

They reduce the set of solutions and search space. They are not implied by the constraints defining the problem. A common technique is to impose an ordering to avoid permutations, making sure that at least one solution from each set of symmetrically equivalent solutions must remain (otherwise I would lose solutions).

Golomb Ruler: Improved Model

- Symmetry Breaking Constraints → $X_1 < ... < X_m$, with $X_1 = 0$ and $D_{12} < D_{(m-1)m}$ (because it is about reversing the distance order).

NB → Symmetry Breaking Constraints are PROBLEM-SPECIFIC!! Also, it is not necessary to break all symmetries (might overcomplicate)

- New Objective is to minimize X_m → We enforced an ordering now, so we know that the last variable will have the maximum value.

Symmetry breaking constraints allow us to simplify previously-individuated constraints: alldifferent becomes redundant (if it doesn't give computational significance, cut - nothing will change by having it or not).
Also now I have $\forall\ i < j, D_{ij} = X_j - X_i$ (I can get rid of the absolute value due to the ordering).

Also, I can enable new additional implied constraints.

Improved N-Queens Model

I can work on the Diagonal Attack constraint, which contains absolute values

$\forall\ i < j\ |X_i - X_j| \neq |i - j| \equiv X_i - X_j \neq i - j \text{ and } X_i - X_j \neq j - i \text{ and } X_j - X_i \neq i - j \text{ and } X_j - X_i \neq j - i \equiv X_i - i \neq X_j - j \text{ and } X_i + i \neq X_j + j$

At this point, I can use global constraints:

$\text{allDifferent}([X_1 - 1, ..., X_n - n])$

$\text{allDifferent}([X_1 + 1, ..., X_n + n])$

Dual Viewpoint

Viewing a problem from different perspectives may result in different models, with the same set of solution and generally a different representation, which means that the search space can have a different size.

Example: N-Queens Models

We don't have complete permutations, but we have 8 geometric symmetries. In this case, you can't impose an ordering. To break the symmetry, change model:

Board Representation

Represent the board with $n \times n$ boolean variables (the queen is there or no). Constraints become: $\sum B_{ij} = 1$ on all rows and columns and $\sum B_{ij} \leq 1$ on all variables

In this case I can break symmetry, by flattening the 2D matrix to a single sequence of variables by appending each row one after another and identify the permutations through the Lexicographic Ordering Constraint

Lexicographic Ordering Constraint

Requires a sequence of variables to be lexicographically less than or equal to another sequence of variables.

$\text{lex} \leq ([Y_1, ..., Y_k], [Z_1, ..., Z_k]) \iff Y_1 \leq Z_1 \wedge (Y_1 = Z_1 \rightarrow Y_2 \leq Z_2) \wedge ... \wedge (Y_1 = Z_1 \wedge ... \wedge Y_{k-1} = Z_{k-1} \rightarrow Y_k \leq Z_k)$

Choosing the Model

If you can’t find an all-around cool model, combine the pros of all so that you balance, using channeling constraints to maintain consistency.

Drop constraints that are redundant, channeling constraints will do the work for you.

Local Consistency, Constraint Propagation and Global Constraints

@March 2, 2023

Constraint Propagation is the examination of the constraints to remove inconsistent values from the domains of the future (still unassigned) variables → so I remove what cannot be part of a solution (general definition of consistency)

We have different notions of consistency according to how much inference we do (I might detect inconsistent assignments earlier). Local Consistency → form of inference which detect inconsistent partial assignments . Local because we examine just individual constraints. I work one constraint at a time. A partial assignment is where only some variables but not all are assigned. Popular local consistencies are domain-based, which means they detect inconsistencies of the type $X_i = j$. If I’m detecting that the assignment is inconsistent, I remove j from the domain $D(X_i)$ by propagation.

Generalized Arc Consistency (GAC)

a.k.a. domain consistency (because it is domain-based) or hyper-arc consistency. The constraint Cdefined on k variables $C(X_1, ..., X_k)$ is a subset of the cartesian product of

the variables' domains ($C \subseteq D(X_1) \times ... \times D(X_k)$). So every tuple $(d_1, ..., d_k)$ that belong to the subset is called support of C (all these values all together satisfy the constraint).

I have $C(X_1, ..., X_k)$ is GAC $\iff$ for all X_i in the set of variables $\{X_1, ..., X_k\}$, for all $v \in D(X_i)$, v belongs to a support. I call it Arc Consistency is if the variables are just 2 ($k = 2$). I have that a Constraint Satisfaction Problem is GAC if all its constraints are GAC.

A local consistency notion defines the properties that a constraint C must satisfy after constraint propagation. The operational behavior is left open (any algorithm could do); the only requirement is to achieve the required property on C. Constraint Propagation is the act of removing the values that are inconsistent based on the local consistency chosen. It maintains a specific level of consistency by removing values.

The level of consistency depends on C.

Bound Consistency (BC)

Weaker than GAC (can't detect all the inconsistencies GAC detects, but it's cheaper).

It is defined for totally ordered domains, in which I can identify a minimum and a maximum (e.g. Integers). I relax the domains: I assume that $D(X_i)$ ranges $[\min(X_i) ... \max(X_i)]$. At this point I can define the bound support. The bound support is a tuple $(d_1, ..., d_k) \in C$ where $d_i \in [\min(X_i) ... \max(X_i)]$

$C(X_1, ..., X_k)$ is Bound Consistent if for all the variables X_i in $\{X_1, ..., X_k\}$, both $\min(X_i)$ and $\max(X_i)$ belong to a bound support.

We need to search more, as bound consistency might not detect all GAC inconsistencies in general, but might be easier to find a bound support than a regular support. It is often cheaper to achieve, and of interest in arithmetic constraints defined on integer variables with large domains. Achieving BC is enough to achieve GAC for monotonic constraints.

Multiple Constraints

When solving a CSP with multiple constraints, we have reciprocal influence between the constraints due to shared variables. Constraints can be propagated more than once, because another constraint might have changed a domain, thus triggering the propagation.

Some events may trigger a propagation according to the algorithm chosen.

In the end, propagation reaches a fixed-point and all the constraints reach a level of consistency. The whole process is referred as constraint propagation.

In general, it may not be enough to remove inconsistent values from the domains once. A propagation algorithm must wake up when necessary, otherwise it may not achieve the desired local consistency property.

Complexity of Propagation Algorithms

Assuming that the domain size is a constant $|D(X_i)| = d$. On a binary constant, one AC propagation on a $C(X_1, X_2)$ costs $O(d^2)$. Which is extremely expensive.

We can use Specialized Propagation, which is specific to a given constraint. Can exploit the semantics of the constraints to gain an advantage and it is potentially much more efficient than a general propagation approach. The issue is that has a limited use and it is not always easy to develop. It is worth developing for recurring constraints.

Use as much global constraints as possible, as they capture complex, non-binary and recurring combinatorial substructures arising in a variety of applications. Plus, they embed specialized propagation, which exploits the substructure.

They have both modeling benefits (they reduce the gap between the problem statement and the model and may allow the expression of constraints that are otherwise not possible to state using primitive constraints) and solving benefits (as they have both a strong inference in propagation and efficient propagation).

Groups:

- Counting → (e.g. the allDifferent constraint): they restrict the number of variables satisfying a condition or the number of times the values are taken.
    - `allDifferent([X1,...,Xk])` $\iff X_i \neq X_j$ for $i < j \in \{1, ...k\}$. If $|D(X_i)| = k$ it is a permutation constraint. Can be useful in a variety of contexts, that go from puzzles to tournament scheduling and configuration problems.
    - Global Cardinality Constraint → constrains the number of times each value is taken by the variables. Can be useful for example in resource allocation `gcc([X1,…,Xk], [v1,…,vm],[O1,…, Om])` $\iff \forall j \in \{1, ..., m\}, O_j = |\{X_i | X_i = v_j, 1 \leq i \leq k\}|$
    - Among Constraint → useful in sequencing problems. Constrains the number of variables taking certain values `among([X1,…, Xk], v, l, u)` $\iff l \geq |\{i | X_i \in v, 1 \leq i \leq k\}| \leq u$
- Sequencing Constraints → they ensure that a sequence of variables obey certain patterns.
    - Sequence Constraint → constrains the number of values taken from a given set in any subsequence of q variables. It is also known as `amongseq` constraint. Can be

useful in rostering and production lines.
`sequence(l, u, q, [X1,...,Xk], v)` $\iff$ `among([Xi,Xi+1, ..., Xi+q-1, v, l, u)` `for` $1 \leq i \leq k+q-1$

- Scheduling Constraints → help us scheduling tasks with respective release times, duration and deadlines, using limited resources in a time interval D
  - noOverlap/Disjunctive Resource Constraint → it requires that tasks do not overlap in time. Useful when a resource can execute at most one task at a time. Given tasks $t_1, ..., t_k$, each associated with a start time S_i and a duration D_i
    `disjunctive([S1,..., Sk], [D1,..., Dk])` $\iff \forall i < j, (S_i + D_i \leq S_j) \vee (S_j + D_j \leq S_i)$
  - Cumulative Resource Constraint → it constrains the usage of a shared resource. Useful when a fixed-capacity resource can execute multiple tasks at a time. Given tasks $t_1, ..., t_k$, each associated with a start time S_i, duration D_i, resource requirement R_i and a resource with capacity C
    `cumulative([S1,...,Sk],[D1,...,Dk],[R1,..., Rk], C)` $\iff \sum_{i \mid S_i \leq u \leq S_i + D_i} R_i \leq C \, \forall u \in D$
- Ordering Constraints → Enforce an ordering between the variables or the values
  - Lexicographic Ordering Constraint → the one we used to break symmetry. It requires a sequence of variables to be lexicographically less than or equal to another sequence of variables.
- Balancing
- Distance
- Packing
- Graph-based
- …

Rules for notations → variables use Uppercase, constants use lowercase.

Specialized Propagation for Global Constraints

Constraint Decomposition

Decompose a global constraint into smaller and simpler constraints that have a known propagation algorithm. Propagating each of the constraints gives a propagation algorithm for the original global constraint.

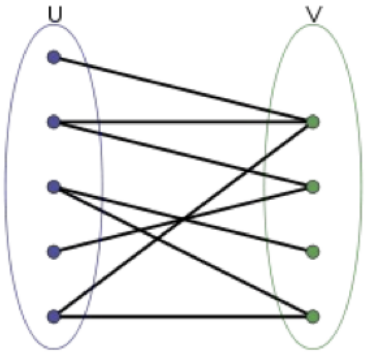
Most of the time doesn't prove to be effective. Often, GAC on the original constraint is stronger than (G)AC on the constraints in the decomposition

Dedicated Propagation Algorithms

Dedicated algorithms provide effective and efficient propagation. Often GAC is maintained in polynomial time, many more inconsistent values are detected compared to the decompositions and computation is done incrementally.

Jean-Charles Régin, "A Filtering Algorithm for Constraints of Difference in CSPs". It maintains GAC on `allDifferent([X1,..., Xk])` and runs in polynomial time. It establishes a relation between the solution of the constraint and the properties of a graph; more precisely, maximal matching in a bipartite graph. I can obtain a similar algorithm using flow theory.

A bipartite graph is a graph whose vertices are divided into two disjoint sets U and V such that every edge connects a vertex in U to one in V.



A matching in a graph is a subset of its edges such that no two edges have a node in common and maximal matching is the largest possible matching. In a bipartite graph, maximal matching covers one set of nodes.

Given a bipartite graph G constructed between the variables $[X_1, ..., X_k]$ and their possible values (variable-value graph), an assignment of values to the variables is a solution if and only if it corresponds to a maximal matching in G (by definition, it covers all the variables). By computing all maximal matchings then, we can find the consistent partial assignments.

It is though inefficient to compute all maximal matchings naïvely; we have theoretical results from matching theory to compute them efficiently: one maximal matching can describe all maximal matchings.

GAC may as well be NP-hard. Algorithms that maintain weaker consistencies are of interest. And if it is difficult to decompose a constraint or build an efficient and effective dedicated algorithm, we can consult global constraints for generic purposes (e.g. table constraints).

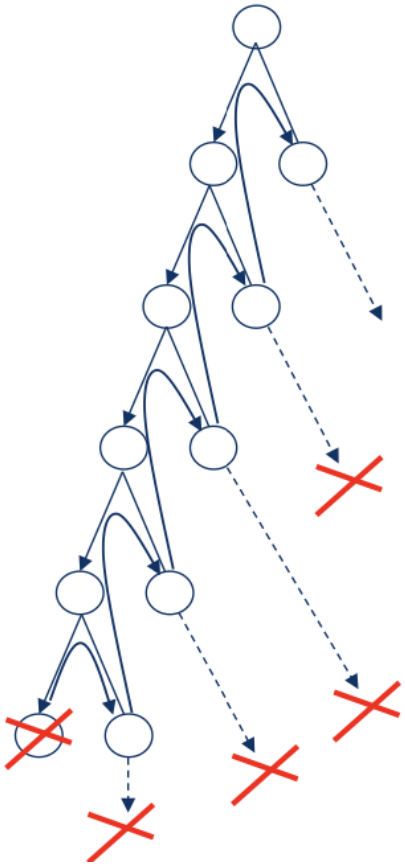
Search in Constraint Programming

@March 3, 2023

Look for possible solutions in the space of possibilities. This process is about enumerating all possible combinations via systematic backtracking tree search (guess one-at-a-time value for each variable). The solver has a different approach: the Left Branch contains the guess, the Right Branch is inferred from the Left Branch. The other method is not used as it “commits” too much to the values. Solvers might also do domain partitioning into subsets.

Backtracking Tree Search (BTS)

We instantiate variables one by one (sequentially). By default, it is a depth-first traversal. If there’s no constraint propagation mechanism, whenever all the variables of a constraint is instantiated, it checks the validity of the constraint, and if it fails, it falls back to the previous decision (closest previous branch) - chronological backtracking. It is a systematic search, because it will eventually find a solution or proves the unsatisfiability. Exponential complexity $O(d^n)$. By default, it is Generate&Test.



When I have a constraint solver, after each assignment, it examines the constraints to remove inconsistent values from the domains of the future still unassigned variables, thus shrinking future domains (if a variable has no longer a domain - null domain - the process backtracks) → all constraints are propagated before search, and all the necessary ones after each search decision. This might even help find that a constraint makes it so the problem has no solution at all.

CP integrates the propagation mechanisms to reduce domains during search.

Propagation does not make the problem any less difficult, but it reduces the search tree size.

Search Heuristics

Guide the search decisions (which variable/value to tackle next). They can be problem specific (we can derive them from the properties of the problems) or generic. Static (you know the order before search) or Dynamic (we decide the variable to take next during the search). Static Heuristics are cheaper, because you do a calculation at the beginning once and for all. For dynamic, you have to spend a little time computing. During search, though, as you can see the evolution of the domains space, you can obtain a much clearer idea of which variable to take next.

During search we choose a variable and progress. If the problem is feasible, it makes sense to choose values that are likely to give us a solution. There's though no guarantee that the problem has a solution. Even if the problem can have a solution, you can enter an infeasible sub-problem (you have to fully explore it to establish you need backtracking) - typical in CP

Fail-First Principle: try first where you're most likely to fail. Objective is to backtrack as soon as possible. This way, you can maximize the propagation. As you don't actually know if a CSP is feasible or not, you need a trade-off:

- choose next the variable that is most likely to cause a failure
- choose next the variable that is most likely to be part of a solution (least constrained value)

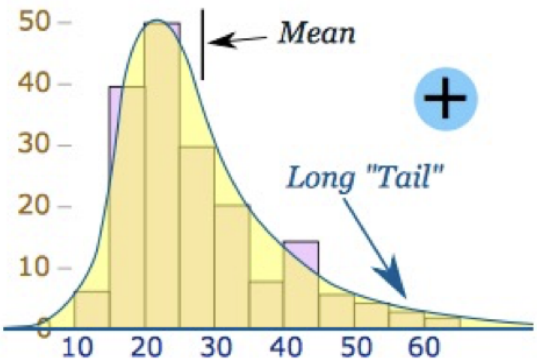
The main focus will be on Variable Ordering Heuristics (VOHs) → to backtrack from an infeasible sub-problem, we need to explore all the values in the domain of a variable.

Generic Dynamic VOHs based on FF

- dom → Choose next the variable with minimum domain size. The objective is to minimize the search tree size
- deg → Choose the variable that's involved in the highest number of constraints (e.g. the middle section of the game of the first lecture). The objective is to maximize the constraint propagation.
- You can combine the two approaches and minimize the ratio → $\min \frac{\mathrm{dom}}{\mathrm{deg}}$
- Weighted Degree Heuristic → $\mathrm{domWdeg}$:
  Every variable is instantiated with a weight (initially set to 1). During the propagation of a constraint c, its weight $w(c)$ is incremented by 1 if the constraint fails. Sum the weights of the constraints, because not all constraints are the same (some are more

difficult than others) $w(X_i) = \sum_{c|X_i \in X(c)} w(c)$. We will choose the variable X_i with $\min \frac{\text{dom}(X_i)}{w(X_i)}$

A problem can be solved in different ways ⇒ different instances. We observe that some of these instances take an incredibly long time. We observe a heavy tail behavior (long tail).



If we play with heuristics, this instance might be easier or more difficult ⇒ it is not a characteristic of the instance itself, more like an issue with the combination.

It happens (intuitively) because when you make an early error in decision, you can get stuck in a sub-tree. It is seemingly random, something we cannot control which is based on luck.

- Randomization → add some randomized parameter in search and sometime will pick variables randomly or follow the heuristics and choose values randomly. With pure luck you can obtain good sub-trees. In general, the solver will explore the same tree, but it will never explore two identical subproblems in the same way
- Restarting → when you get stuck in an unfeasible sub-tree and you spend a threshold of resources (search steps/visited nodes), restart all. The idea is to restart and search differently, e.g. changing the heuristic or adding randomization.

By combining the two instances, you can reduce the variance in solvers

domWdeg works well with restart, because you can use the collected fail counts in the subsequent runs. Can be combined with random choice of values and be effective

Restart Strategies

- **Constant** restart → after I use L resources
- **Geometric** restart → I have a parameter α and at every step I increase the time by multiplying the α to L. So it ends up being $L, \alpha L, \alpha^2 L, ...$
- **Luby** restart → s[i] is the ith iteration of the Luby Sequence: it repeats two copies of the sequence ending in 2^i before adding 2^{i+1}

Constraint Optimization Problems (COPs)

The mechanism is called “Branch & Bound Algorithm”. I solve a sequence of CSPs and each time a feasible solution is found, I introduce a bounding constraint to make sure that future solutions must be better than what I already have. I go on until I fail (I’m not able to improve it anymore).

It is included in CP solvers.

Complementary Strengths

Combining the strengths of CP and Heuristic Search. **CP** is rather generic and complete (we can search the whole tree). Focus on constraints and feasibility. We have a library and ways to control the search. Poor optimization with loose bounds (upper bound is usually set by CP, lower bound you can set - integrated methods they compute a linear relaxation of the problem to obtain automatically a lower bound).
Heuristic Search is another way to look at optimization that normally scales to large optimization problems, unlike CP. The objective is not to find the best solution, but to find a good-quality solution with easily expressible constraints. Neighborhood specific. Can be expensive if the initial solution is no quality.

They actually complement each other (one’s good when the other fails).

Large Neighborhood Search

Philosophy that takes the best of both worlds.

I define a generic and large neighborhood. Given a solution s: I fix part of the variables to the values they have in s (called fragment) and I relax the remaining variables. This way, I can explore the neighborhood with a complete optimization method like CP.

The exploration of a neighborhood is seen as the solution of a sub-problem and I can use propagation and advanced search techniques of CP to exhaustively and efficiently explore it. I can use CP also to find an initial solution.

Advantages: you can use propagation and advanced search techniques of CP (efficiency) and the neighborhood is easy and generic to define (LNS is therefore easier to develop than HS). Wins in scalability w.r.t. basic CP, as subproblems are typically much smaller, we can control the size (propagation works better in small domains) and the fixed variables reduce the domain sizes.

Search is a huge topic in CP and there are other forms of search.