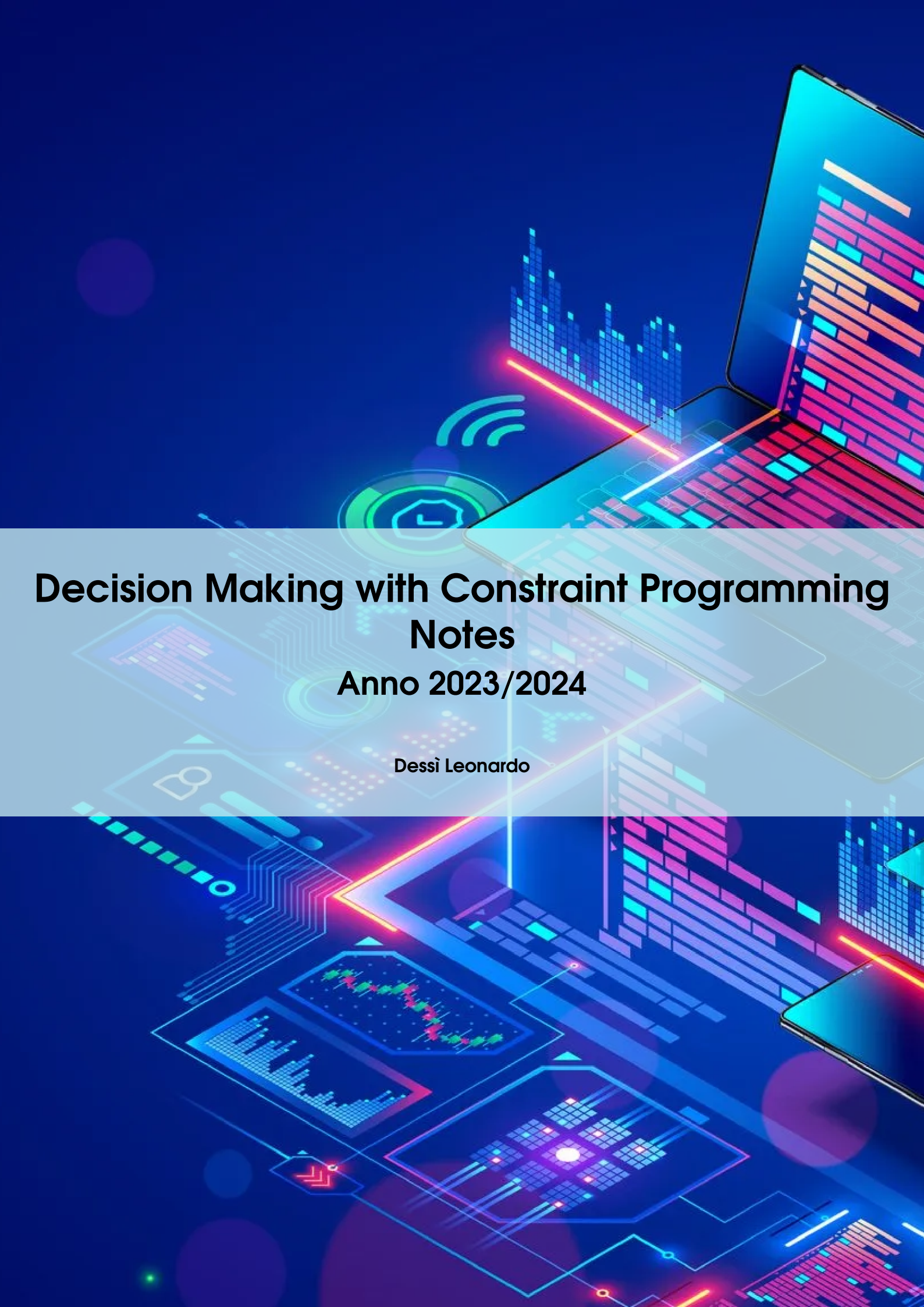
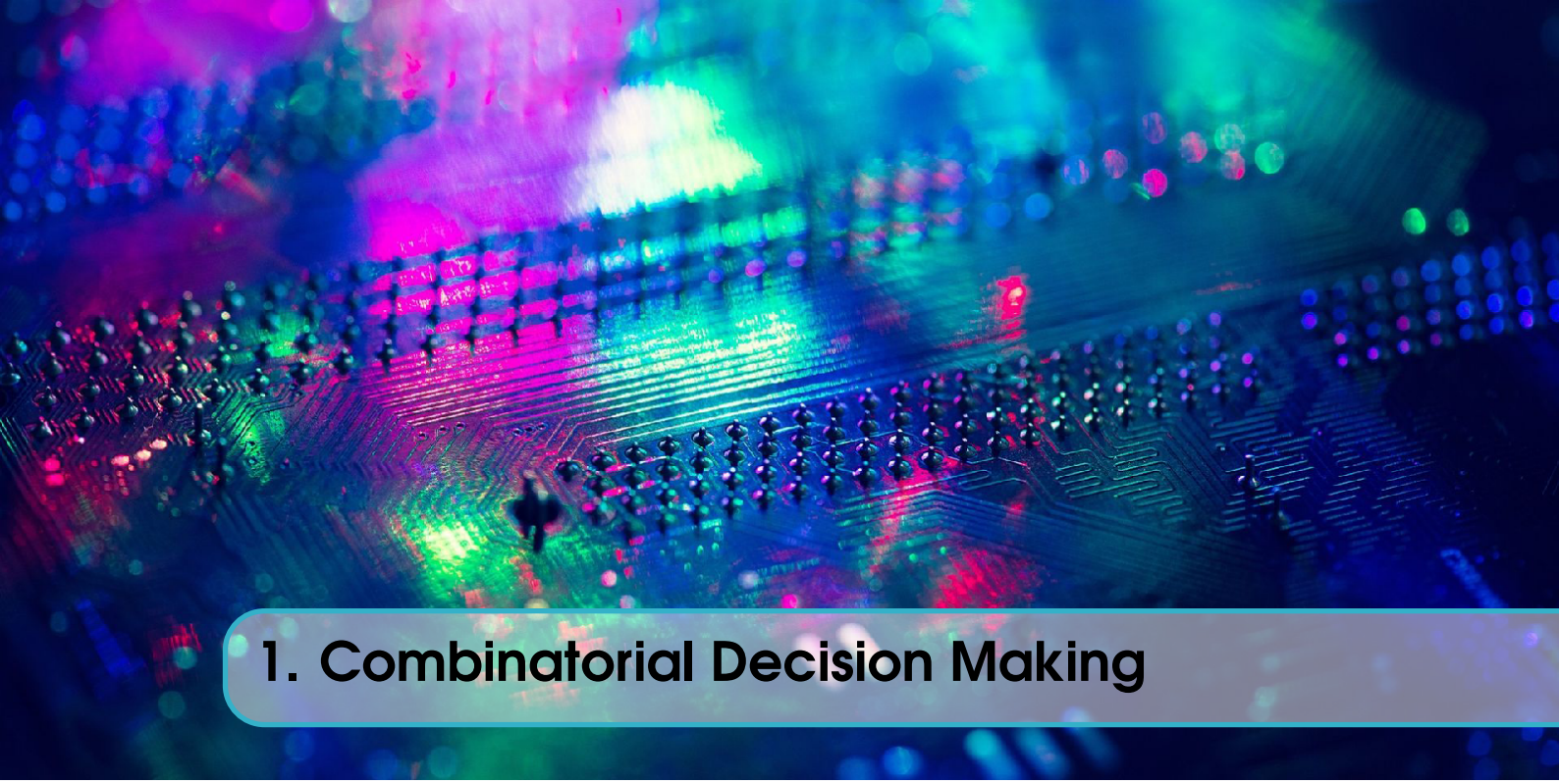


Decision Making with Constraint Programming Notes

Anno 2023/2024

Dessì Leonardo

Contents



1. Combinatorial Decision Making

1.1 Introduction

In the vast landscape of decision-making processes, we often encounter complex situations where choices must be made among a large number of possibilities, taking into account various constraints or restrictions. This is what we define as "**Combinatorial Decision Making**". Essentially, in these situations, we seek solutions that satisfy all imposed constraints.

Among these solutions, there may be one that is the best possible, the so-called "***optimal solution***", according to a specific objective criterion. This is a crucial aspect of combinatorial decision-making processes.

It is interesting to note that this type of decision can be called by various names depending on the context. We call it "***combinatorial optimization***" when seeking the best possible outcome. At the same time, if we focus only on finding solutions that meet all constraints, we call it "***constraint satisfaction/optimization***".

This type of decision-making is an integral part of our daily lives. We encounter it in industrial sectors, sciences, and even in our daily activities. It underlies transportation planning, activity scheduling, resource management, and much more.

However, there is an aspect we cannot ignore: combinatorial decision-making is computationally challenging. In general, it is known to be an "***NP-hard***" problem, meaning that there is no efficient algorithm to solve it in reasonable times for large instances of the problem. Therefore, solving these challenges requires the use of intelligent search methods.

It is important to emphasize that solving these problems is often an experimental process. There is no magic formula, but rather a **continuous experiment** to find valid solutions. And this is significant because finding good or optimal solutions can lead to time and cost savings and a reduction in environmental impact.

Various techniques address the combinatorial decision-making problem. **Integer Linear Programming** (ILP) is one such technique that seeks to translate the problem into an integer linear programming format. Then we have **Boolean SATisfiability**, **SAT Modulo Theories** (SMT), which focuses on boolean logic. Additionally, there are **heuristic search methods**, which use approximate strategies to solve the problem. Lastly, **Constraint Programming** (CP) is an approach based on a declarative description of constraints.

It is interesting to note how Constraint Programming (CP) has become increasingly popular. This field is a crucial part of artificial intelligence and receives contributions from universities, research centers, and companies worldwide. Companies like IBM, Google, Ericsson, Siemens, Renault, Oracle, Sap, Intel, and Tacton have successfully adopted CP. CP has become a preferred technology in the logistics, planning, and programming sectors.

Furthermore, we cannot overlook the fact that possessing CP skills can represent a significant asset in the job market. In a world increasingly oriented towards intelligent resource and decision management, CP is a skill that can make a difference.

■ **Example 1.1 — Covid-19 Test Scheduling.** To better understand how Constraint Programming (CP) is a powerful tool in solving complex problems, consider the case of Ocado Retail Ltd, one of the world's largest online grocery retailers.

Ocado Retail Ltd employs over 15,000 people, many of whom perform frontline roles such as warehouse packing, order delivery, and customer service in call centers. With the onset of the Covid-19 pandemic, the company made the decision to subject all its frontline employees to weekly tests, a crucial move to ensure the safety of workers and customers.

However, scheduling these weekly tests for such a large number of employees across multiple sites posed a formidable challenge. Attempting it manually proved nearly impossible given the numerous variables and constraints involved.

Ocado Retail Ltd's Data Science team tackled this challenge using a Constraint Programming (CP)-based solution. CP proved to be the key to successfully resolving this intricate situation. This solution enabled scheduling tests for up to 3,500 employees distributed across 4 different sites.

This example highlights how CP is a powerful tool capable of addressing real-world complex problems, such as scheduling Covid-19 tests in a company with a vast workforce. CP's ability to handle a large number of constraints and variables, providing optimized solutions, demonstrated its effectiveness in critical situations like this one. ■

1.2 Constraint Programming

Constraint Programming (CP) represents a **declarative programming paradigm** used to formulate and solve combinatorial optimization problems. In simpler terms, it is an approach to addressing situations where decisions must be made among a set of possibilities subject to constraints or restrictions. To do this, the user formalizes the decision problem through three key elements:

- **Decision Variables** (X_i): These represent the unknowns of the problem. For example, if planning a distribution, variables might be the routes of vehicles to be assigned.
- **Variable Domains** ($D(X_i) = v_j$): Here, we define the possible values each variable can take. For instance, possible destinations or departure times for vehicles.
- **Constraints between Variables** ($r(X_i, X_{i'})$): These represent relationships or restrictions between variables. For example, a constraint could be that two vehicles cannot share the same route.

A **constraint solver** is then used to find a solution to the model by assigning a value to each variable through a search algorithm. This process can lead to determining a valid solution to the problem or demonstrating that no solution exists.

1.2.1 Why Constraint Programming?

Choosing CP might seem similar to Integer Linear Programming (ILP), but it has distinct advantages:

1. **Rich Language for Constraint Expression**: CP offers a more intuitive and expressive language for expressing constraints and defining search procedures. This simplifies problem modeling.
2. **Rapid Prototyping**: CP allows rapid prototyping due to a variety of available constraints, speeding up solution development.
3. **Ease of Maintenance**: CP-based programs are easier to maintain and adapt to new requirements. Changes to constraints can be made more seamlessly.
4. **Extensibility**: CP enables easy control of search and experimentation with advanced strategies. This is useful for solving complex problems.
5. **Focus on Constraints and Feasibility**: CP focuses on constraints and verifying the feasibility of solutions. More constraints mean a reduction in the search space, making the problem more manageable.

Differences between ILP and CP:

Integer Linear Programming	Constraint Programming
Modeling with linear inequalities	Rich language of modeling and search procedures
Numerical computations	Logical reasoning
Focus on the objective function and optimality	Focus on constraints and feasibility
Bounding $\rightarrow$ elimination of suboptimal assignments	**Propagation** $\rightarrow$ elimination of infeasible assignments
Exploits global structure	Exploits local structure
Relaxations, Cutting Planes, and duality theory	Domain reductions based on individual constraints

Strengths of Constraint Programming include:

- **Success in Irregular Problems**: CP proves its value, especially in irregular problems such as scheduling, activity sequencing, resource allocation, and shift creation.
- **Handling Complex Constraints**: CP excels in managing "messy" and non-linear constraints, which would be challenging with other approaches. These constraints can be intricate and often involve multiple disjunctions, making it difficult to obtain accurate information from a simple linear relationship.

While CP's weaknesses and opportunities are:

1. **Optimality**: One of CP's weaknesses is that it does not specifically focus on the objective function and optimality of solutions. In contrast, Integer Linear Programming (ILP) has shown effectiveness in less constrained optimization problems. Additionally, heuristic search methods often quickly find high-quality solutions.
2. The best approaches to optimality are often **hybrids**: The most effective optimal solutions often derive from a hybrid approach, combining elements of CP, ILP, and heuristic search algorithms. In other words, hybridizing these techniques can lead to more satisfying results.

1.2.2 Overview of Constraint Programming

The CP solver enumerates all possible variable-value combinations through a systematic tree-based search strategy with backtracking. Here, each variable is assigned a value, a kind of "guess". As it

progresses in its search, it examines constraints, trying to **eliminate incompatible values from the domains of unexplored variables**. In other words, it narrows down the domains of future variables, making the search path more targeted (propagation).

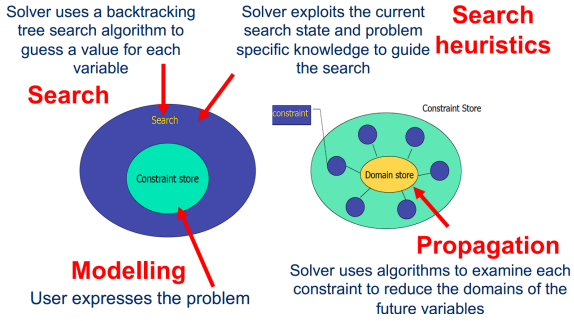


Figure 1.1: Constraint Programming

An interesting aspect of the process is the dual role of the model. On one hand, the model captures the combinatorial structures of the problem, providing the solver with a map of the territory to explore. On the other hand, the model helps the solver reduce the search space. This happens through constraints acting as propagation algorithms, pushing towards the reduction of variable domains. Variables, in turn, act as a communication mechanism, transmitting crucial information for the search.

Another important aspect to note is that search and propagation are two processes that occur interchangeably. Propagation occurs when constraints are examined to remove incompatible values from the domains of variables not yet explored. This cycle repeats continuously, allowing the solver to gradually refine the solution.

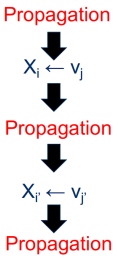


Figure 1.2: Search and Propagation

In theory, CP is expected to be a declarative programming paradigm. In other words, the system's user should be able to declaratively model the problem, expressing what they want to achieve, and the solver should do the rest, autonomously determining the solution through a predefined search.

However, in reality, modeling is crucial. Often, to achieve optimal results, it is necessary to resort to **advanced modeling techniques**, as the default behavior of the solver may not be sufficient. Users must be able to guide the search strategy, deciding which **search algorithm** to use and which heuristics to adopt.

■ **Example 1.2 — Puzzle.** In this example, we see how the choice of one model over another influences efficiency in problem resolution. We have a graph where we need to place numbers from 1 to 8 in such a way that:

- each number appears only once
- there are no connected nodes with consecutive numbers

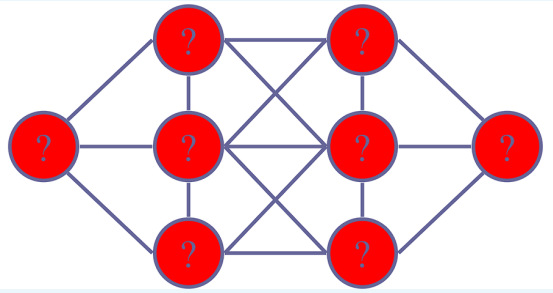
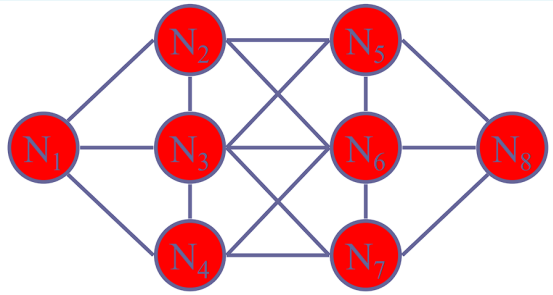


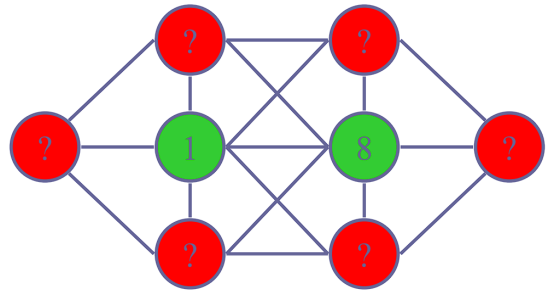
Figure 1.3: Puzzle

We can model the problem as follows:

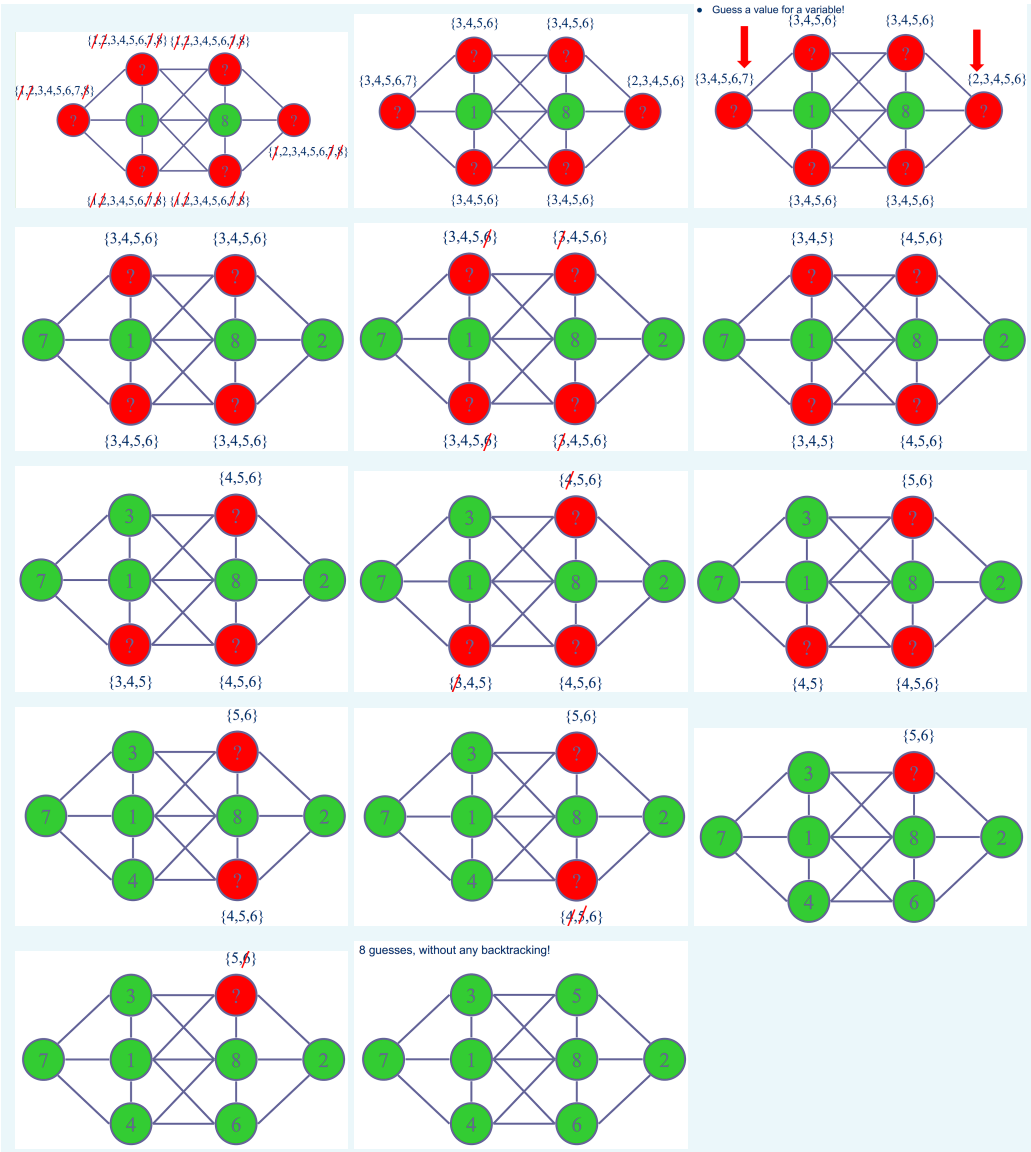
- **Variables**: $N_1, \ldots, N_8$;
- **Domains**: the set of values $\{1, 2, 3, 4, 5, 6, 7, 8\}$ that $N_1, \ldots, N_8$ can take;
- **Constraints**:
  - $\forall i < j \mid N_i$ and N_j are adjacent. $|N_i - N_j| > 1$
  - $\forall i < j.\ N_i \neq N_j$



Now, through a heuristic, we can make 2 initial assignments, i.e., the safest 1 and 8 since they are the numbers that, in this case, have fewer successors or predecessors, and place them in positions with the most connections.



Now let's see how propagation progresses from this initial state.



As can be seen, thanks to the heuristic, we found a solution in only 8 assignments without ever backtracking. In the professor's virtual animation, it can be seen how without using a heuristic and without the initial assignment of 1 and 8, the solver fails numerous times. ■

From this example, it can be deduced that:

- A poor choice of variables and variable assignments can lead to blind alleys and increase computation time.
- Having a good heuristic is important. Here arises the question: is it always possible to make wise heuristic choices? The answer is "yes" and "no." Good heuristics are crucial, but there is no guarantee of always finding them. Therefore, a more robust approach is needed to address the problem.

So what can be done is to apply a more advanced form of propagation during the search. In practice, this means that during the execution of the resolution process, more powerful strategies can be used to explore possibilities and refine solutions.

Another aspect to consider is the quality of the **modeling**. A well-constructed model can generate a more robust form of propagation, making the resolution process more effective.

■ **Example 1.3 — Improved Model.** Another thing that can be done to improve the model is to replace non-global constraints with global constraints where possible. For example, the model from the previous example becomes:

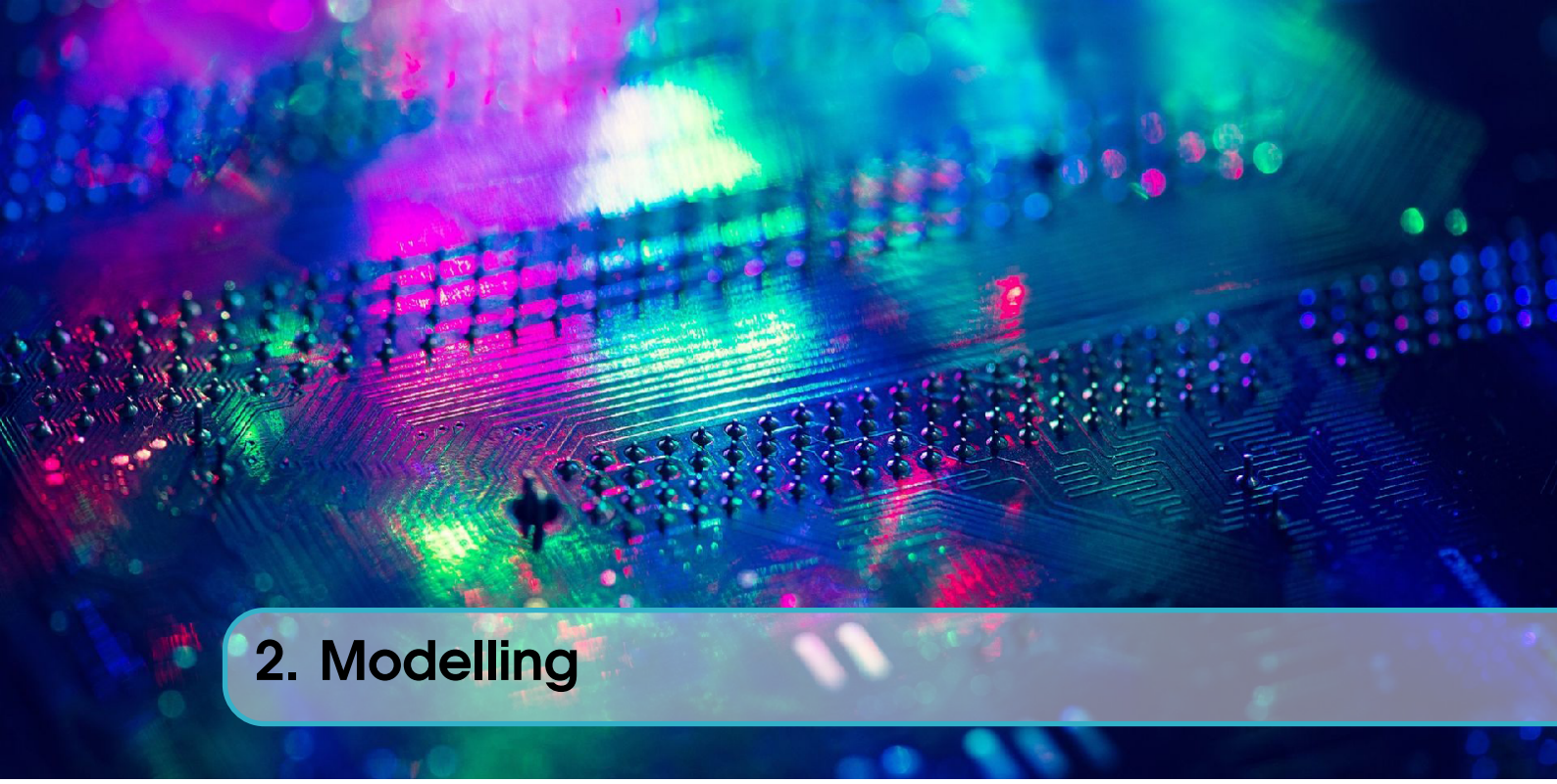
- **Variables**: $N_1, \ldots, N_8$;
- **Domains**: the set of values $\{1, 2, 3, 4, 5, 6, 7, 8\}$ that $N_1, \ldots, N_8$ can take;
- **Constraints**:
  - $\forall i < j \mid N_i$ and N_j are adjacent. $|N_i - N_j| > 1$
  - $\texttt{alldifferent}(N_1, N_2, N_3, N_4, N_5, N_6, N_7, N_8)$

■

In conclusion, in Constraint Programming:

- The user models the problem.
- A search-based solver returns a solution.
- The user must program a strategy to search for a solution with a search algorithm, heuristic, or else the resolution process can be inefficient.

Modeling



2. Modelling

When engaging in constraint programming, users embark on the process of formally articulating a decision problem. This involves defining the elements of the decision, namely the unknowns, the potential values for these unknowns, and the relationships between them.

To break it down, the unknowns in the decision problem are represented as **decision variables**, denoted as X_i. These variables encompass the various aspects of the decision that the user aims to resolve. Simultaneously, the potential values these variables can take are captured by their respective **domains**, symbolized as $D(X_i)$. Each D_i signifies a set of conceivable values for its corresponding variable X_i, and it's worth noting that these domains are generally non-binary and are assumed to have a finite range.

Furthermore, the interconnections and dependencies among the decision variables are expressed through **constraints**, represented as $r(X_i, X_{i'})$. Each constraint C_i is essentially a relation over a subset of the decision variables, denoted as $C_i(X_j, ..., X_k)$. It specifies the acceptable combinations of values that these variables can assume, expressed as $C_i \subseteq D(X_j) \times \cdots \times D(X_k)$.

2.1 Constraint Satisfaction Problems

In the realm of constraint satisfaction problems (CSP), this modeling process is encapsulated as a triple $<X, D, C>$. Here,

- X constitutes the set of decision variables $X_1, \ldots, X_n$,
- D encompasses the set of domains $D_1, \ldots, D_n$ for these variables, and
- C comprises the set of constraints $C_1, \ldots, C_m$.

In essence, the **objective in constraint programming** is to **find a solution** to the CSP, which entails assigning values to the decision variables in a manner **that satisfies all specified constraints simultaneously**. In other words, a solution to the CSP is a **configuration of values** for the variables that adheres to the predefined constraints, rendering it a feasible and valid resolution to the decision problem at hand.

■ **Example 2.1** Let's delve into some practical examples to illustrate these concepts. Consider a scenario with two variables, X_1 and X_2, where the domains for both are limited to the range of 1 to 3. The constraints, denoted as C_1 and C_2, define specific combinations of values these variables can take. Solutions to this problem could be $X_1 = 1, X_2 = 2$ or $X_1 = 2, X_2 = 3$. The problem is defined as follows:

- Variables
  - $X = \{X_1, X_2\}$
- Domains
  - $D(X_1) = [1..3]$
  - $D(X_2) = [1..3]$
- Constraints
  - $C_1(X_1, X_2) = \{(1, 2), (1, 3), (2, 3)\}$
  - $C_2(X_1, X_2) = \{(1, 2), (2, 3)\}$
- Solutions
  - $X_1 = 1,\ X_2 = 2$
  - $X_1 = 2,\ X_2 = 3$ ■

■ **Example 2.2** In a three-variable case, X_1, X_2, and X_3 all share the same domain of 1, 3, 5. Constraints, such as $X_1 + X_2 \leq X_3$ and ensuring all values are different, further shape the feasible solutions. For instance, one solution could be $X_1 = 1, X_2 = 3, X_3 = 5$, and another could be $X_1 = 3, X_2 = 1, X_3 = 5$.

- Variables:
  - $X = \{X_1, X_2, X_3\}$
- Domains:
  - $D(X_1) = D(X_2) = D(X_3) = \{1, 3, 5\}$
- Constraints:
  - $X_1 + X_2 \leq X_3$
  - $\texttt{alldifferent}([X1, X2, X3])$
- Solutions.
  - $X_1 = 1,\ X_2 = 3,\ X_3 = 5$
  - $X_1 = 3,\ X_2 = 1,\ X_3 = 5$ ■

2.1.1 Constraint Optimization Problems

Constraint Optimization Problems take the fundamental ideas of Constraint Satisfaction Problems and add an extra layer – the quest for optimization. This enhancement involves introducing an optimization criterion, which could be anything from minimizing costs to finding the fastest route or maximizing profit.

Formally, in the realm of COP, we use the notation $<X, D, C, f>$, where

- X represents the decision variables,
- D their domains,
- C the constraints, and
- f encapsulates the formalization of the optimization criterion as an objective variable.

The ultimate goal in COP is typically to minimize f or, in some cases, to maximize its negative counterpart (i.e., $-f$).

2.2 Key Elements of Constraint Programming

2.2.1 Variables and Domains

In the realm of constraint modeling, variables aren't limited to typical types like binary, integer, or continuous. They can also belong to any finite set, allowing for flexibility in representation, such as X taking values from a, b, c, d, e. Moreover, There exist special "structured" variable types like set variables(take a set of elements as value) or interval variables (for scheduling applications).

2.2.2 Constraints

Constraints serve as the glue that binds decision variables. They can be expressed in different ways, either:

- through an **extensional representation**, listing all allowed combinations explicitly,
- through a **declarative**, intentional representation, defining the relations among objects more compactly.

The Extensional representation is general but possibly inconvenient and inefficient with large domains, while Declarative representation is more compact and clear but less general.

Properties of Constraints

Understanding the properties of constraints is crucial.

Constraints are non-directional, meaning a constraint between X and Y can infer domain information on Y based on X and vice versa.

Constraints are **rarely independent**; they often share variables, acting as a communication mechanism between different constraints.

The order of imposition is also inconsequential, doesn't matter. E.g. $X + Y \leq Z, Z \geq X + Y$ represent the same constraint.

Definizione 2.2.1 — Implied Constraint. This constraint are logically implied by the constraints of the problem which cannot be deduced by the solver. Implied Constraint are **semantically redundant** because they do not change the set of solutions, but they are **computationally significant** because they can greatly reduce the search space.

Definizione 2.2.2 — Redundant Constraint. Redundant Constraints are like Implied Constraints, but they do not improve the computation time. Redundant Constraints are redundant semantically and computationally. A redundant constraint is a constraint that can be removed from a constraints model without changing the feasible region and without changing solve time.

Types of constraints

Understanding constraint types is crucial for effective modeling.

- **Algebraic expressions** like $X_1 > X_2$ or $X_1 + X_2 = X_3$ express relationships among variables using mathematical operations, defining the problem's structure.
- **Extensional constraints**, often denoted as table constraints, specify allowed combinations of variable values. For instance, (X, Y, Z) in $\{(a, a, a), (b, b, b), (c, c, c)\}$ restricts possible value assignments.
- **Element constraints**,**Variables** can serve **as subscripts**, as seen in $Y = \text{cost}[X]$, where both Y and X are variables, and 'cost' is an array of parameters. This allows for more intricate relationships between variables.
- **Logical relations**, such as $(X < Y) \vee (Y < Z) \rightarrow C$, introduce conditional constraints,

guiding the decision-making process based on logical conditions.

- **Global constraints**, like $\texttt{alldifferent}([X1, X2, X3])$, offer a more compact representation than listing individual pairwise constraints. They enhance clarity and efficiency in problem formulation.
- **Meta-constraints**, exemplified by $\sum_i (X_i > t_i) \leq 5$, introduce a higher-level perspective, involving the summation of individual constraints. This allows for the expression of complex conditions.

2.3 The Crucial Role of Modeling in Constraint Programming

In the realm of constraint programming, the art of modeling is essential, wielding profound influence on problem-solving outcomes. Let's explore why this process is so critical and how it extends beyond mere declarative specification.

The choices made regarding variables and their domains wield considerable power, directly determining the size of the search space. This size is not to be underestimated, as it escalates exponentially with each additional variable and domain. So, a seemingly innocuous decision at this stage can significantly impact the complexity of the problem at hand.

$$|D(X_1)| \times |D(X_2)| \times \cdots \times |D(X_n)| \tag{2.1}$$

Size of Search Space

Equally important is the selection of constraints, which define the problem and also play a crucial role in shaping the search space. Constraints are not mere rules; they shape the landscape on which the problem-solving journey takes place, dictating the size of the search space and the path the search process takes through it.

Modelling goes beyond a basic declarative statement of the problem. It requires engaging in the strategic decision-making process, which involves answering nuanced questions derived from human insight into the problem. These questions include considerations such as which variables are most appropriate, which constraints will be most effective, and whether there are opportunities to leverage global constraints.

2.4 Symmetry in CSP

Symmetry in Constraint Satisfaction Problems (CSPs) introduces a kind of mirroring effect in the search space. This means that if you reach a particular state that either leads to a solution or a dead end, there will be many other states with identical arrangements—symmetrically equivalent states. However, this might not be ideal when you're trying to prove if your solution is optimal, if the problem is not feasible, or if you're aiming to find all possible solutions. It can lead to a confusing and redundant situation, which is sometimes referred to as "thrashing."

Symmetry often arises from **permutations**, a concept rooted in rearranging elements within a set. For instance, consider a set of elements numbered 1 through 5. A permutation π could rearrange them, changing the order from 1 2 3 4 5 to 3 5 4 2 1.

This notion of rearrangement extends to **variable** and **value symmetry** in CSPs. **Variable symmetry** involves permuting the assignments of variables in a way that preserves the feasibility of

a solution. For example, if you have a set of variables and a permutation π that shuffles their order, applying this permutation to a feasible (or infeasible) assignment yields another feasible (or infeasible) assignment.

Similarly, **value symmetry** allows for the reordering of values within feasible assignments. A permutation π applied to the values can result in a new assignment that maintains feasibility. This concept of permuting values underscores the intuitive idea of rearranging the building blocks of a solution.

To address the challenges posed by symmetry, **symmetry-breaking constraints** come into play. These constraints deliberately disrupt the symmetrical patterns, reducing the set of solutions and making the search space more manageable. It's important to note that these constraints are not logically implied by the problem's constraints; they are intentionally introduced to break the symmetry. One common strategy involves imposing a specific ordering on variables. For instance, if a set of variables exhibits symmetry, enforcing an order like $X_1 \leq X_2 \leq \cdots \leq X_n$ helps avoid unnecessary permutations.

Besides symmetry breaking constraints enable constraint simplification and enable additional implied constraints.

However, a word of caution is necessary: even as symmetry is broken to streamline the problem-solving process, **at least one solution** from each set of symmetrically equivalent solutions must be retained. This ensures that the essence of the problem is preserved, and the solutions obtained remain meaningful and representative of the underlying constraints.

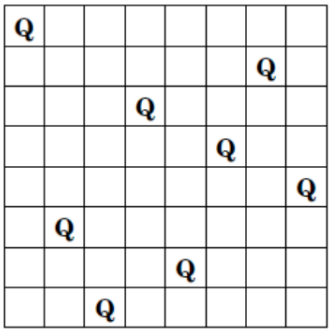
2.5 Modeling Examples

When we look at a problem, denoted as P, from different perspectives, it may lead to the creation of distinct models for the same underlying challenge. While these models share a common set of solutions, they exhibit variations in how the problem is represented.

Each model presents its unique lens on the problem, featuring different variables, domains, and constraints. This diversity in representation translates into variations in the size of the search space—an essential consideration in constraint programming. Deciding which model is superior or more suitable can be a challenging task, as each brings its own advantages and intricacies to the table.

2.5.1 N-queens problem

Let's delve into an illustrative example—the N-queens problem. The task at hand is to strategically place n queens on an $n \times n$ chessboard, ensuring that no two queens can attack each other.



Interestingly, there are various ways to define and model this problem, and we'll explore a couple of them.

Row model

Nqueens model can be represented as follow:

- Variables
  - $X = [X_1, X_2, \ldots, X_n]$
- Domains
  - $D(X_i) = [1..n]$
- Constraints
  - $\texttt{alldifferent}([X_1, X_2, \ldots, X_n])$ no column attack
  - $|X_i - X_j| \neq |i - j|,\ \forall i < j$ no diagonal attack

This model can be improved. Consider the diagonal attack constraint, it can be changed as follows:

$$
\begin{aligned}
&|X_i - X_j| \neq |i - j|,\ \forall i < j \\
&\equiv X_i - X_j \neq i - j \wedge X_i - X_j \neq j - i \wedge X_j - X_i \neq i - j \wedge X_j - X_i \neq j - i,\ \forall i < j \\
&\equiv \texttt{alldifferent}([X_1 + 1, X_2 + 2, \ldots, X_n + n]) \wedge \texttt{alldifferent}([X_1 - 1, X_2 - 2, \ldots, X_n - n])
\end{aligned}
$$

Column model

The N-queens model can be also modeled in this way:

- Variables
  - $Y = [Y_1, Y_2, \ldots, Y_n]$
- Domains
  - $D(Y_i) = [1..n]$
- Constraints
  - $\texttt{alldifferent}([X_1, X_2, \ldots, X_n])$ no column attack
  - $|X_i - X_j| \neq |i - j|,\ \forall i < j$ no diagonal attack

This model represents a different viewpoint of the row model. It use a variable Y_i. $Y_i = j$ means that the queen in column i is in row j.

Row-Column Combined model 1

We can combine Row model and Column model using a channeling constraint:

- Variables
  - $X = [X_1, X_2, \ldots, X_n]$
  - $Y = [Y_1, Y_2, \ldots, Y_n]$
- Domains
  - $D(X_i) = D(Y_i) = [1..n]$
- Constraints
  - $\texttt{alldifferent}([X_1, X_2, \ldots, X_n])$ no column attack
  - $\texttt{alldifferent}([Y_1, Y_2, \ldots, Y_n])$ no column attack
  - $|X_i - X_j| \neq |i - j|,\ \forall i < j$ no diagonal attack
  - $|Y_i - Y_j| \neq |i - j|,\ \forall i < j$ no diagonal attack
- Channeling Constraints
  - $X_i = X_j \Leftrightarrow Y_j = i$

In this model there are redundant constraints that, if removed, leave the solutions unchanged. But if we try solving the problem first with these redundant constraints and then removing them, we find

that the performance of the solver is altered. From this we can deduce that these constraints are not redundant constraints but implied constraints in that they are computationally significant.

Row-Column Combined model 2

The RC2 Model is derived by eliminating two `alldifferent` constraints from RC1 Model.

- Variables
  - $X = [X_1, X_2, \ldots, X_n]$
  - $Y = [Y_1, Y_2, \ldots, Y_n]$
- Domains
  - $D(X_i) = D(Y_i) = [1..n]$
- Constraints
  - ~~`alldifferent`$([X_1, X_2, \ldots, X_n])$~~ no column attack
  - ~~`alldifferent`$([Y_1, Y_2, \ldots, Y_n])$~~ no column attack
  - $|X_i - X_j| \neq |i - j|,\ \forall i < j$ no diagonal attack
  - $|Y_i - Y_j| \neq |i - j|,\ \forall i < j$ no diagonal attack
- Channeling Constraints
  - $X_i = X_j \Leftrightarrow Y_j = i$

This is possible thanks to the presence of the channeling constraint, which defines that the values of X_i and Y_j must differ. However, when the two global constraints are eliminated, the solver's performance deteriorates. This is because global constraints are implemented with specialized algorithms that outperform solutions lacking such global constraints. In essence, by removing these two constraints, the search space expands, leading to a decrease in solver effectiveness.

Row-Column Combined model 3

Similarly, the RC3 Model is derived by removing an implied constraint that gives additional information about diagonal attacks.

- Variables
  - $X = [X_1, X_2, \ldots, X_n]$
  - $Y = [Y_1, Y_2, \ldots, Y_n]$
- Domains
  - $D(X_i) = D(Y_i) = [1..n]$
- Constraints
  - ~~`alldifferent`$([X_1, X_2, \ldots, X_n])$~~ no column attack
  - ~~`alldifferent`$([Y_1, Y_2, \ldots, Y_n])$~~ no column attack
  - $|X_i - X_j| \neq |i - j|,\ \forall i < j$ no diagonal attack
  - ~~$|Y_i - Y_j| \neq |i - j|,\ \forall i < j$~~ no diagonal attack
- Channeling Constraints
  - $X_i = X_j \Leftrightarrow Y_j = i$

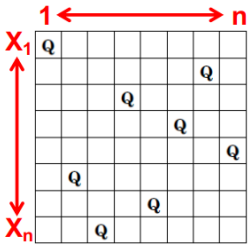
Once more, when we eliminate even this implied constraint, we notice a more evident drop in the solver's performance, resulting in an increased rate of failure

Alldiff model

Substituting the diagonal constraint of the row model with the alldifferent equivalent constraint we obtain this model that use only global constraints.

- Variables
  - $X = [X_1, X_2, \ldots, X_n]$

- Domains
  - $D(X_i) = [1..n]$
- Constraints
  - $\texttt{alldifferent}([X_1, X_2, \ldots, X_n])$ no column attack
  - $\texttt{alldifferent}([X_1 + 1, X_2 + 2, \ldots, X_n + n])$ no diagonal attack
  - $\texttt{alldifferent}([X_1 - 1, X_2 - 2, \ldots, X_n - n])$ no diagonal attack

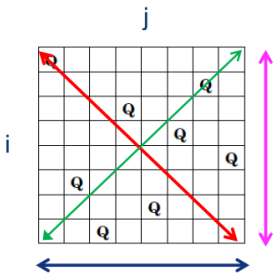


However, while this model benefits from a global constraint, it lacks straightforward symmetry-breaking mechanisms.

Boolean Symmetry Breaking Model

Another approach is the Boolean Symmetry Breaking Model. Here, binary variables B_{ij} are introduced, representing whether a queen is placed in a particular cell ($B_{ij} = 1$) or not ($B_{ij} = 0$). Constraints are then imposed to ensure only one queen per row and column and at most one queen on diagonals. The lexicographic constraint $\texttt{lex} \leq (B, \pi(B))$ breaks symmetries.

- Variables
  - B_{ij} with i and j in $[1..n]$
- Domains
  - $D(B_{ij}) = \{0, 1\}$
- Constraints
  - $\sum B_{ij} = 1$ on all row and columns
  - $\sum B_{ij} \leq 1$ on diagonals
  - $\texttt{lex} \leq (B, \pi(B))$ for all π



While in this model there is easy symmetry breaking, it lacks the global constraints present in the Alldiff Model.

Combined Model

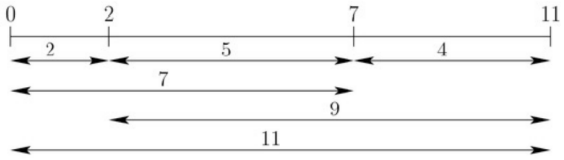
Recognizing the strengths of each model, a strategic decision emerges—why not combine them? The combined model merges both the Alldiff Model and the Boolean Symmetry Breaking Model. Through channeling constraints, consistency is maintained between the variables of the two models.

This hybrid approach offers a balanced solution, reaping the benefits of both models. It facilitates the expression of constraints, enhances constraint propagation, and provides more flexibility in choosing search variables.

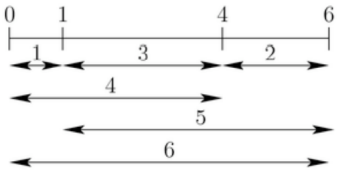
- Variables
  - $X = [X_1, X_2, \ldots, X_n]$
  - B_{ij} with i and j in $[1..n]$
- Domains
  - $D(X_i) = [1..n]$
  - $D(B_{ij}) = \{0, 1\}$
- Constraints
  - $\texttt{alldifferent}([X_1, X_2, \ldots, X_n])$ no column attack
  - $\texttt{alldifferent}([X_1 + 1, X_2 + 2, \ldots, X_n + n])$ no diagonal attack
  - $\texttt{alldifferent}([X_1 - 1, X_2 - 2, \ldots, X_n - n])$ no diagonal attack
  - $\texttt{lex} \leq (B, \pi(B))$ for all π
- Channeling constraint
  - $forall\ i, j\ X_i = j \leftrightarrow B_{ij} = 1$

2.5.2 Golomb Ruler

The Golomb Ruler problem revolves around the task of placing m marks on a ruler in such a way that the distance between each pair of marks is distinct, while keeping the overall length of the ruler to a minimum. This problem finds applications in diverse fields such as radio astronomy and information theory, adding significance to its study. Notably, solving this problem is challenging, with the largest known ruler being of order 28.



A non optimal Golomb ruler of order 4.



An optimal Golomb ruler of order 4.

Naive Model

A naive model could be described as follows

- Variables and Domains:
  - $[X_1, X_2, \ldots, X_m]$
  - X_i, representing the position of the i^{th} mark, taking a value from $\{0, 1, \ldots, 2^{m-1}\}$
- Constraints:
  - for all $i_1 < j_1, i_2 < j_2, i_1 \neq i_2$ or $j_1 \neq j_2, |X_{i1} - X_{j1}| \neq |X_{i2} - X_{j2}|$
- Objective: minimize $(\max([X_1, X_2, \ldots, X_m]))$

However, this naive model presents challenges. It introduces a significant computational burden

with $O(m^4)$ quaternary constraints, making it computationally expensive. Additionally, there is a notable loosening of the reduction in domain space, further complicating the problem.

Better Model

To enhanced the model, the introduction of auxiliary variables could be used. These variables are incorporated into the model due to the complexity or impossibility of expressing certain constraints on the primary decision variables. Additionally, some constraints on the primary decision variables might not result in significant reductions in the domain space. For this purpose, new variables, denoted as D_{ij} for all $i < j$, are introduced, representing the distance between the i^{th} and the j^{th} marks.

The constraints imposed on these auxiliary variables are twofold. Firstly, for all $i < j$, the equation $D_{ij} = |X_i - X_j|$ is enforced. Secondly, a global constraint, $\texttt{alldifferent}([D_{12}, D_{13}, \ldots, D_{(m-1)m}])$, ensures that all the distances between pairs of marks are distinct.

The model with this modification is the following:

- Variables and Domains:
  - $[X_1, X_2, \ldots, X_m]$
  - X_i, representing the position of the i^{th} mark, taking a value from $\{0, 1, \ldots, 2^{m-1}\}$
  - D_{ij} for all $i < j$:, representing the distance between the i^{th} and the j^{th} marks.
- Constraints:
  - for all $i < j,\ D_{ij} = |X_i - X_j|$
  - $\texttt{alldifferent}([D_{12}, D_{13}, \ldots, D_{(m-1)m}])$
- Objective: minimize $(\max([X_1, X_2, \ldots, X_m]))$

This refined model brings about notable improvements. The computational complexity is reduced to quadratic $O(m^2)$ ternary constraints, offering a more efficient approach to the problem. The incorporation of a **global constraint** contributes to the overall optimization of the model.

Better Model 2

Building upon the advancements made in the previous model, an additional set of constraints is introduced to further refine the model. An extra implied constraint, $\texttt{alldifferent}([X_1, X_2, \ldots, X_m])$, is incorporated.

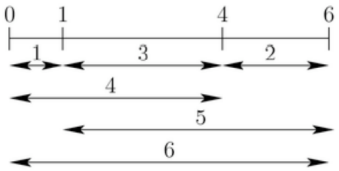
The model with this modification is the following:

- Variables and Domains:
  - $[X_1, X_2, \ldots, X_m]$
  - X_i, representing the position of the i^{th} mark, taking a value from $\{0, 1, \ldots, 2^{m-1}\}$
  - D_{ij} for all $i < j$:, representing the distance between the i^{th} and the j^{th} marks.
- Constraints:
  - for all $i < j,\ D_{ij} = |X_i - X_j|$
  - $\texttt{alldifferent}([D_{12}, D_{13}, \ldots, D_{(m-1)m}])$
  - $\texttt{alldifferent}([X_1, X_2, \ldots, X_m])$ (implied constraint)
- Objective: minimize $(\max([X_1, X_2, \ldots, X_m]))$

This augmented model boasts improvements in efficiency, now with $O(m^2)$ ternary constraints. The utilization of a global constraint continues to optimize the problem-solving approach. Furthermore, an **implied constraint** is introduced—logic deduced by the problem's constraints but not directly discernible by the solver. Though semantically redundant (with no alteration in the set of solutions), it holds computational significance by substantially curtailing the search space.

Improved Model

To further enhance the model, valuable insights can be drawn from Golomb Rulers of smaller order. Consider any k consecutive marks of a Golomb Ruler of order $n > k$; they inherently form a Golomb Ruler of order k. This observation leads to the deduction that these k consecutive marks must span a distance at least as long as the optimal size of Rulers of order k. Formally, for all $i < j$, it holds that $D_{ij} \geq$ optimal value of the ruler of order $(j - i + 1)$.



An optimal Golomb ruler of order 4.

An exploration of variable symmetries in Golomb Rulers sheds light on the impact of permuting variable assignments. Consider a scenario where $X_1 = 0, X_2 = 1, X_3 = 4, X_4 = 6$ is a feasible assignment. By permuting these variables, numerous other feasible assignments are generated, such as $X_1 = 0, X_2 = 1, X_3 = 6, X_4 = 4$ or $X_1 = 0, X_2 = 6, X_3 = 4, X_4 = 1$. In fact, there are precisely $m!$ permutations, leading to $m!$ variable symmetries for a given (un)feasible assignment.

Exploring a different facet, values can also be permuted in various ways. For instance, reversing the ruler involves the permutation $0 \to 0, 1 \to 2, 2 \to 1, 3 \to 3, 4 \to 5, 5 \to 4, 6 \to 6$. This permutation introduces an additional factor of 2, resulting in $2 * m!$ (un)feasible assignments.

Symmetry breaking constraints play a crucial role in simplifying constraints, making certain expressions redundant both semantically and computationally. For example, the constraint $X_1 < X_2 < \cdots < X_m$ makes $\texttt{alldifferent}([X_1, X_2, \ldots, X_m])$ redundant. Furthermore, symmetry breaking constraints yield additional implied constraints, exemplified by expressions like $D_{ij} < D_{ik}$ and $D_{jk} < D_{ik}$ for all $i < j < k$.

So a final model with Symmetry breaking constraint is the following:

- Variables and Domains:
  - $[X_1, X_2, \ldots, X_m]$
  - X_i, representing the position of the i^{th} mark, taking a value from $\{0, 1, \ldots, 2^{m-1}\}$
  - D_{ij} for all $i < j$:, representing the distance between the i^{th} and the j^{th} marks.
- Constraints:
  - for all $i < j,\ D_{ij} = |X_j - X_i|$ became for all $i < j,\ D_{ij} = X_j - X_i$
  - $\texttt{alldifferent}([D_{12}, D_{13}, \ldots, D_{(m-1)m}])$
  - ~~$\texttt{alldifferent}([X_1, X_2, \ldots, X_m])$~~ (becomes redundant due to symmetry breaking constraints)
- Symmetry Breaking Constraints:
  - $X_1 < X_2 < \ldots < X_m$
  - $X_1 = 0$
  - $D_{12} < D_{(m-1)m}$
- Implied constraints enabled by Symmetry Breaking Constraints:
  - $D_{ij} < D_{ik}$ and $D_{jk} < D_{ik}$ and $D_{ik} = D_{ij} + D_{jk}$ for all $i < j < k$
- New Objective: minimize (X_m)

II Constraint Propagation & Global Constraints

3. Local Consistency

3.1 Constraint solver

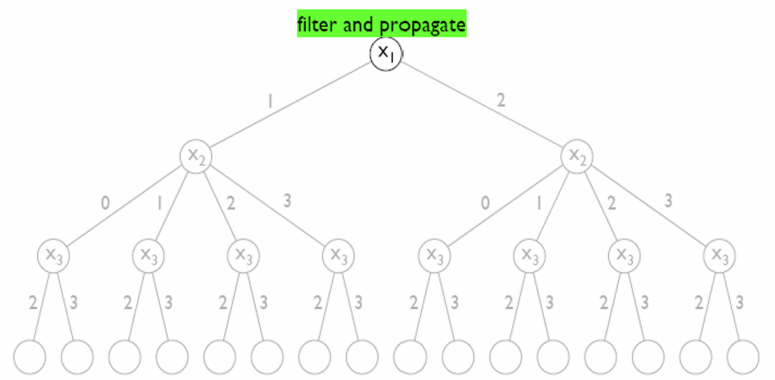
The Constraint Solver is a powerful tool used to find solutions of a CSP. It works by systematically exploring all possible variable-value combinations through a backtracking tree search. In this process, the solver makes educated guesses for the values of each variable.

During the search, the solver examines constraints to eliminate incompatible (inconsistent) values from the domains of unexplored variables. This elimination process, known as propagation, effectively shrinks the domains of future variables.

The interplay between search decisions and propagation is crucial. Search decisions and propagation occur in tandem, creating a dynamic interaction. For instance, propagation might involve setting a variable X_i to a specific value v_j, which in turn triggers further propagation and decision-making.

■ **Example 3.1 — Backtracking tree search.** In the Backtracking tree search and propagation process, let's consider a scenario with three variables: X in the domain $\{1, 2\}$, X_2 in $\{0, 1, 2, 3\}$, and X_3 in $\{2, 3\}$.

The constraints include $X_1 > X_2$, $X_1 + X_2 = X_3$, and $\texttt{alldifferent}([X_1, X_2, X_3])$.



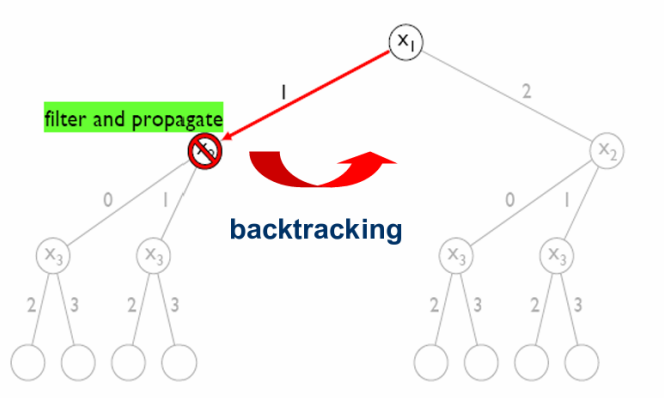
In the initial propagation, before assigning a value to any variable, the solver recognizes that X_2

cannot be 2 or 3, leading to their removal from the domain of X_2. This is depicted in the first diagram.

So now we have X in the domain $\{1, 2\}$, X_2 in $\{0, 1, \not{2}, \not{3}\}$, and X_3 in $\{2, 3\}$

After assigning the value 1 to X_1 and propagating the constraints, the solver realizes that 1 is not less than 1 (as per the first constraint), leading to the removal of 1 from the domain of X_2. The second constraint further restricts the domain of X_3. This results in a situation where no valid values exist for some variables, necessitating backtracking.

So now we have $X_1 = 1, \quad X_2 \in \{0, \not{1}\}, \quad X_3 \in \{\not{2}, \not{3}\}$

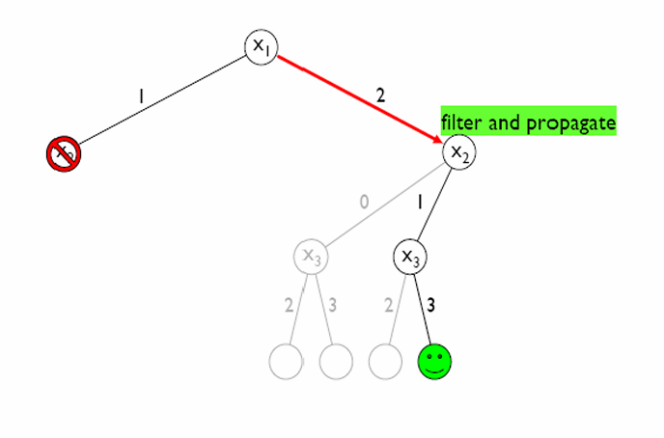


The process continues with the assignment of 2 to X_1, altering the domains of X_2 and X_3 accordingly. Propagating the constraints again, the solver reaches a state where each variable has only one possible value in its domain.

So now we have $X_1 = 2, \quad X_2 \in \{\emptyset, 1\}, \quad X_3 \in \{\not{2}, 3\}$

Consequently, the solver proceeds to assign these values to the corresponding variables, ultimately reaching a solution.

This iterative process of assigning values, propagating constraints, and backtracking continues until a solution is found or all possibilities are exhausted. The visual representations in the diagrams aid in understanding the dynamic nature of this Constraint Programming approach.



3.2 Local consistency

Let's delve into the concept of Local Consistency in Constraint Programming. It's a form of inference that helps identify inconsistent partial assignments, and it's termed "local" because it focuses on individual constraints.

One prevalent type of local consistency is domain-based, and two well-known techniques are Generalized Arc Consistency (GAC) and Bounds Consistency (BC). GAC is also known as Hyper-arc or Domain Consistency. These techniques excel at detecting inconsistent partial assignments of the form $X_i = j$. When such inconsistencies are identified, the value j can be eliminated from the domain $D(X_i)$ through a process known as propagation.

3.2.1 Generalized Arc Consistency

Let's explore Generalized Arc Consistency (GAC) further. In a constraint C defined on k variables $C(X_1, \ldots, X_k)$, it specifies the allowed combinations of values, often referred to as allowed tuples. Mathematically, C is a subset of the Cartesian product of the variable domains: $C \subseteq D(X_1) \times \cdots \times D(X_k)$.

Consider an example with three variables, where $D(X_1) = \{0, 1\}$, $D(X_2) = \{1, 2\}$, and $D(X_3) = \{2, 3\}$, with the constraint $C : X_1 + X_2 = X_3$. In this case, $C(X_1, X_2, X_3)$ consists of the allowed tuples, such as $(0, 2, 2)$, $(1, 1, 2)$, and $(1, 2, 3)$. Each allowed tuple, denoted as $(d_1, \ldots, d_k)$, where d_i belongs to the domain X_i, is termed a "**support**" for the constraint C.

A constraint $C(X_1, \ldots, X_k)$ is considered GAC if, for every variable X_i in the set $X_1, \ldots, X_k$ and for every value v in the domain $D(X_i)$, there **exists a support** for C containing v. When the constraint involves only two variables, it's referred to as Arc Consistency (AC).

■ **Example 3.2** We have two variables X_1 and X_2 with domains $D(X_1) = \{1, 2, 3\}$ and $D(X_2) = \{2, 3, 4\}$. The constraint $C : X_1 = X_2$ imposes that X_1 must be equal to X_2. Now, let's check Arc Consistency ($AC(C)$). In this case, 1 in $D(X_1)$ and 4 in $D(X_2)$ do not have a support, meaning there's no valid combination of values that satisfies the constraint. Therefore, the partial assignments $X_1 = 1$ and $X_2 = 4$ are inconsistent. ■

■ **Example 3.3** we have three variables X_1, X_2, and X_3, with respective domains $D(X_1) = \{1, 2, 3\}$, $D(X_2) = \{1, 2\}$, and $D(X_3) = \{1, 2\}$.

The constraint C : `alldifferent`$([X1, X2, X3])$ ensures that all three variables must take different values. Let's check Generalized Arc Consistency ($GAC(C)$). Here, both 1 and 2 in $D(X_1)$ do not have support, meaning there's no valid combination of values for the other variables that satisfies the constraint. Consequently, the partial assignments $X_1 = 1$ and $X_1 = 2$ are inconsistent. ■

3.2.2 Bound Consistency

Let's explore the concept of Bounds Consistency (BC). It's applicable to totally ordered domains, such as integer domains.

BC works by relaxing the domain of a variable X_i from $D(X_i)$ to the range $[\min(X_i).. \max(X_i)]$. For example, if $D(X_i) = 1, 3, 5$, BC would consider the range $[1..5]$ for X_i.

In BC, a "**bound support**" is a tuple $(d_1, \ldots, d_k)$ belonging to the constraint C, where each d_i is within the range $[\min(X_i).. \max(X_i)]$.

A constraint $C(X_1, \ldots, X_k)$ is considered BC if, for every variable X_i in the set $X_1, \ldots, X_k$, both $\min(X_i)$ and $\max(X_i)$ belong to a bound support.

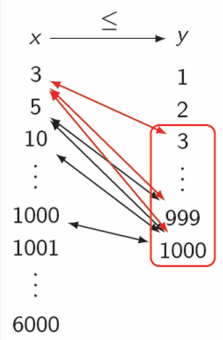
However, BC has a drawback – it might not identify all inconsistencies that would be detected by Generalized Arc Consistency (GAC). This limitation means that additional search efforts may be required to find these inconsistencies.

On the positive side, BC offers advantages. Searching for a support within a range may be more straightforward than searching within a domain. Additionally, achieving BC is often computationally more efficient than achieving GAC, which makes it particularly attractive in scenarios involving arithmetic constraints defined on integer variables with large domains. It's noteworthy that achieving BC is sufficient to ensure GAC for monotonic constraints, providing a practical and efficient approach in certain cases.

■ **Example 3.4 — GAC = BC.** Consider two decision variables, X and Y, each with their respective domains, $D(X)$ and $D(Y)$. If there exists a constraint $X \leq Y$, then the following conditions hold true for GAC:

1. All values in the domain $D(X)$ must be less than or equal to the maximum value of Y.
2. All values in the domain $D(Y)$ must be greater than or equal to the minimum value of X.

To achieve GAC in this context, it's sufficient to adjust the maximum value of X to be less than or equal to the maximum value of Y, and the minimum value of X to be less than or equal to the minimum value of Y.



Mathematically, this can be expressed as:

1. $\max(X) \leq \max(Y)$
2. $\min(X) \leq \min(Y)$

This set of conditions ensures that all valid combinations of values for X and Y are considered, satisfying the constraint $X \leq Y$ under GAC.

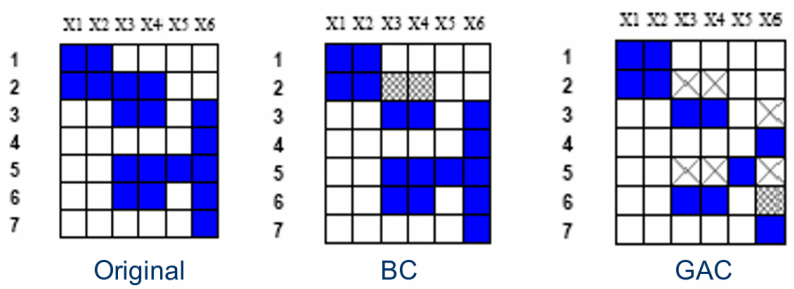
Importantly, the formula concludes that, given these conditions, using BC alone is sufficient, as it is equivalent to GAC in this specific scenario. Therefore, when dealing with constraints like $X \leq Y$, Bounds Consistency provides a practical and equivalent approach to Generalized Arc Consistency. ■

Let's explore the comparison between GAC and BC, highlighting when GAC proves to be stronger.

Consider an example scenario with several decision variables: X_1, X_2, X_3, X_4, X_5, and X_6, each with its own respective domains. The constraint in place ensures that all these variables must take different values, as expressed by $\texttt{alldifferent}([X_1, X_2, X_3, X_4, X_5, X_6])$.

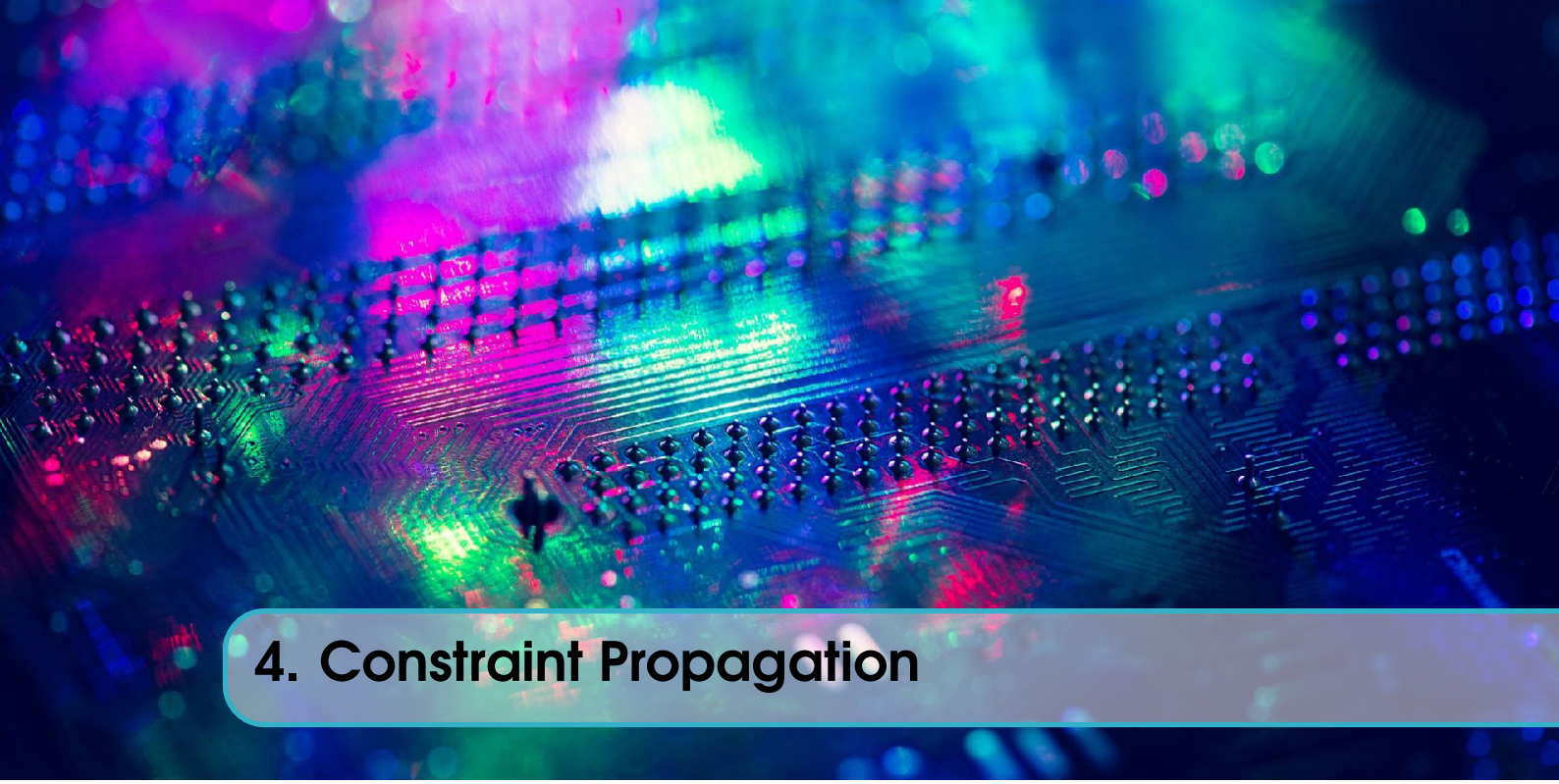
- Variables: $D(X_1) = D(X_2) = \{1, 2\}$, $D(X_3) = D(X_4) = \{2, 3, 5, 6\}$, $X_5 = 5$, $D(X_6) =$

$\{3, 4, 5, 6, 7\}$
- Constraints: `alldifferent`$([X_1, X_2, X_3, X_4, X_5, X_6])$



Now, focusing on the BC and GAC checks:

- Under BC, only the values 2 in $D(X_3)$ and 2 in $D(X_4)$ lack support, as illustrated in the provided diagram.
- However, when we consider GAC, the strength of GAC becomes evident. Not only do the values 2 in $D(X_3)$ and 2 in $D(X_4)$ lack support, but also the sets $\{2, 5\}$ in $D(X_3)$, $\{2, 5\}$ in $D(X_4)$, and $\{3, 5, 6\}$ in $D(X_6)$ have no GAC support. This emphasizes that GAC is more robust in detecting inconsistencies, capturing a broader range of unsupported combinations compared to BC.



4. Constraint Propagation

Constraint Propagation is a fundamental aspect of CP and it's often known by different names such as constraint relaxation, filtering, or local consistency enforcing.

Local consistency defines specific properties that a constraint C must satisfy to after undergoing constraint propagation. The term "constraint propagation" include various operational behaviors, and there can be multiple algorithms, each with different complexities, designed to achieve the same desired effect. The key requirement is to attain the specified property on constraint C.

4.1 Propagation Algorithm

Now, when we talk about Propagation Algorithms, their primary goal is to establish a certain level of consistency on a given constraint C. This is achieved by systematically removing inconsistent values from the domains of the variables involved in C. The level of consistency achieved depends on the nature of C. If an efficient propagation algorithm can be developed, it may achieve GAC. However, if GAC is not feasible, we may resort to BC or a lower level of consistency.

In the context of solving a CSP problem with multiple constraints, several propagation algorithms come into play. These algorithms interact with each other, and a single propagation algorithm may cause another previously propagated constraint to propagate. This process continues until a fixed point is reached where all constraints reach a certain level of consistency. The entire iterative process is called **constraint propagation**.

■ **Example 4.1** Suppose we have three decision variables, X_1, X_2, and X_3, each with the same domain $D(X_1) = D(X_2) = D(X_3) = \{1, 2, 3\}$. Now, we have three constraints:

1. $C_1 : \texttt{alldifferent}([X_1, X_2, X_3])$;
2. $C_2 : X_2 < 3$;
3. $C_3 : X_3 < 3$.

Assuming that the order of propagation is C_1, C_2, C_3, and that the propagation algorithms maintain (G)AC, let's observe the propagation process:

1. Propagation of C_1: Since C_1 is already satisfied (all values are different in the initial domain), nothing changes. C_1 remains GAC.

2. Propagation of C_2: The value 3 is removed from $D(X_2)$ to satisfy the condition. As a result, C_2 becomes Arc Consistent (AC).
3. Propagation of C_3: Similar to C_2, the value 3 is removed from $D(X_3)$ to make C_3 Arc Consistent (AC).

However, this series of propagation affects C_1. The supports of $\{1, 2\} \in D(X_1)$ in $D(X_2)$ and $D(X_3)$ are removed by the propagation of C_2 and C_3. As a result, C_1 is no longer GAC.

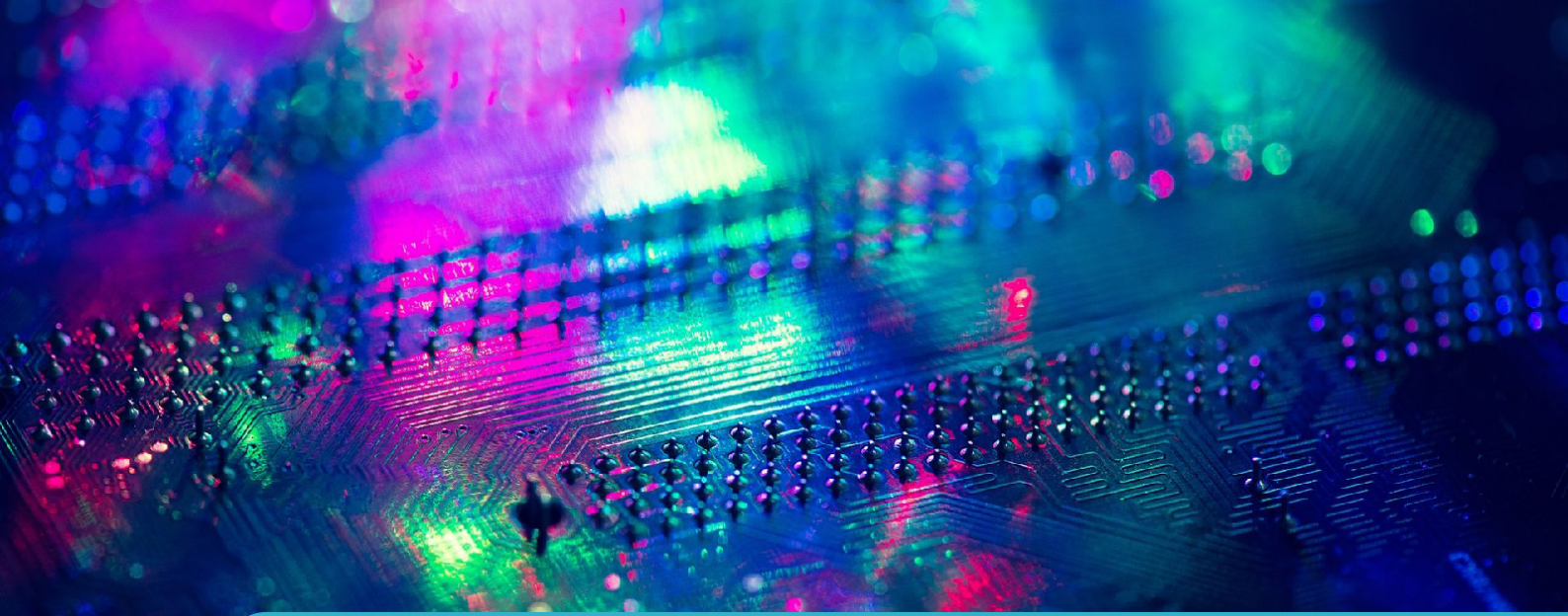
To rectify this, a re-propagation of C_1 is necessary. In this step, the values 1 and 2 are removed from $D(X_1)$ to restore the Arc Consistency (AC) of C_1.

This example illustrates how the order of constraint propagation and the impact of individual constraints can lead to changes in the consistency levels of the overall system. The iterative process of propagation and re-propagation helps in refining the constraints until a consistent solution is achieved. ■

When dealing with propagation algorithms, it's crucial to recognize that merely removing inconsistent values from domains once may not suffice. These algorithms must be **designed to "wake up" again** when necessary to ensure they achieve the desired local consistency property.

Several events can trigger constraint propagation, such as:

- changes in the domain of a variable (for GAC),
- adjustments in the domain bounds of a variable (for BC),
- when a variable is assigned a value.
- ...



5. Specialized Propagation

5.1 Specialized Algorithm

When we consider the complexity of propagation algorithms, let's assume that the size of the domain of a variable X_i is denoted as d. Following the definition of the local consistency property, a single-time Arc Consistency (AC) propagation on a constraint $C(X_1, X_2)$ takes a time complexity of $O(d^2)$.

The complexity of $O(d^2)$ is high, so we can improve the efficiency of constraint propagation by exploiting constraint semantics, with the goal of achieving faster propagation without exhaustively exploring every value in the domain.

■ **Example 5.1** Consider the constraint $C : X_1 = X_2$. In this case, we can simplify the domains by taking the intersection of $D(X_1)$ and $D(X_2)$. The complexity of this operation is determined by the cost of set intersection, providing a more efficient way to propagate the constraint without exploring each value in the domain. ■

■ **Example 5.2** Consider the constraint $C : X_1 \neq X_2$. When the domains $D(X_i)$ reduce to a single value v, we can promptly remove that value v from $D(X_j)$. This straightforward operation has a constant complexity of $O(1)$. ■

■ **Example 5.3** Consider the constraint $C : X_1 \leq X_2$. To propagate this constraint, we can compare the maximum value of X_1 with the maximum value of X_2 and ensure that the minimum value of X_1 is less than or equal to the minimum value of X_2. This comparison involves basic operations with a constant complexity of $O(1)$. ■

Specialized propagation adapts the propagation approach for specific constraints. This approach leverages the semantics of the constraint, potentially resulting in a more efficient process compared to a general propagation approach.

This approach is tailor-made for specific constraints, and it comes with its own set of advantages and disadvantages.

- One major plus point is that specialized propagation makes the most of the unique character-

istics of a constraint. By doing so, it has the potential to be significantly more efficient than a more general propagation method. This efficiency can be especially valuable when dealing with complex problems.
- However, like any technique, specialized propagation has its **limitations**. It's not a universal solution and may only be applicable to certain cases. Additionally, developing a specialized propagation technique is not always a straightforward task; it can be a bit challenging.
- Despite these challenges, specialized propagation is worth exploring, especially for constraints that **frequently appear** in the problems you're working on.

The specifics of Specialized Bounds Consistency (BC) Propagation are then discussed using an example constraint: $C : X_1 = X_2 + X_3$. In this scenario, specialized rules for propagation can be derived based on the following observations:

- $\min(X_1)$ cannot be smaller than $\min(X_2) + \min(X_3)$.
- $\max(X_1)$ cannot be larger than $\max(X_2) + \max(X_3)$.
- $\min(X_2)$ cannot be smaller than $\min(X_1) - \max(X_3)$.
- $\max(X_2)$ cannot be larger than $\max(X_1) - \min(X_3)$.
- X_3 analogous to X_2.

Based on these observations, the propagation rules for Specialized BC Propagation are as follows:

- $\max(X_1) \leq \max(X_2) + \max(X_3)$
- $\min(X_1) \geq \min(X_2) + \min(X_3)$
- $\max(X_2) \leq \max(X_1) - \min(X_3)$
- $\min(X_2) \geq \min(X_1) - \max(X_3)$

These rules are also applicable to X_3. By adapting the propagation approach to the specific characteristics of the constraint, we can efficiently enforce bounds consistency for the given equation.

■ **Example 5.4** Let's walk through an example to understand how constraint propagation works in a practical scenario.

Consider three decision variables: X_1, X_2, and X_3, with their respective domains initially set as follows:

$$D(X_1) = [4, 9], \quad D(X_2) = [3, 5], \quad D(X_3) = [2, 3].$$

Now, let's introduce a constraint: $C : X_1 = X_2 + X_3$. The goal is to propagate this constraint through the variables and refine their domains accordingly.

After the initial propagation, the domains get updated:

$$D(X_1) = [\mathbf{5}, \mathbf{8}], \quad D(X_2) = [3, 5], \quad D(X_3) = [2, 3].$$

The propagation rules are applied:

$$\max(X_1) \leq \max(X_2) + \max(X_3), \quad \min(X_1) \geq \min(X_2) + \min(X_3).$$

Subsequently, the domains are further refined:

$$D(X_1) = [\mathbf{5}, \mathbf{8}], \quad D(X_2) = [3, 5], \quad D(X_3) = [2, 3].$$

Continuing the propagation, additional rules for X_2 and X_3 are applied:

$$\max(X_2) \leq \max(X_1) - \min(X_3), \quad \min(X_2) \geq \min(X_1) - \max(X_3).$$

The process repeats for X_3 and then for X_1, resulting in further refinements to the domains.

After several iterations, the domains become:

$$D(X_1) = 5, \quad D(X_2) = [3], \quad D(X_3) = [2].$$

The final refined domains satisfy the constraint $X_1 = X_2 + X_3$. This example illustrates the step-by-step process of constraint propagation, where the domains of decision variables are adjusted based on the constraints, ultimately leading to a solution that satisfies the given conditions. ■

5.2 Global Constraints

Global Constraints serve a crucial role by capturing intricate, non-binary, and frequently occurring combinatorial patterns found in diverse applications. What makes them particularly powerful is their ability to embed specialized propagation methods that take advantage of these specific substructures.

The advantages offered by global constraints are as follows:

- From a **modeling perspective**, they offer a significant advantage. They help narrow the gap between the problem statement and the model, making it easier to represent complex relationships. Moreover, Global Constraints sometimes allow us to express constraints that would be challenging or even impossible to articulate using basic constraints, thanks to their semantic richness.
- On the **solving side** of things, Global Constraints offer two key benefits. First, they provide **strong inference** during propagation, enhancing the operational efficiency of the solving process. Second, they facilitate **efficient propagation** algorithms, making the solving process more streamlined and effective.

Some groups of Global Constraints are:

- Counting
- Sequencing
- Scheduling
- Ordering
- Balancing
- Distance
- Packing
- Graph-based
- ...

5.2.1 Counting Constraints

These constraints limit the number of variables that satisfy a condition or the number of times values are used.

Alldifferent Constraint

The `alldifferent` constraint imposes a requirement on a set of variables to have distinct values. Specifically, if you have a set of variables $X_1, X_2, \ldots, X_k$, the `alldifferent` constraint ensures that each variable in the set takes on a unique value, meaning that no two variables have the same value.

Mathematically, the `alldifferent` constraint is represented as:

$$\texttt{alldifferent}([X_1, X_2, \ldots, X_k]) \\ \text{iff } X_i \neq X_j \text{ for } i < j \in \{1, \ldots, k\}$$

This constraint functions as a permutation constraint, if the number of possible values in the domain for each variable is equal to the number of variables: $|D(X_i)| = k$

This constraint is commonly used in various problem-solving scenarios, particularly in combinatorial optimization and puzzle-solving tasks. For example: solving puzzles like Sudoku and N-queens, handling timetabling issues, such as allocating activities to different time slots. It also finds application in scheduling tasks (e.g. a team plays at most once in a week), or in configuring products (e.g. preventing the repetition of specific components).

Nvalue Constraint

The `nvalue` constraint limits the count of distinct values assigned to a set of variables.

Mathematically, the `alldifferent` constraint is represented as:

$$\texttt{nvalue}([X_1, X_2, \ldots, X_k], N) \\ \text{iff } N = |\{X_i \mid 1 \leq i \leq k\}|$$

It signifies that the number of distinct values, denoted by N, is equal to the cardinality of the set $\{X_i \mid 1 \leq i \leq k\}$. For instance, if we have $\texttt{Nvalue}([1, 2, 2, 1, 3], 3)$, the constraint enforces that there are three distinct values among the given variables.

When the count N equals the total number of variables k, the Nvalue Constraint essentially transforms into an Alldifferent Constraint, ensuring that all variables have unique values.

This constraint proves to be especially handy in scenarios such as resource allocation, where we might want to limit the number of different resource types assigned.

Global Cardinality Constraint

the Global Cardinality constraint (`gcc`) limits the frequency with which different values are taken by a set of variables.

$$\texttt{gcc}([X_1, X_2, \ldots, X_k], [v_1, \ldots, v_m], [O_1, \ldots, O_m]) \\ \text{iff } \forall j \in \{1, \ldots, m\}, O_j = |\{X_i \mid X_i = v_j, 1 \leq i \leq k\}|$$

It signifies a regulation on the occurrence of each value v_j in the variables. The constraint ensures that, for every j in the range from 1 to m, the count O_j is equivalent to the number of variables X_i taking the value v_j.

In simpler terms, if we have a scenario like $\texttt{gcc}([1, 1, 3, 2, 3], [1, 2, 3, 4], [2, 1, 2, 0])$, it implies that the value 1 occurs twice, the value 2 occurs once, and the value 3 occurs twice among the given variables.

Beside, if $O_j \leq 1$ `gcc` is equivalent to `alldifferent`.

This constraint is notably handy in practical situations like resource allocation, where we might want to limit the usage of each resource.

Among Constraint

The `Among` constraint counts and constrains the number of decision variables that take present values in the set s passed as input

$$\texttt{among}([X_1, X_2, \ldots, X_k], s, N) \\ \text{iff } N = |\{i \mid X_i \in s, 1 \leq i \leq k\}|$$

It signifies a restriction on the number of variables chosen from a set of values s. In simpler terms, it ensures that the count N is precisely the number of variables X_i that take values that belong to the set s among the given variables.

For example, consider the scenario $\texttt{among}([1, 5, 3, 2, 5, 4], \{1, 2, 3, 4\}, 4)$. Here, the constraint dictates that exactly four variables among the given set must take values from the set $1, 2, 3, 4$.

Moreover, there exists a more nuanced version of the Among Constraint, denoted as:

$$\texttt{among}([X_1, X_2, \ldots, X_k], s, l, u) \\ \text{iff } l \leq |\{i \mid X_i \in s, 1 \leq i \leq k\}| \leq u$$

This version expands the scope, allowing us to control the number of variables chosen from the set s within a specified range, denoted as l to u. In the example $\texttt{among}([1, 5, 3, 2, 5, 4], 1, 2, 3, 4, 3, 4)$, it ensures that the count falls within the range of 3 to 4 variables selected from the set $1, 2, 3, 4$.

These `among` constraints find their utility in solving sequencing problems.

5.2.2 Sequencing Constraint

These constraints ensure a sequence of variables obeys certain patterns.

Sequence/AmongSeq Constraint

The `sequence`/`amongseq` constraint constrains the number of values taken from a given set in any subsequence of q variables.

$$\texttt{sequence}(l, u, q, [X_1, X_2, \ldots, X_k], s) \\ \text{iff } \texttt{among}([X_i, X_{i+1}, \ldots, X_{i+q-1}], s, l, u) \text{ for } 1 \leq i \leq k - q + 1$$

It ensures that, in any subsequence of q variables, the count of values taken from the set s falls within the specified range of l to u. This is applied for each possible subsequence, starting from X_1 up to X_{k-q+1}

For instance, consider the $\texttt{sequence}(1, 2, 3, [1, 0, 2, 0, 3], \{0, 1\})$ constraint. This means that in any subsequence of three variables, there should be at least one and at most two occurrences of values from the set $0, 1$.

This constraint can find valuable applications in various scenarios, such as in rostering (e.g. ensuring that every employee has precisely 2 days off within any 7-day period) or in a production line (e.g. at most 1 in every 3 cars can have a sun-roof fitted).

5.2.3 Scheduling Constraint

These constraints help schedule tasks with respective release times, duration, and deadlines, using limited resources in a time interval.

Disjunctive Resource Constraint

The "Disjunctive Resource Constraint" ensures tasks do not overlap in time. This constraint is also commonly known as the `noOverlap` constraint.

Given tasks $t_1, \ldots, t_k$, each associated with a start time S_i and duration D_i:

$$\texttt{disjunctive}([S_1, \ldots, S_k], [D_1, \ldots, D_k])$$
$$\text{iff } \forall i < j (S_i + D_i \leq S_j) \vee (S_j + D_j \leq S_i)$$

It ensures that for any two tasks t_i and t_j (where $i < j$), either the start time of t_i plus its duration D_i is less than or equal to the start time of t_j OR the start time of t_j plus its duration D_j is less than or equal to the start time of t_i.

In simpler terms, this constraint guarantees that tasks do not overlap or share the same time slot. It's particularly useful in scenarios where a resource, such as a machine or a person, can handle only one task at any given time.

To illustrate, consider the visual representation provided in the image:

Cumulative Resource Constraint

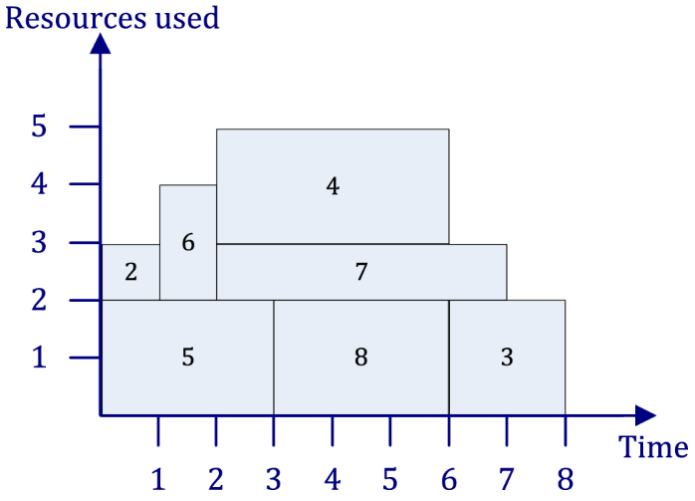
The "Cumulative Resource Constraint," regulates the usage of a shared resource.

Given tasks $t_i, \ldots, t_k$, each associated with a start time S_i, duration D_i, resource requirement R_i, and a resource with a capacity C:

$$\texttt{cumulative}([S_1, \ldots, S_k], [D_l, \ldots, D_j, [R_i, \ldots, R_k], C)$$
$$\text{iff} \sum_{i|S_i \leq u < S_i + D_i} R_i \leq C, \; \forall u \in D$$

This constraint ensures that for any time unit u within the domain D, the sum of the resource requirements of all tasks t_i running at that time is less than or equal to the resource capacity C. In simpler terms, it guarantees that the total resource consumption at any given time does not exceed the capacity of the shared resource.

To illustrate, consider the visual representation provided in the image:



This visual demonstrates how the Cumulative Resource Constraint ensures that the cumulative resource usage across multiple tasks aligns with the resource capacity, preventing any resource overload.

5.2.4 Ordering Constraint

These constraint enforce an ordering between the variables or the values.

Lexicographic Ordering Constraint

The "Lexicographic Ordering Constraint" is designed to ensure that a sequence of variables is lexicographically less than or equal to another sequence of variables. In other words, we can express $\texttt{lex} \leq ([X_1, X_2, \ldots, X_k], [Y_1, Y_2, \ldots, Y_k])$ as follows:

$$
\begin{aligned}
&X_1 \leq Y_1 \wedge \\
&(X_1 = Y_1 \Rightarrow X_2 \leq Y_2) \wedge \\
&(X_1 = Y_1 \wedge X_2 = Y_2 \Rightarrow X_3 \leq Y_3) \wedge \ldots \\
&(X_1 = Y_1 \wedge X_2 = Y_2 \ldots X_{k-1} = Y_{k-1} \Rightarrow X_k \leq Y_k)
\end{aligned}
$$

This constraint guarantees that at each position in the sequence, the corresponding variables satisfy the lexicographical order. To illustrate, consider the example $\texttt{lex} \leq ([1, 2, 4], [1, 3, 3])$. This evaluates to true, indicating that the sequence [1, 2, 4] is lexicographically less than or equal to the sequence [1, 3, 3].

The primary utility of the Lexicographic Ordering Constraint lies in **symmetry breaking**. It helps avoid permutations of (groups of) variables, allowing for a more structured and controlled representation of solutions.

■ **Example 5.5 — Permutation of variables.** For example we can use the `lex` constraint on permutation of a set of variables

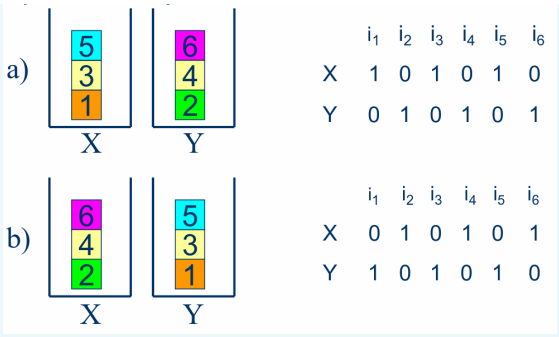
$$\texttt{lex} \leq ([X_1, X_2, \ldots, X_k], \pi([X_1, X_2, \ldots, X_k])) \text{ for some } \pi.$$

E.g. with n-Queens:

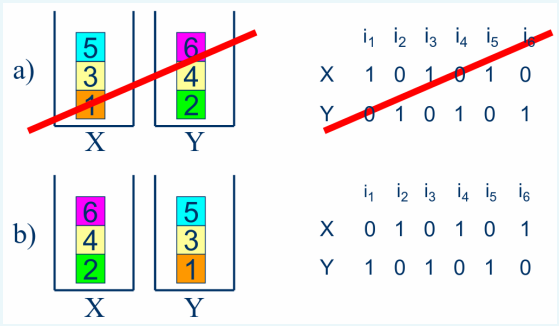
```
constraint
    lex_lesseq(array1d(qb), [ qb[j,i] | i,j in 1..n ])
/\  lex_lesseq(array1d(qb), [ qb[i,j] | i in reverse(1..n), j in 1..n ])
/\  lex_lesseq(array1d(qb), [ qb[j,i] | i in 1..n, j in reverse(1..n) ])
/\  lex_lesseq(array1d(qb), [ qb[i,j] | i in 1..n, j in reverse(1..n) ])
/\  lex_lesseq(array1d(qb), [ qb[j,i] | i in reverse(1..n), j in 1..n ])
/\  lex_lesseq(array1d(qb), [ qb[i,j] | i,j in reverse(1..n) ])
/\  lex_lesseq(array1d(qb), [ qb[j,i] | i,j in reverse(1..n) ])
;
```

■

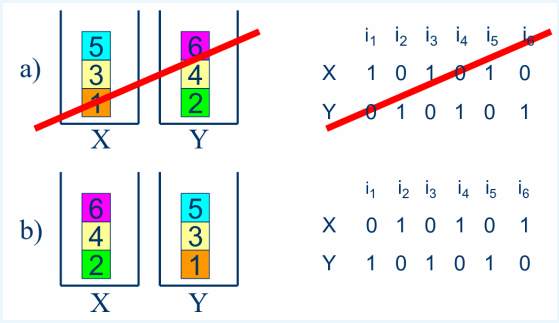
■ **Example 5.6 — Permutation of Two Sequences of Variables.** In this example, we have that assignments of items to two identical bins can be represented by a matrix of Boolean variables:



In this case we have row symmetry between the bin of figure a and the bin of figure b. To avoid this we can use the constraint $\texttt{lex} \leq (X, Y)$. With this constrain we have:



In addition, cubes of the same color are equivalent and therefore interchangeable. So we can obtain symmetries in columns such as the following:



To avoid this permutation the constraint $\texttt{lex} \leq (i_3, i_4)$ is used. ■

Value Precedence Constraint

The "Value Precedence Constraint is designed to ensure that one value precedes another in a sequence of variables.

We express `value_precede` as follows:

$$\texttt{value\_precede}(v_{j1}, v_{j2}, [X_1, X_2, \ldots, X_k]) \\ \text{iff } \min\{i \mid X_i = v_{j1} \vee i = k+1\} < \min\{i \mid X_i = v_{j2} \vee i = k+2\}$$

This constraint ensures that the earliest occurrence of v_{j1} in the sequence is before the earliest

occurrence of v_{j2}.

For instance, consider the example $\texttt{value\_precede}(5, 4, [2, 5, 3, 5, 4])$. In this case, it holds true, indicating that the value 5 precedes the value 4 in the sequence $[2, 5, 3, 5, 4]$.

The primary utility of the Value Precedence Constraint lies in symmetry breaking. By enforcing specific orderings of values in a sequence, this constraint helps avoid unnecessary permutations, contributing to a more organized and controlled solution space.

5.3 Specialized Propagation for Global Constraints

This section explores the techniques for creating specialized propagation mechanisms tailored to global constraints. Two key methods are employed in this process: constraint decomposition and the development of dedicated ad-hoc algorithms.

- **Constraint Decomposition**: In the constraint decomposition approach, a global constraint is broken down into smaller, more manageable sub-constraints. Each of these sub-constraints is then assigned a known propagation algorithm. The propagation of these smaller constraints collectively contributes to achieving the desired effect of the original global constraint.
- **Dedicated Ad-Hoc Algorithm**: Alternatively, we can opt for a dedicated ad-hoc algorithm designed specifically for a particular global constraint. This approach involves crafting a specialized algorithm tailored to exploit the semantics and characteristics of the given global constraint. While this method might be more intricate and requires careful development, it can lead to highly efficient and optimized propagation mechanisms.

5.3.1 Constraint Decomposition

This approach involves breaking down a global constraint into smaller, more manageable constraints. Each of these smaller constraints comes equipped with a known propagation algorithm, making them easier to handle.

By propagating each of these simplified constraints individually, we effectively achieve propagation for the original global constraint. This technique proves to be highly effective and efficient, especially for certain types of global constraints. It simplifies the overall constraint-solving process by dealing with smaller, more specialized components, each contributing to the resolution of the larger constraint puzzle.

The process of constraint decomposition, while useful in simplifying the handling of complex constraints, comes with its own set of considerations. One important point to note is that constraint decompositions may **not always result in effective propagation**.

In some cases, achieving Generalized Arc-Consistency (GAC) on the original constraint proves to be more powerful than achieving (G)AC on the individual constraints resulting from the decomposition.

A Decomposition of Among

We will now examine the decomposition of the `among` constraint into more manageable components. The original `among` constraint is defined as follows:

$$\texttt{among}([X_1, X_2, \ldots, X_k], s, N)$$

To break this down, we introduce several auxiliary variables and logical constraints:

1. B_i for $1 \leq i \leq k$, where $D(B_i) = \{0, 1\}$.
2. C_i: $B_i = 1$ if and only if $X_i \in s$ for $1 \leq i \leq k$.

3. C_{k+1}: The sum constraint $\sum_i B_i = N$.

In simpler terms, each B_i serves as a binary indicator, representing whether X_i is in the set s or not. The C_i constraints ensure that the binary indicators accurately reflect the membership status of X_i in s. The final C_{k+1} constraint ensures that the total count of B_i variables set to 1 is exactly N.

By enforcing AC on each C_i and BC on the sum constraint $\sum_i B_i = N$, we guarantee that the original `among` constraint achieves Generalized Arc-Consistency (GAC).

A Decomposition of Alldifferent

We will now examine the breakdown of the `alldifferent` constraint, which aims to ensure that a set of variables takes distinct values. The original `alldifferent` constraint is defined as follows:

$$\texttt{alldifferent}([X_1, X_2, \ldots, X_k])$$

To simplify the handling of this constraint, we employ a decomposition strategy, expressing it as a conjunction of difference constraints. Specifically, we introduce constraints C_{ij} for each pair $i < j$ in the index set $\{1, \ldots, k\}$:

$$C_{ij} : X_i \neq X_j$$

In simpler terms, each C_{ij} constraint asserts that the values of X_i and X_j must be distinct.

It's crucial to note that achieving Arc-Consistency (AC) on all individual C_{ij} constraints is not equivalent to achieving GAC on the original `alldifferent` constraint.

An example illustrates this point: consider $\texttt{alldifferent}([X_1, X_2, X_3])$ with $D(X_1) = D(X_2) = D(X_3) = \{1, 2\}$. In this scenario, the original `alldifferent` constraint is not GAC, meaning it fails to ensure global consistency. However, the decomposition, despite achieving AC on each C_{ij}, doesn't provide additional pruning in this specific case.

A Decomposition of Sequence

We will now examine the breakdown of the `sequence` constraint, designed to impose constraints on the number of values taken in any subsequence of a sequence of variables. The original `sequence` constraint is expressed as:

$$\texttt{sequence}(l, u, q, [X_1, X_2, \ldots, X_k], s)$$

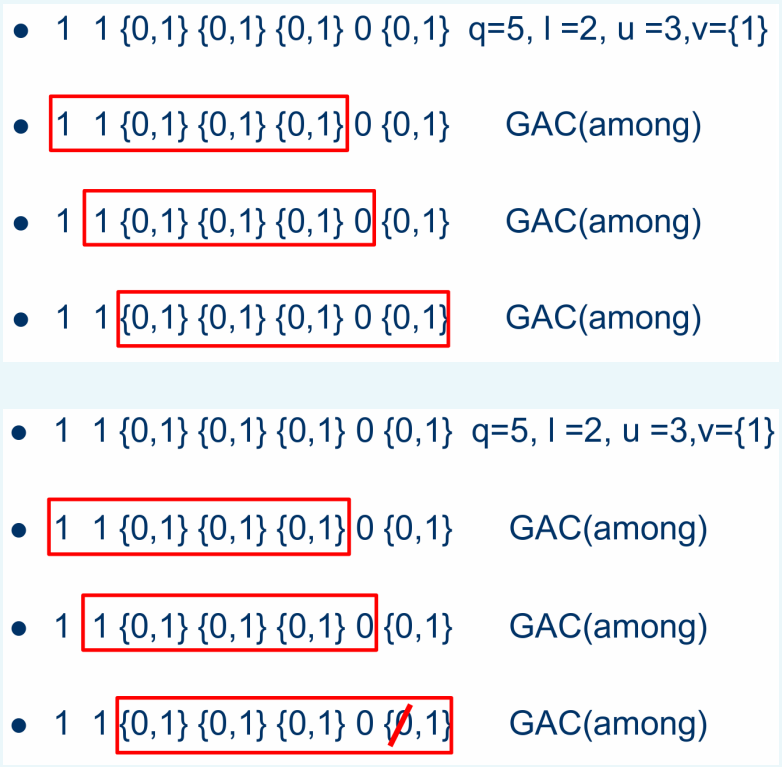
To simplify the handling of this complex constraint, we adopt a decomposition strategy, representing it as a conjunction of `among` constraints. Specifically, we introduce constraints C_i for each index i ranging from 1 to $k - q + 1$:

$$C_i : \texttt{among}([X_i, X_{i+1}, \ldots, X_{i+q-1}], s, l, u)$$

In simple terms, each C_i constraint asserts that the subsequence of variables $[X_i, X_{i+1}, \ldots, X_{i+q-1}]$ should have between l and u occurrences of values from set s.

It's noteworthy that achieving Generalized Arc-Consistency (GAC) on each individual C_i constraint does not guarantee the same level of global consistency as the original `sequence` constraint. An example illustrates this point: consider $\texttt{sequence}(2, 3, 5, [X_1, X_2, \ldots, X_7], \{1\})$ with $X_1 = X_2 = 1$, $X_6 = 0$ and $D(X_i) = \{0, 1\}$ for $i \in \{3, 4, 5, 7\}$. In this scenario, the original `sequence` constraint is not GAC, meaning it fails to ensure global consistency. However, even decomposition does not get GAC on every C_i and does not provide additional pruning.

■ **Example 5.7 — GAC on decomposition of `sequence`.** This example show that decomposition of `sequence` is not GAC.



■

A Decomposition of Lex

We will now examine the decomposition of the `lex` constraint, denoted as

$$\texttt{lex} \leq ([X_1, X_2, \ldots, X_k], [Y_1, Y_2, \ldots, Y_k])$$

Decomposition using Disjunctions

The first approach involves expressing the `lex` constraint as a conjunction of disjunctions. We introduce a binary variable B with domain $D(B) = \{0, 1\}$ for each position i from 1 to $k + 1$. This variable serves as an indicator that the vectors have been appropriately ordered by position $i - 1$. Additionally, B_1 is set to 0 to establish the initial condition. We formulate constraints C_i for each i within the range of 1 to k:

$$C_i : (B_i = B_{i+1} = 0 \wedge X_i = Y_i) \vee (B_i = 0 \wedge B_{i+1} = 1 \wedge X_i < Y_i) \vee (B_i = B_{i+1} = 1)$$

Ensuring Generalized Arc-Consistency (GAC) on all C_i constraints guarantees GAC on the original $\texttt{lex} \leq$ constraint.

Decomposition using Implications

An alternative approach involves decomposing the `lex` constraint as a conjunction of implications:

$$X_1 \leq Y_1 \wedge (X_1 = Y_1 \Rightarrow X_2 \leq Y_2) \wedge \ldots \wedge (X_1 = Y_1 \wedge X_2 = Y_2 \wedge \ldots \wedge X_{k-1} = Y_{k-1} \Rightarrow X_k \leq Y_k)$$

However, achieving Arc-Consistency (AC) on this decomposition does not guarantee the same level of consistency as GAC on `lex` $\leq$. An illustrative example involves `lex` $\leq ([X_1, X_2], [Y_1, Y_2])$

with $D(X_1) = \{0, 1\}$, $X_2 = 1$, $D(Y_1) = \{0, 1\}$ and $Y_2 = 0$. Despite $\texttt{lex} \leq$ not achieving GAC, the decomposition doesn't result in additional pruning.

In the intricate landscape of constraint programming, these decomposition strategies provide a nuanced understanding of how to break down complex constraints for effective handling and analysis.

5.3.2 Decomposition vs Ad-hoc Algorithm

The choice between Decomposition and Ad-hoc Algorithms involves a delicate balance of effectiveness and efficiency. While a decomposition may effectively break down a complex constraint into simpler components, it may not always translate to efficient propagation.

Consider this: Propagating a constraint through an ad-hoc algorithm often proves to be swifter than propagating the numerous constraints resulting from decomposition. This efficiency is attributed to incremental computation.

Incremental Computation

In constraint propagation, algorithms are frequently invoked multiple times. The inefficiency arises when recomputing everything from scratch during each call. Incremental computation comes to the rescue by introducing a more nuanced approach.

During the initial invocation of a propagation algorithm, specific partial results are strategically cached. This cache becomes a valuable resource during subsequent invocations, allowing us to leverage the stored data. This sophisticated approach to computation not only enhances efficiency but also streamlines the overall process.

However, to fully harness the power of incremental computation, one must delve into the granular details of propagation. This includes understanding which variables have undergone pruning and which values have been trimmed from the domain.

■ **Example 5.8** Now we examine a Dedicated BC Algorithm for Sum.

Dedicated BC Algorithm for Sum

Consider the constraint $\sum_i X_i = N$, where X_i and N are integer variables. This dedicated algorithm ensures the integrity of the summation while adhering to specific conditions:

- $\min(N) \geq \sum_i \min(X_i)$: The minimum value of N should be greater than or equal to the sum of the minimum values of X_i.
- $\max(N) \leq \sum_i \max(X_i)$: Similarly, the maximum value of N should not exceed the sum of the maximum values of X_i.
- $\min(X_i) \geq \min(N) - \sum_{j \neq i} \max(X_j)$ for $1 \leq i \leq n$: The minimum value of each X_i should be greater than or equal to the difference between the minimum value of N and the sum of the maximum values of all other X_j where $j \neq i$.
- $\max(X_i) \leq \max(N) - \sum_{j \neq i} \min(X_j)$ for $1 \leq i \leq n$: Analogously, the maximum value of each X_i should not exceed the difference between the maximum value of N and the sum of the minimum values of all other X_j where $j \neq i$.

BC Decomposition for Sum

Delving into the decomposition strategy for $\sum_i X_i = N$, we unravel the intricate structure by breaking it down into simpler constraints. This decomposition unfolds as a series of equations:

- $X_1 + X_2 = Y_1$
- $Y_1 + X_3 = Y_2$
- ...
- $Y_{n-1} + X_n = N$

This sequential arrangement of equations ensures that the sum of the variables gradually reaches the desired total, N.

Filtering min(N)

To optimize the algorithm, filtering conditions are introduced to minimize access and enhance efficiency:

- $\min(X_1) + \min(X_2) \leq \min(Y_1)$
- $\min(Y_1) + \min(X_3) \leq \min(Y_2)$
- ...
- $\min(Y_{n-1}) + \min(X_n) \leq \min(N)$

This method with decomposition is semantically equivalent to the constraint $\sum_i \min(X_i) \leq \min(N)$, but there are difference in terms of efficiency

	Filtering min(N)	$\sum_i \min(X_i) \leq \min(N)$
Read access	$2 * (n-1)$	n
Write access	$n-1$	1
Sum	$n-1$	$n-1$

Incremental Computation

Now we focus on managing the upper bounds, seeking to reduce the computational complexity from $O(n)$ to a remarkably efficient $O(1)$.

For the constraint $\max(N) \leq \sum_i \max(X_i)$, we strategically introduce a caching mechanism, encapsulated as $\texttt{max\$}(N)$. This cache stores the maximum value of N, preserving it for future reference. The pivotal innovation lies in dynamically updating this cache whenever the bounds of a variable X_i undergo pruning.

Upon pruning the bounds of X_i, the constraint adapts as follows:

$$\max(N) \leq \max \$(N) - \text{old}(\$ \max(X_i)\$) - \$ \max(X_i)$$

This dynamic adjustment ensures that the constraint stays synchronized with the evolving bounds of the variables. It is an elegant dance of computational finesse.

Comparing the classical summation with its incremental counterpart unveils a substantial reduction:

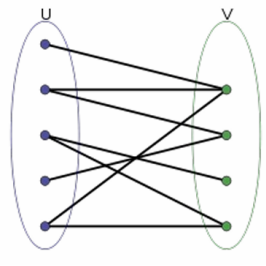
	Classical Sum	Incremental Sum
Read access	n	3
Write access	1	1
Sum	$n-1$	2

5.4 Dedicated Propagation Algorithms

Ad-hoc methods exhibit both effectiveness and efficiency in constraint handling. These algorithms not only maintain Generalized Arc-Consistency (GAC) in polynomial time but also excel in the detection of inconsistent values when compared to decomposition techniques.

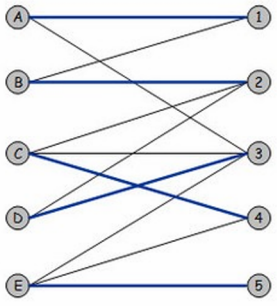
One notable GAC propagation algorithm focuses on the constraint `alldifferent`$([X_1, X_2, \ldots, X_k])$. This algorithm, introduced by Jean-Charles Régin in the proceedings of AAAI'1994, operates within polynomial time constraints while ensuring GAC is consistently maintained. The crux of its functionality lies in establishing a profound relationship between the solutions of the constraint and the inherent properties of a graph, specifically, a **maximal matching** in a **bipartite graph**. Remarkably, a similar algorithmic achievement can be obtained through the application of flow theory.

Definizione 5.4.1 — Bipartite graph. 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.



Definizione 5.4.2 — Matching. A matching in a graph is a subset of its edges such that no two edges have a node in common. **Maximal matching** is the largest possible matching.

In a bipartite graph, maximal matching covers one set of nodes.



Observation 5.4.1 Consider a **bipartite graph** denoted as G, constructed between the variables $[X_1, X_2, \ldots, X_k]$ with their feasible values, forming what is known as the **variable-value graph**.

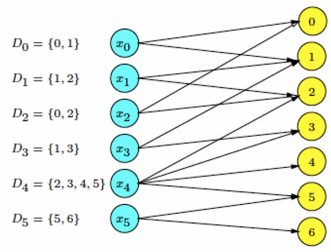


Figure 5.1: Variable-Value Graph

An **assignment** of values to these variables is a solution if and only if it corresponds to a **maximal matching** in graph G.

Here is the main point.: a maximal matching, a configuration where no more edges can be added without violating the matching condition, covers all the variables. Thus, by **computing all maximal matchings, we can find all the consistent partial assignments**.

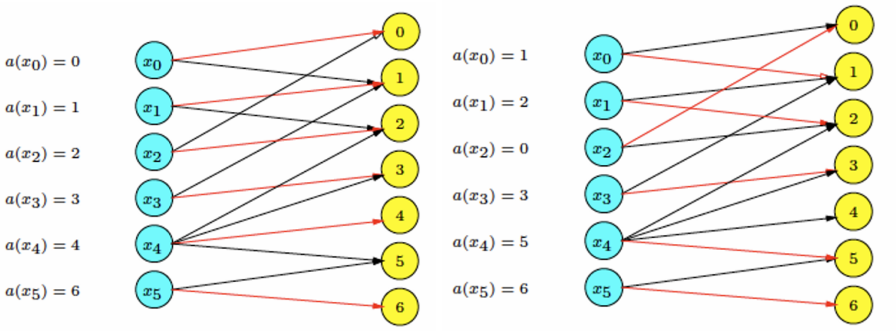


Figure 5.2: Two Maximal Matching

Notation 5.4.1 — Matching edge. An edge is called matching edge if takes part in a matching; **free** otherwise.

Notation 5.4.2 — Matched node. A node is called matched node if incident to a matching edge; **free** otherwise.

Notation 5.4.3 — Vital edge. An edge is called vital edge if belongs to every maximal matching

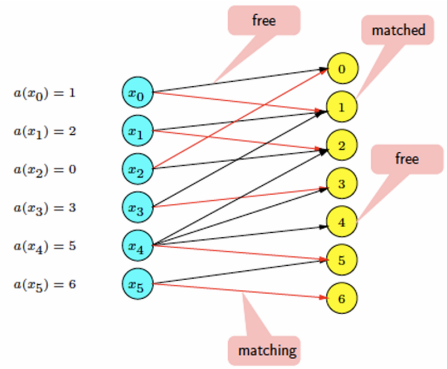


Figure 5.3: Notation

5.4.1 Algorithm

The algorithm to maintain GAC in polinomial time is the following:

- Compute all maximal matchings.
- If no maximal matching exists: failure.
- If An **edge free** in all maximal matchings:
  - Remove the edge.
  - Amounts to **removing** the corresponding **value** from the domain of the corresponding **variable**.
- A vital edge:
  - **Keep** the edge.

  - Amounts to **assigning** the corresponding **value** to the corresponding **variable**.
- Edges matching in some but not all maximal matchings:
  - Keep the edge.

The computation of all maximal matchings is inefficient to compute naïvely. However, employing matching theory unveils a more efficient approach. A single maximal matching has the power to represent the entirety of all maximal matchings!

Notation 5.4.4 — Alternating path. A simple path with edges alternating between being free and matching.

Notation 5.4.5 — Alternating cycle. A cycle distinguished by the alternation of edges between free and matching.

Notation 5.4.6 — Length of path/cycle. The length of a path or cycle is determined by the number of edges it include.

Notation 5.4.7 — Even path/cycle. A path or cycle with an even length

A pivotal concept within Matching Theory is that of a **maximal matching**.

An edge, denoted as e, is deemed part of a maximal matching under the following conditions:

1. e belongs to the maximal matching, denoted as M.
2. e forms part of an **even alternating path** originating from a free node.
3. e is an integral part of an **even alternating cycle**.

To compute alternating path/cycles, we will orient edges of an arbitrary maximal matching:

- Matching edges are directed from variables to values.
- Free edges are directed from values to variables.

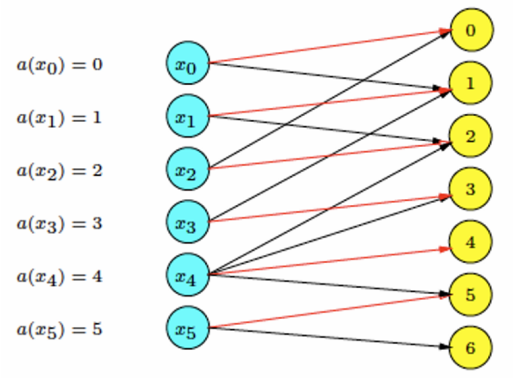


Figure 5.4: Before orientation

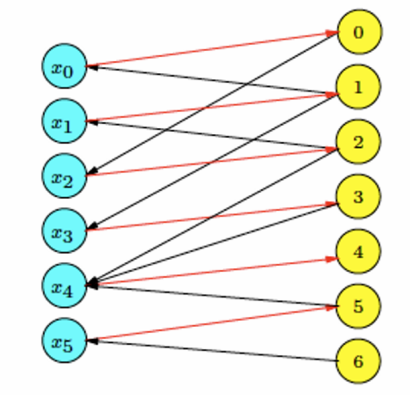


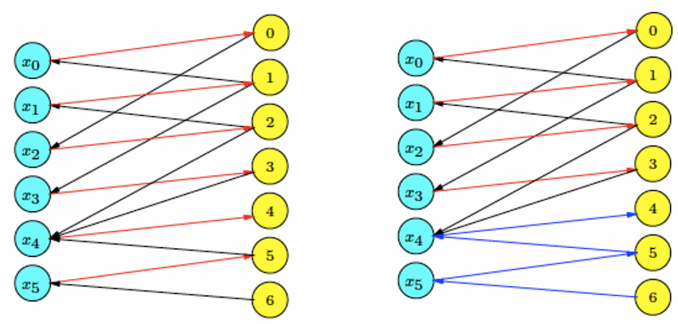
Figure 5.5: After orientation

Now we will use the concept of "Even Alternating Paths" and "Even Alternating Cycles".

Even Alternating Paths

- Start from a free node and search for all nodes on directed simple path.
  - Mark all edges on path.
  - Alternation built-in.
- Start from a value node.
  - Variable nodes are all matched.
- Finish at a value node for even length.

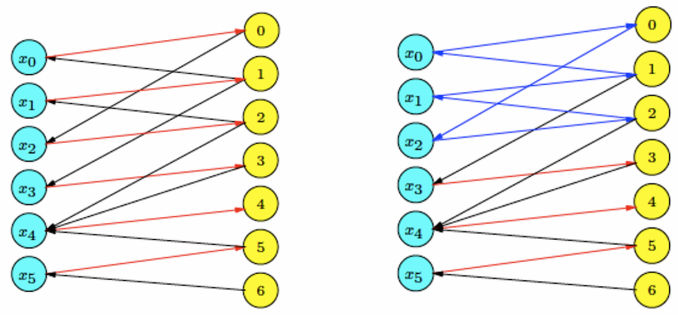
The intuition is that the edges crossed along the way can be permuted.



Even Alternating Cycles

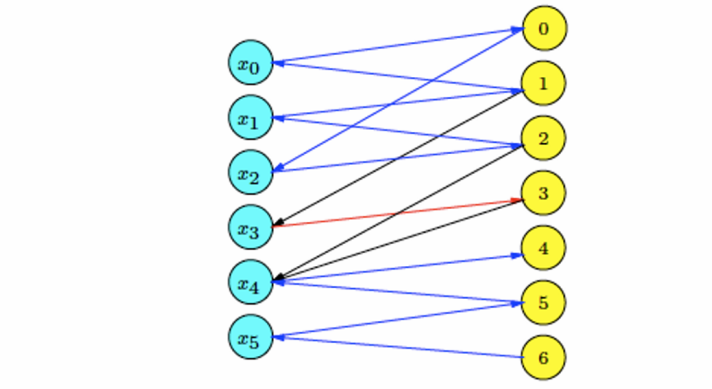
- Compute strongly connected components (SCCs).
  - Two nodes a and b are strongly connected iff there is a path from a to b and a path from b to a.
  - **Strongly connected component**: any two nodes are strongly connected.
  - Alternation and even length built-in.
- Mark all edges in all strongly connected components.

The intuition is that variables, represented by nodes in the graph, consume all possible values.



Removing edges

Now that all the edges involved have been marked we find ourselves in the following situation



We can proceed with the final part of the algorithm, edge removal.

- Remove the edges which are:

  - free, indicating that they do not occur in the arbitrary maximal matching, and are simultaneously not marked, implying their absence in any maximal matching. This includes edges marked in black within the provided example.
- Keep the edge **matched** and **not marked**.
  - edges falling into the category of being "matched" and remaining unmarked are to be retained. These edges, highlighted in red in the example, hold a crucial status, denoted as **vital edges**.

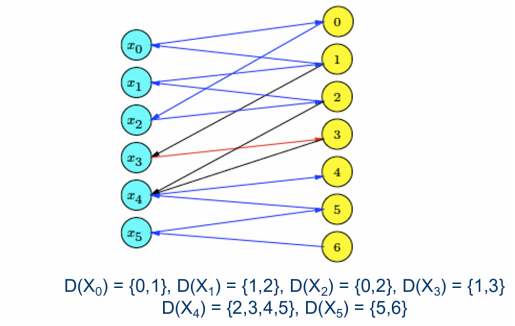


Figure 5.6: Before removing edges

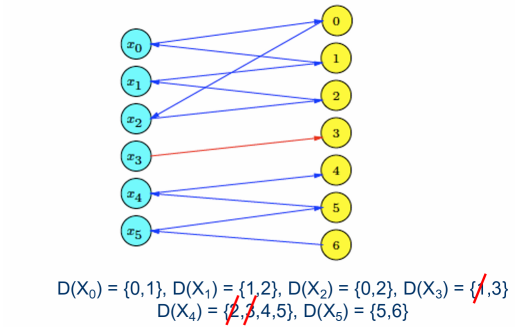


Figure 5.7: After removing edges

5.4.2 Algorithm using matching theory

The algorithm for using matching theory is as follows:

- Construct the variable-value graph.
- Find a maximal matching M; otherwise fail.
- Orient graph (done while computing M).
- Mark edges starting from free value nodes using graph search.
- Compute SCCs and mark joining edges.
- Remove not marked and free edges.

Beside it's possible to use incremental properties to increase the efficiency of the algorithm. The Incremental Properties associated with the algorithm are defined to maintain the variable and value graph across different invocations, ensuring a seamless transition between executions. When re-executed, the following steps are undertaken:

1. Clear marks on edges.
2. Remove edges not present in the domains of the respective variables.
3. If a matching edge is removed, compute a new maximal matching; otherwise, repeat the marking and removal process.

The runtime complexity of the algorithm is analyzed in terms of specific scenarios:

1. First Call:
   - Consistency check in $O(\sqrt{k}m)$ time.
     - Matching: $O(\sqrt{k}m)$
     - Alternating path: $O(m)$
     - Strongly Connected Components (SCCs): $O(k+m)$
   - Establishing Generalized Arc-Consistency (GAC) in $O(m)$ time.
2. After q Variable Domains have modified:
   - Matching in $O(\min\{qm, \sqrt{k}m\})$ time.
   - Establishing GAC in $O(m)$ time.

5.4.3 Conclusions

Developing dedicated algorithms for specific constraints may not always be straightforward, and the ease of creating such algorithms is often influenced by the underlying semantics of the constraint. Here are some considerations:

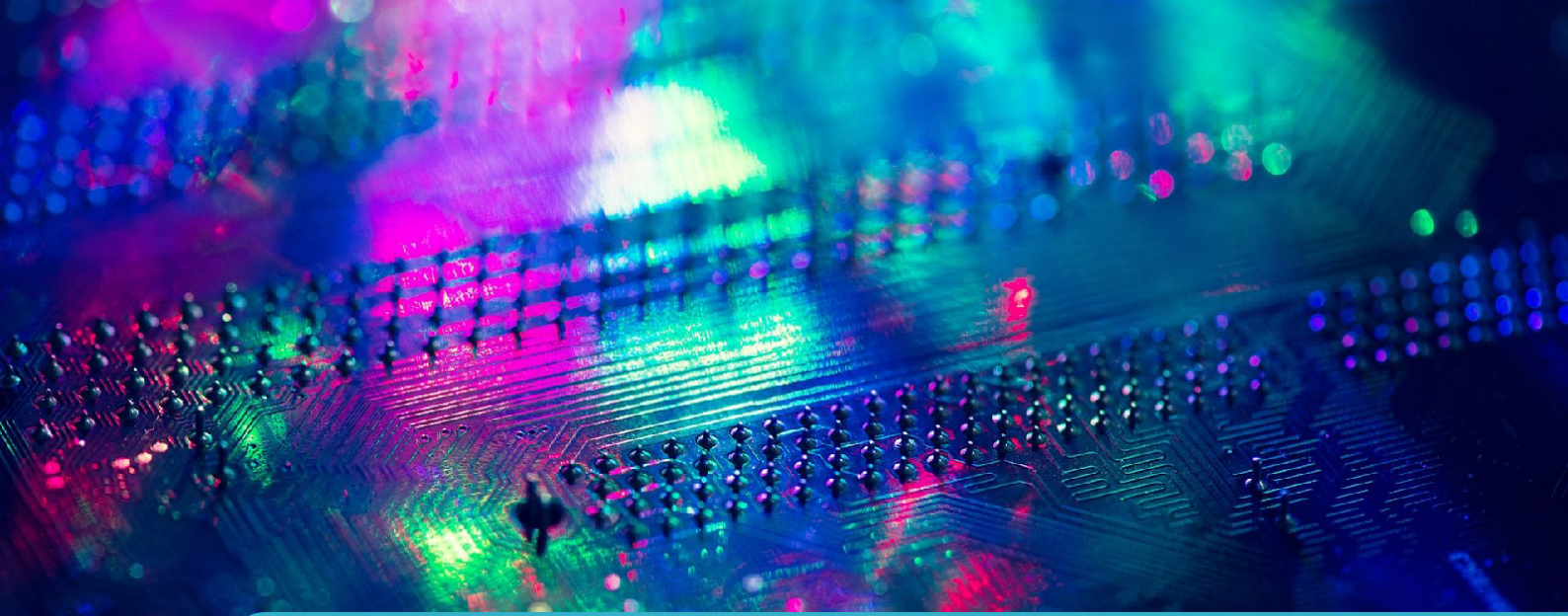
A precise understanding of constraint semantics can provide valuable hints for the design of dedicated algorithms. Semantics based on graph theory, flow theory, combinatorics, automata theory, dynamic programming, or complexity theory often guide the algorithmic design process.

GAC for certain constraints may be NP-hard, making the development of dedicated algorithms challenging. Examples include constraints like `nvalue`, `sequence` + `gcc`, and `gcc` using variables for occurrences.

- In situations where achieving GAC is computationally demanding, algorithms maintaining weaker consistencies become of interest. This includes Bound Consistency and variations that strike a balance between GAC and BC, or use GAC on some variables and BC on others.

Some constraints may resist easy decomposition into simpler sub-problems, making it challenging to build an efficient dedicated algorithm. The ability to decompose a constraint is often crucial for developing specialized algorithms.

Even if decomposition is possible, building an algorithm that is both efficient and effective can be a non-trivial task. Balancing computational complexity and practical utility is a key consideration.



6. Global Constraints for Generic Purposes

The utilization of global constraints serves as a versatile tool for propagating various constraints efficiently. Two notable examples in this regard are:

- Table constraint.
- Formal language-based constraints.

6.1 Table Constraint

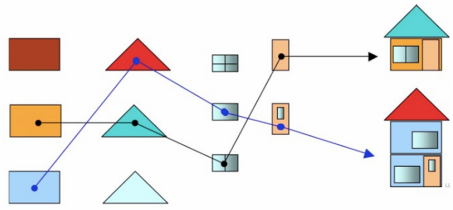
The Table Constraint, also known as the Extensional Constraint, involves a set of tuples defined by $C(X_1, X_2) = \{(0,0), (0,2), (1,3), (2,1)\}$. This set represents valid combinations of values for the variables X_1 and X_2.

To ensure GAC for the Table Constraint, various algorithms have been developed. These algorithms offer efficiency advantages over a brute-force approach with a time complexity of $O(|D(X_1)| \cdot |D(X_2)| \cdot \ldots \cdot |D(X_k)|)$, where $D(X_i)$ denotes the domain of variable X_i. Moreover, these algorithms are more effective than decomposition methods.

For instance, consider the decomposition of the Table Constraint into individual constraints, as expressed by the logical disjunction of conjunctions:

$$(X_1 = 0 \land X_2 = 2 \land X_3 = 2) \lor (X_1 = 1 \land X_2 = 1 \land X_3 = 2) \lor (X_1 = 1 \land X_2 = 2 \land X_3 = 3)$$

Product configuration problems



These problems are characterized by compatibility constraints imposed on the product components. Essentially, not all combinations of components are viable, and certain configurations may be incompatible due to specific constraints.

It's important to note that the nature of compatibility in product configuration problems is not necessarily a simple pairwise relationship. The constraints may involve more intricate dependencies among multiple components, making the determination of valid configurations a complex task.

■ **Example 6.1 — A Configuration Problem.** This is an example of a Configuration Problem, focusing on the configuration of valid hardware products. The compatibility of components is defined in a table that outlines the acceptable combinations for each product:

Products	Motherboard	CPU	Freq	RAM	Hard drive
Product1	TypeA	Intel	2GHz	5GB	100GB
Product2	TypeB	Intel	3GHz	8GB	200GB
Product3	TypeB	AMD	2GHz	5GB	200GB
...					

Table 6.1: Compatible Components for Hardware Products

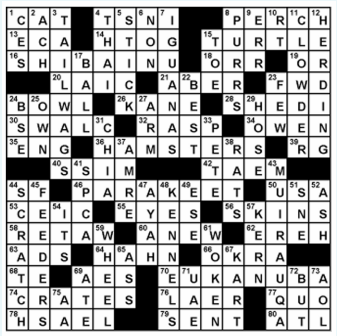
In this scenario, let's assume we have a set of products denoted as P_i, each requiring configuration with five components: motherboard (X_{i1}), CPU (X_{i2}), Frequency (X_{i3}), RAM (X_{i4}), and hard drive (X_{i5}).

For every product P_i, a constraint is established using the

$$\texttt{table}([X_{i1}, X_{i2}, X_{i3}, X_{i4}, X_{i5}], Products)$$

statement, ensuring that the configuration adheres to the compatibility specifications outlined in the Products table. ■

■ **Example 6.2 — Crossword puzzle.**



The validity of words is defined by establishing constraints based on the compatibility of letters, effectively forming a dictionary. Consider the following examples:

1. $\texttt{table}([X_1, X_2, X_3], \text{dictionary})$: This constraint ensures that the combination of letters in positions X_1, X_2, and X_3 forms a valid word according to the dictionary.
2. $\texttt{table}([X_1, X_{13}, X_{16}], \text{dictionary})$: Similarly, this constraint specifies the validity of the word formed by the letters in positions X_1, X_{13}, and X_{16} based on the provided dictionary.
3. $\texttt{table}([X_4, X_5, X_6, X_7], \text{dictionary})$: Extending this approach, the combination of letters in positions X_4, X_5, X_6, and X_7 is constrained to represent a valid word in the dictionary.

This methodology is applied iteratively throughout the puzzle, ensuring that each word conforms

to the acceptable combinations of letters. There is no straightforward method to determine acceptable words other than to compile them into a comprehensive table, effectively capturing the compatibility constraints within the puzzle. ■

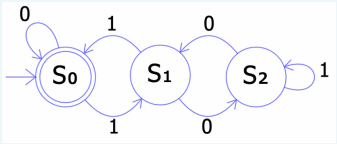
6.2 Formal Language-based Constraints

The table constraint may present challenges, particularly in situations where precomputing all possible solutions is either impractical or impossible. To overcome this limitation, a valuable alternative is to leverage a deterministic finite-state automaton (DFA) to define the solutions.

Unlike the table constraint, which necessitates the enumeration of all conceivable solutions, a DFA offers a more compact and structured representation of valid assignments. This becomes particularly advantageous when valid assignments are required to adhere to specific patterns or follow predetermined rules.

A Deterministic Finite State Automaton (DFSA) is a finite-state machine designed to either accept or reject a given string of symbols. Its operation involves traversing through a sequence of states uniquely determined by the input string. DFSA are used to recognize regular languages.

■ **Example 6.3** As an example, let's consider a DFSA designed to accept binary numbers that are multiples of 3:



This DFSA acknowledges certain strings as valid, representing binary numbers that are multiples of 3, while rejecting others. For instance:

- Accepted strings: 0, 11, 110, 1100, 1001, 10111101, and so forth.
- Not accepted strings: 10, 100, 101, 10100, and others.

■

6.2.1 Regular Constraint

The concept of `regular` constraint involves the utilization of a Deterministic Finite State Automaton (DFSA) to define and enforce constraints on a set of variables. A DFSA is represented by a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where:

- Q: a finite set of states.
- Σ: a set of symbols (referred to as the alphabet).
- δ: a partial transition function $Q \times \Sigma \rightarrow Q$ that determines state transitions.
- q_0: the initial state, belonging to Q.
- $F \subseteq Q$: a set of accepting (final) states.

The `regular` constraint, denoted as `regular`$([X_1, X_2, \ldots, X_k], A)$, is satisfied if and only if the sequence of variables $< X_1, X_2, \ldots, X_k >$ forms a string that is accepted by the DFSA A. In other words, the DFSA defines a language, and the `regular` constraint ensures that the assignment of values to the specified variables adheres to the accepted strings defined by the automaton.

`regular` constraint is particularly useful in sequencing and rostering problems, beside many

constraints are instances of regular (e.g. `among`, `lex`, `precedence`, `stretch`, ...). `regular` constraint has an efficient GAC propagation with a dedicated algorithm and a decomposition into a sequence of ternary constraints.

■ **Example 6.4 — A rostering problem.** Let's delve into an example of applying the Regular Constraint to address a nurse rostering problem, where shifts are subject to specific regulations. Consider the following constraints:

- Successive night shifts must be limited.
- Each nurse is scheduled for each day either on a day shift (d), night shift (n), or off (o).
- In each four-day period, a nurse must have at least one day off.
- No nurse can be scheduled for 3 night shifts in a row.

To model this problem using a Deterministic Finite State Automaton (DFSA), we define the DFSA as follows:

- $Q = \{q_1, \ldots, q_6\}$: a finite set of states.
- $\Sigma = \{d, n, o\}$: the alphabet representing the possible shifts.
- δ: transition function defined by the table provided.

	d	n	o
1	2	3	1
2	4	4	1
3	4	5	1
4	6	6	1
5	6	0	1
6	0	0	1

- $q_0 : q_1$: the initial state.
- $F = Q = \{q_1, \ldots, q_6\}$: all states are accepting (final) states.

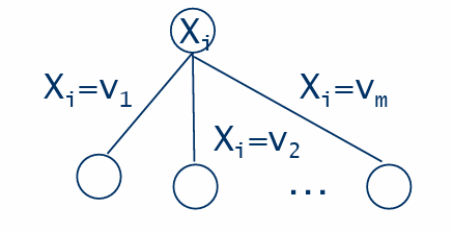
Assume nurses N_i are scheduled for 30 days, denoted as $[D_{i1}, \ldots, D_{i30}]$. To enforce the constraints, we post $\texttt{regular}([D_{i1}, \ldots, D_{i30}], A)$ for each nurse N_i, ensuring that the sequence of shifts for each nurse aligns with the language recognized by the DFSA A. This formulation guarantees compliance with the specified regulations regarding shift scheduling for the nurse rostering problem. ■

III Search

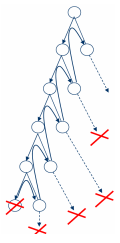
7. Introduction

7.1 Backtracking Tree Search

A Backtracking Tree Search (BTS) serves as a systematic method for exploring variable-value combinations to find solutions or prove unsatisfiability. The basic components of BTS are nodes representing variables and branches representing decisions on those variables.



Each node in the tree corresponds to a variable X_i, and each branch represents a decision regarding the value of X_i. For instance, the enumeration of values from the domain $D(X_i)$ constitutes the decision process. Variables are instantiated sequentially, and the traversal is by default a depth-first search.



In the **absence of propagation**, BTS systematically enumerates all possible variable-value combinations through a backtracking tree search. At each step, a value is guessed for a variable, and during the search, constraints are examined to eliminate inconsistent values from the domains of future (unassigned) variables.

This process leads to the shrinking of future variable domains based on constraint satisfaction. The algorithm continues until either a solution is found or unsatisfiability is proven. In the case of a

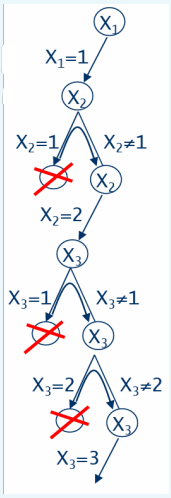
dead-end, chronological backtracking is employed, retracting the most recently made branching decision.

BTS without propagation is a systematic search approach that eventually finds a solution or proves unsatisfiability. However, its complexity is exponential, with a time complexity of $O(d^n)$, where d is the maximum domain size and n is the number of variables.

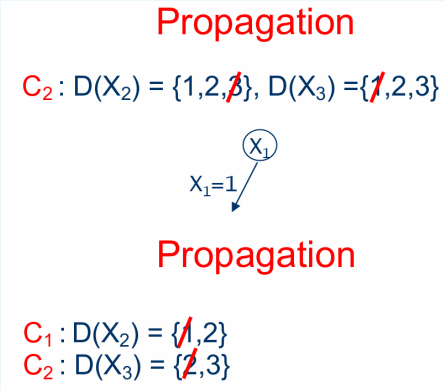
■ **Example 7.1 — BTS without propagation VS BTS with propagation.** Consider the following CSP:

- $D(X_1) = \{1, 2\}, \qquad D(X_2) = D(X_3) = \{1, 2, 3\}$
- $C_1 : X_1 \neq X_2, \qquad C_2 : X_2 < X_3$

If we solve this problem with a BTS without propagation we simply enumerates all possible variable-value combinations, until a solution is found. The following figure shows the BTS:



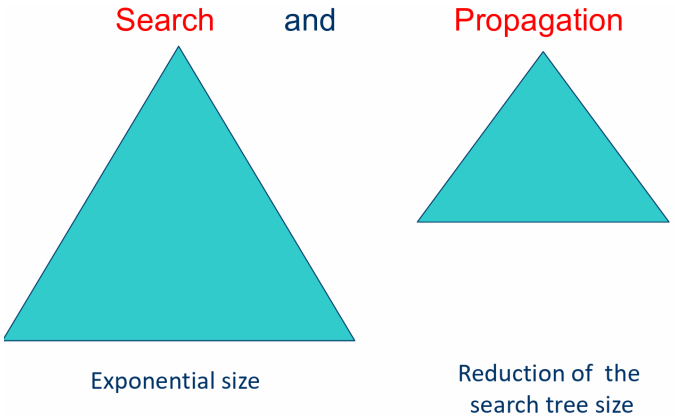
Instead if propagation is used, values that do not satisfy the constraint are removed from the variable domain. This results in the following situation:



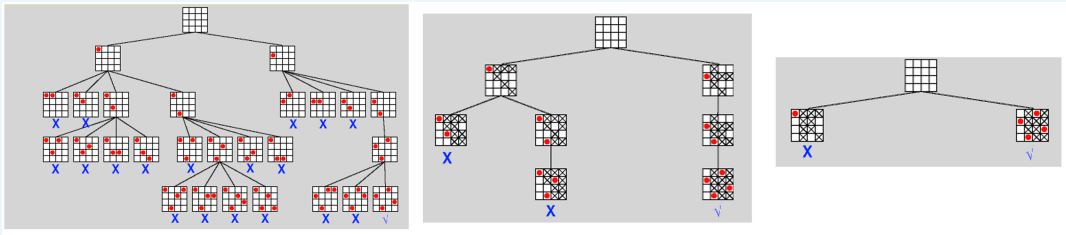
In this case before taking the first choice in BTS, propagation in constraint C_2 is used and we remove 3 from $D(X_2)$ and 1 from $D(X_3)$. Variable X_1 is assigned the value 1, and re-running the constraint propagation removes 1 from $D(X_1)$ and 2 from $D(X_3)$. This results in a solution since there is only one value in the domain of all variables. ■

As could be seen from the example, propagation improves the efficiency in solving a CSP, as the

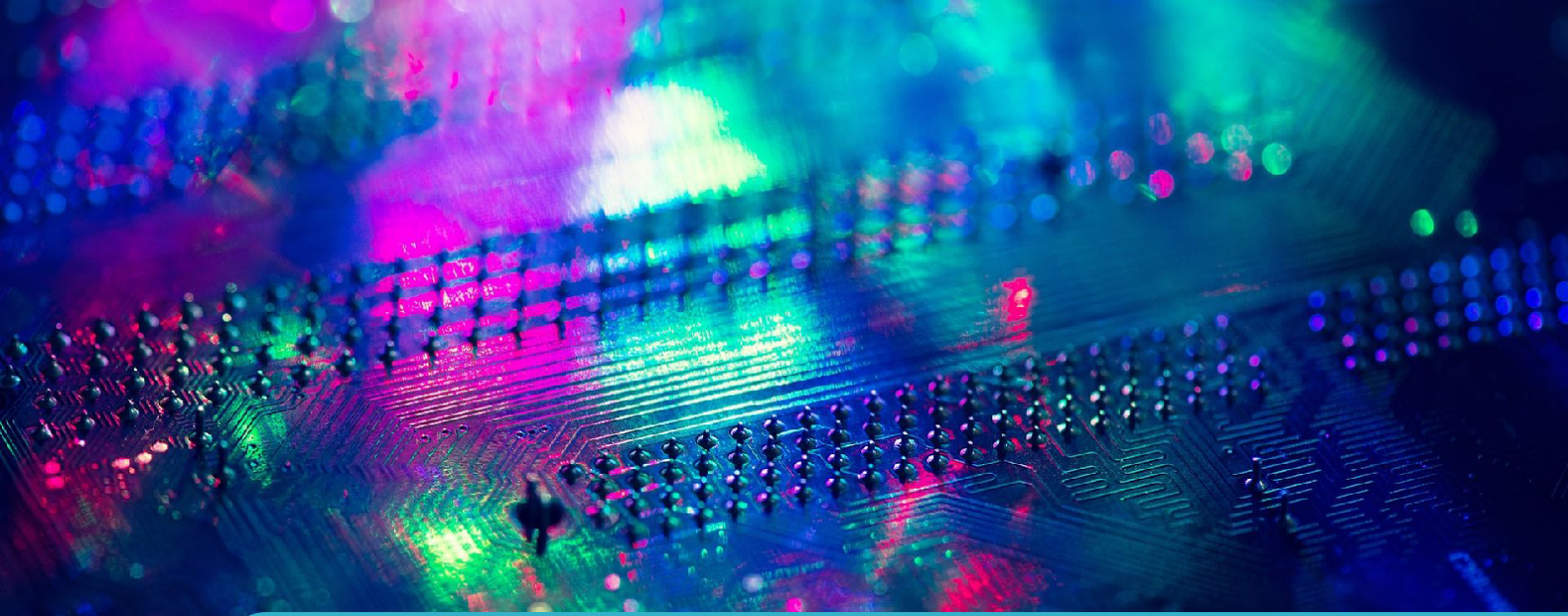
solution space is reduced compared to simply enumerating all possible values as is done without propagation.



■ **Example 7.2** This is another example of BTS tree with and without propagation



■



8. Depth-first Search (DFS)

8.1 Branching Decision

Branching decisions play a crucial role in the exploration of solution spaces. Typically, these decisions involve posting a unary constraint on a selected variable X_i. The process of branching often entails the **enumeration** or **labeling** of values from the domain $D(X_i)$, and the specific type of branching determines the structure of the exploration.

8.1.1 Enumeration (or Labeling)

d-way Branching

In d-way branching, one branch is generated for each possible value in the domain $D(X_i)$. This approach explores all potential values for the chosen variable, leading to a branching factor equal to the size of the domain.

2-way Branching

For 2-way branching, two branches are generated: one by setting $X_i = v$ and the other by excluding $X_i = v$ for some specific value $v \in D(X_i)$. This binary branching strategy narrows down the possibilities more rapidly than d-way branching.

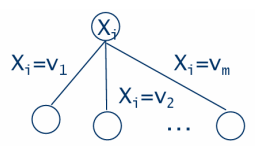


Figure 8.1: d-way branching

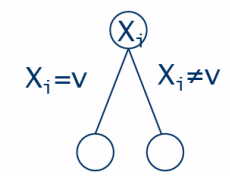


Figure 8.2: 2-way branching

8.1.2 Domain Partitioning of $D(X_i)$

k-way Branching

In k-way branching, where k is greater than 2, one branch is generated for each partition S_j of the domain $D(X_i)$.

2-way Branching (Subset)

Similar to 2-way branching, but in this case, two branches are generated by selecting or excluding a subset $S \subseteq D(X_i)$.

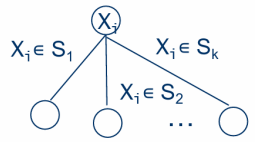


Figure 8.3: k-way branching

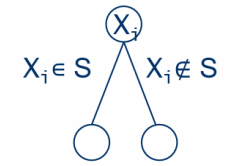


Figure 8.4: 2-way branching(subset)

8.2 Branching/Search Heuristics

Branching and search heuristics assist in making decisions about which variable to choose and which value(s) to assign, thus influencing the efficiency of the search process. Variable and value ordering heuristics (VVO) play a significant role in this regard, and they can be classified into static and dynamic categories.

8.2.1 Static Variable Ordering Heuristics

Static heuristics associate a variable with each level, and branches are generated in the same order throughout the entire search tree. These heuristics are calculated once and for all before the search starts, making them cost-effective to evaluate.

Some examples of static generic VVO heuristics include:

1. Lexicographic Ordering: variables are ordered based on their definition in a sequence: $X_1, X_2, \ldots, X_n$.
2. Top-Down, Left-to-Right Ordering: applicable in the case of a matrix of variables, arranged row by row:

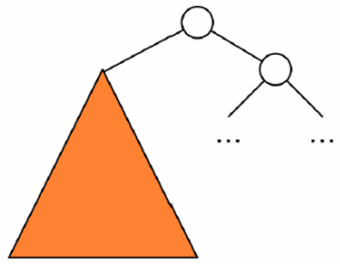
$$\begin{matrix} X_{11} & X_{12} & \ldots & X_{1m} \\ X_{21} & X_{22} & \ldots & X_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ X_{n1} & X_{n2} & \ldots & X_{nm} \end{matrix}$$

8.2.2 Dynamic Variable Ordering Heuristics

In dynamic variable ordering heuristics, decisions about which variable and branch to consider are made dynamically during the search. This flexibility comes at the cost of increased computation, as the heuristics take into account the current state of the search tree.

Search Heuristics

For feasible problems, search heuristics guide the choice of variables and values likely to lead to a solution. However, there's no guarantee of feasibility in general. If a mistake is made, resulting in an infeasible sub-problem, the entire sub-tree needs to be explored before backtracking. The goal is to explore the infeasible sub-tree as quickly as possible.



In the context of infeasible problems, the Fail-first (FF) principle is employed. The goal is to try first where failure is most likely to occur, aiming to quickly prove that the current sub-tree has no feasible solutions. This approach involves a trade-off:

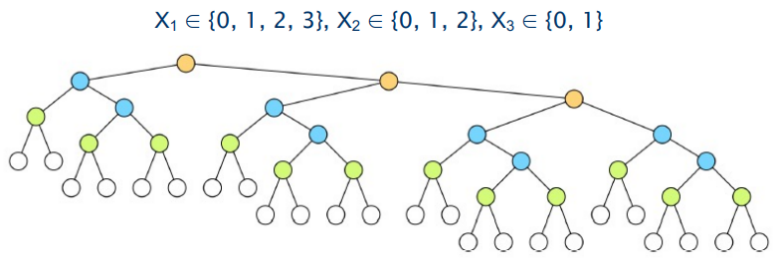
- Choose the variable most likely to cause failure.
- Choose the value most likely to be part of a solution (least constrained value).

The emphasis is on Variable Ordering Heuristics (VOHs) in the context of infeasible problems. To backtrack from an infeasible sub-problem, it's necessary to explore all values in the domain of a variable, making the choice of variable ordering crucial in efficiently exploring and proving the infeasibility of a sub-tree.

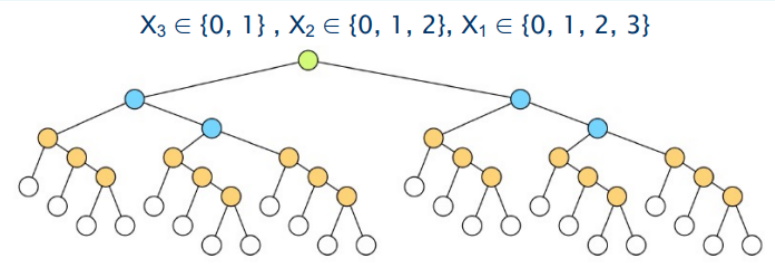
Minimum domain heuristic

One heuristic is the Minimum Domain (dom), where the next variable to be considered is chosen based on having the smallest domain size. The rationale behind this approach is to minimize the size of the search tree, potentially leading to more efficient problem-solving.

■ **Example 8.1** Consider the following variable order: X_1, X_2, X_3, with $X_1 \in 0, 1, 2, 3$, $X_2 \in 0, 1, 2$, and $X_3 \in 0, 1$ (no Minimum Domain).



Now, let's consider a different order: X_3, X_2, X_1, with $X_3 \in 0, 1$, $X_2 \in 0, 1, 2$, and $X_1 \in 0, 1, 2, 3$ (utilizing Minimum Domain).



If propagation prunes a value at depth 1, the impact is more substantial with the second ordering.

The effect of pruning is significantly stronger with the Minimum Domain ordering, showcasing the importance of choosing an effective heuristic to guide the search process. ■

Most Constrained Heuristic (deg)

Another valuable heuristic in Constraint Programming is the **Most Constrained** (deg) principle. This heuristic dictates the choice of the next variable based on the one involved in the most significant number of constraints. The underlying idea is to maximize constraint propagation, potentially leading to more informed decisions during the search process.

To strike a balance between minimizing the domain size (dom) and maximizing the number of constraints (deg), a combination of these heuristics can be employed.

Weighted Degree Heuristic

Weighted Degree Heuristic considers constraints as entities with initially set weights of 1. During the propagation of a constraint, if it fails, the weight of that particular constraint is incremented by 1.

The **weighted degree** of a variable X_i is then calculated as the sum of weights of all constraints in which X_i participates:

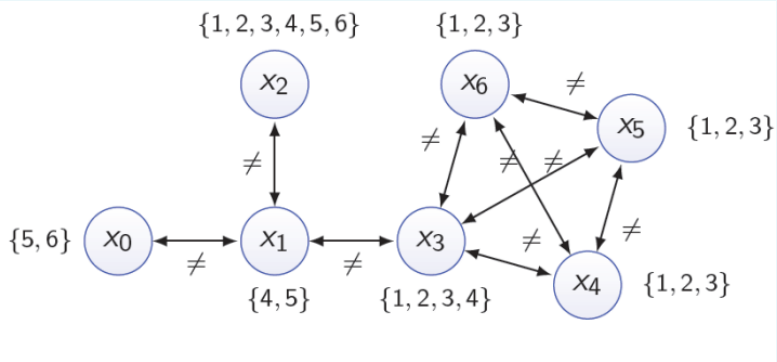
$$w(X_i) = \sum_{c|X_i \in X(c)} w(c)$$

With this information, the heuristic (called Domain over Weighted Degree (**domWdeg**)) is employed. In this heuristic, the variable X_i is chosen based on the minimum ratio of its domain size to the weighted degree:

$$domWdeg(X_i) = \frac{dom(X_i)}{w(X_i)}$$

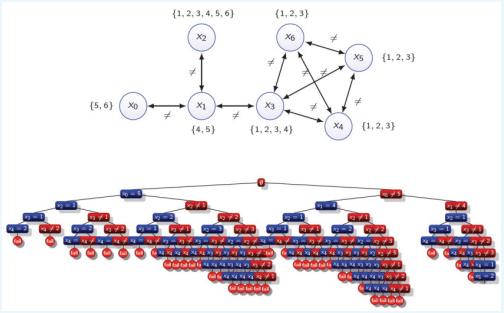
The rationale behind this approach is to balance the consideration of both the size of the domain and the impact of constraints on a variable. By selecting variables with a smaller domain size relative to their weighted degree, this heuristic aims to guide the search process towards more efficient search.

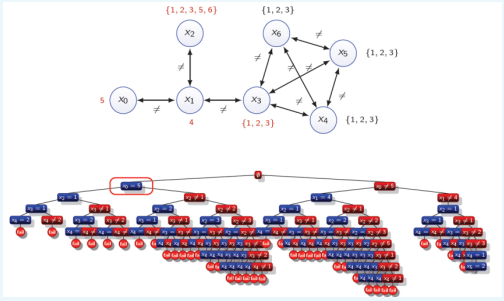
■ **Example 8.2** Let's delve into an illustrative example involving map coloring.



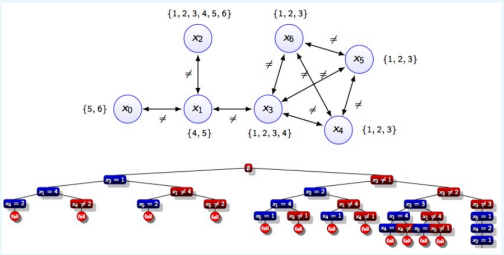
In this scenario, we have a map with regions that need to be colored in a way that adjacent regions have different colors. The process involves maintaining Arc-Consistency as we explore the solution space with 2-way branching. The choice of heuristics becomes crucial in guiding the search and ensuring efficient constraint propagation.

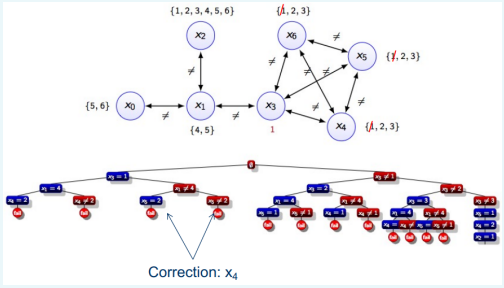
With **lexicographic ordering** we have:



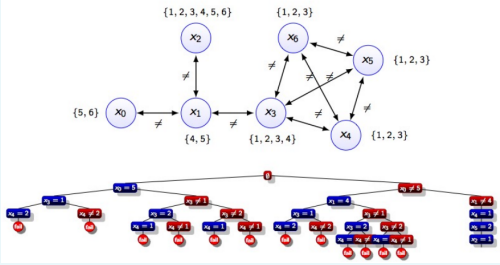


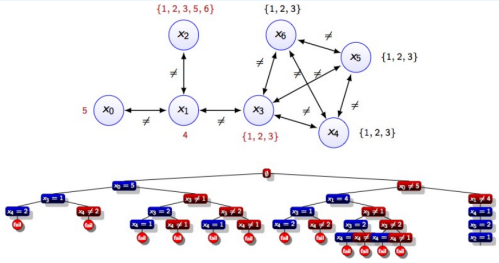
With **Maximum Degree** ordering we have:



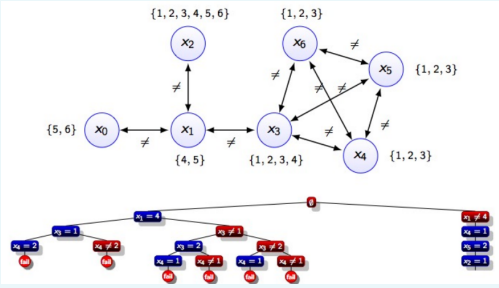


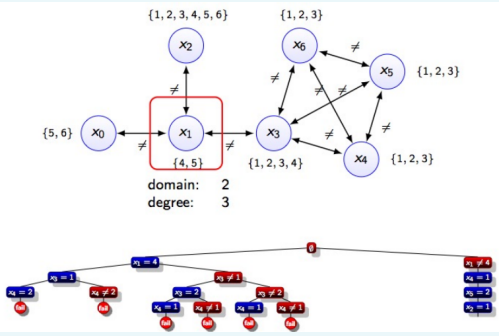
With **Minimum Domain** ordering we have:

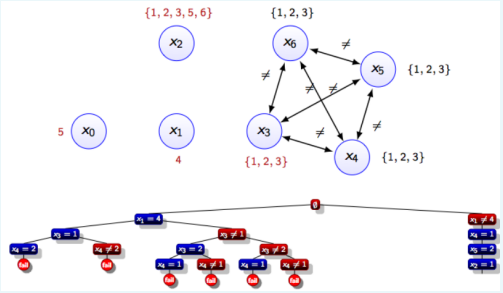


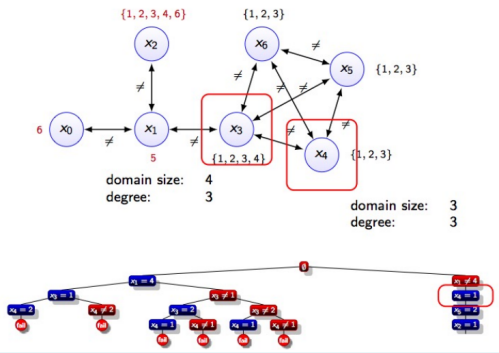


With **Minimum Domain/Degree** ordering we have:





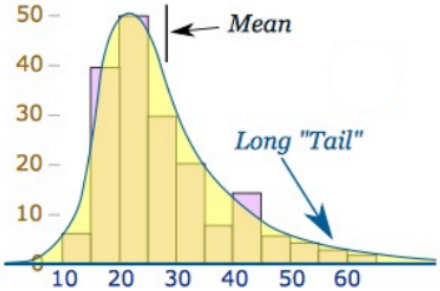




As can be seen, searching with domWdeg in this example produces the smallest Backtracking Tree of all. ■

8.2.3 Heavy Tail Behaviour

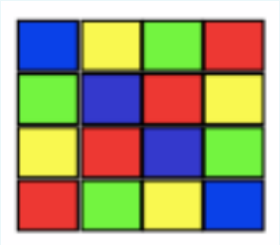
Solving CSP, a phenomenon known as **Heavy Tail Behaviour** often comes into play. This phenomenon is characterized by the presence of exceptionally challenging instances that demand an unusually long time to be resolved.



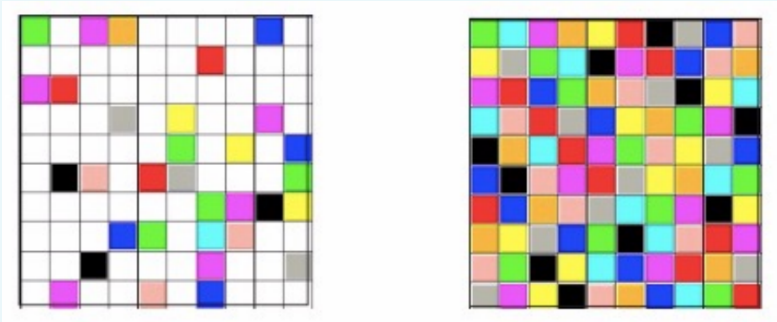
This heavy-tail effect significantly impacts the runtime distributions for a set of instances, creating variations in the solving time for different cases. Importantly, this behavior is not inherent to the instances themselves; rather, it is observed even when repeatedly running the same instance with variations in solver parameters, such as altering the variable ordering.

The intuitive reason behind heavy tail behavior lies in the early stages of the search process. **If a mistake is made early on**, the solver may become trapped in a **problematic subtree**, leading to a significant increase in solving time. This effect is reminiscent of the puzzle example, where different heuristics can result in unfavorable mistakes on various instances.

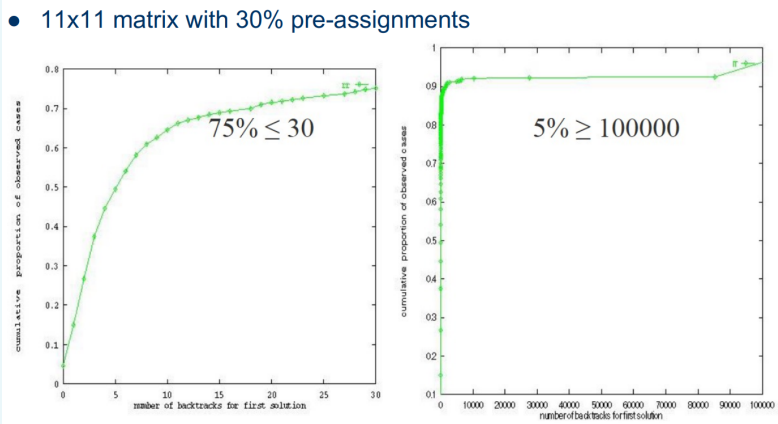
■ **Example 8.3 — Quasigroup problem.** A **Latin square** of order n is a $n \times n$ matrix adorned with n distinct colors, it possesses the remarkable property that each color appears exactly once in every row and every column. It finds applications in diverse fields such as fiber optic networks, the design of statistical experiments, and the intricate domains of scheduling and timetabling.



Given a partial assignment of colors, the question that arises is whether this partial Latin square (quasigroup) can be extended or completed to form a valid Latin square. The challenge lies in filling in the missing pieces of the puzzle while adhering to the rules of Latin squares.



- 11x11 matrix with 30% pre-assignments



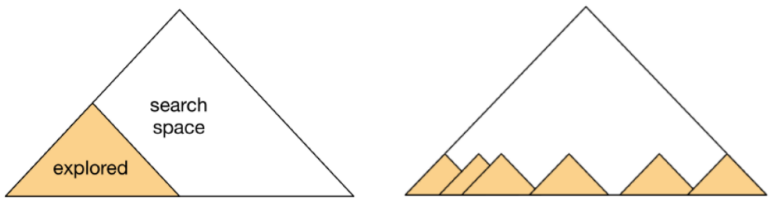
As depicted in the graphical representation, heavy tail behavior may manifest in the quasi-group completion problem. Instances exist where the solving process encounters exceptionally challenging situations, leading to prolonged solving times. ■

8.3 Randomization and Restart

A notable observation is that these mistakes appear to be seemingly random. To address this randomness, certain strategies such as **randomization** and **restarting** are employed in the search process. Randomization involves introducing a degree of randomness in the search, such as picking variables or values at random and breaking ties randomly. While using the same random seed ensures the exploration of the same tree, it prevents the solver from traversing identical subproblems in the same way.

Restarting the search after a certain resource threshold, like the number of visited nodes, is another approach. In subsequent runs, the search process is restarted differently, incorporating elements of randomization and learning from the accumulated experiences of previous runs. These strategies aim to mitigate the impact of heavy tail behavior, providing more robust and adaptable approaches to constraint programming.

The incorporation of randomization and restarts proves to be a strategy in addressing the vast variance in solver performance. This strategic approach effectively mitigates the heavy tail behavior, ensuring a more stable and predictable solving process.



Several restart strategies come into play, each tailored to address specific aspects of the solving process:

- **Constant restart**: The solver initiates a restart after utilizing a predefined quantity of resources.
- **Geometric restart**: Restarting occurs after L resources, with subsequent restart limits following a geometric progression ($\alpha * L$, $\alpha^2 * L$, and so on).
- **Luby restart**: Restarting is triggered after consuming $s[i] * L$ resources, where $s[i]$ corresponds to the i-th element in the Luby sequence. This sequence repeats two copies of the pattern ending in 2^i before adding the number 2^{i+1}.

The **domWdeg** (domain over weighted degree) heuristic synergizes effectively with restarts. Fail counts accumulated during the solving process can be carried over to subsequent runs, enhancing the solver's ability to navigate through complex instances. When coupled with a random choice of values, the domWdeg heuristic becomes a formidable ally, contributing to highly effective search strategies.

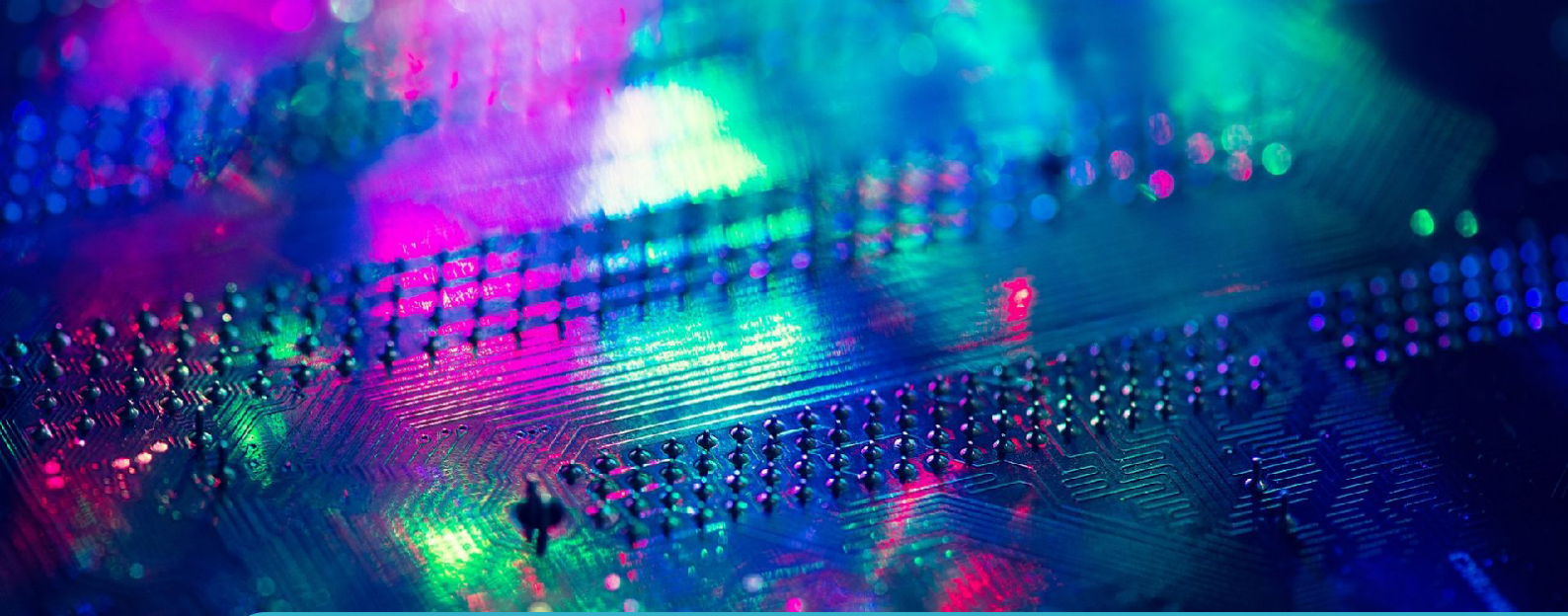
8.4 Problem with DFS

Depth-First Search (DFS) approach tends to place a substantial burden on heuristics, especially in the early stages of the search process.

The problem lies in the fact that heuristics often prove to be more accurate when applied at deeper nodes in the search tree. Unfortunately, the initial decisions made during the early steps of DFS are prone to inaccuracy. This characteristic poses a significant drawback, as DFS heavily relies on heuristics early on and places a lighter burden on them as the search delves deeper.

This skewed distribution of burden becomes problematic, as any mistakes made near the root of the tree can have lasting and costly repercussions. To illustrate, consider the analogy of solving a puzzle, where an erroneous move early on can lead to a complex and convoluted problem that is challenging to rectify.

Recognizing these limitations, the exploration of an alternative strategy becomes imperative. One such strategy that garners interest is the Best-First Search (BFS). In contrast to DFS, BFS adopts a more nuanced approach by prioritizing the exploration of nodes based on heuristic evaluations. This prioritization allows BFS to focus on the most promising nodes from the outset, potentially mitigating the challenges associated with DFS.



9. Best-First Search (BFS)

9.1 Limited Discrepancy Search

A **discrepancy** is identified as any decision within a search tree that deviates from the heuristic, typically observed as taking a right branch out of a node.

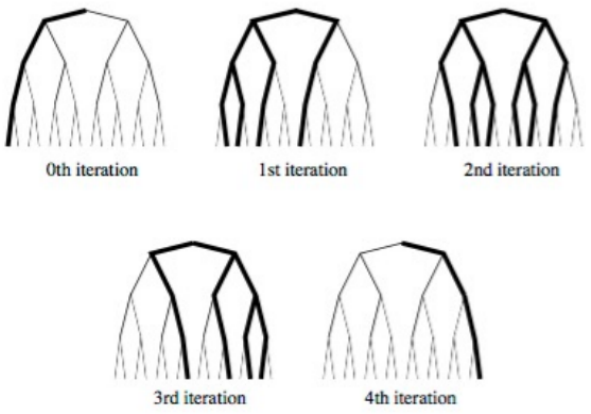
LDS strategically places **trust in the heuristic** and assigns **priority to left** branches during the exploration process. The key feature of LDS lies in its iterative nature, systematically searching the tree while incrementally considering an increasing number of discrepancies.

On each iteration of LDS:

- During the 0^{th} iteration, the search focuses on exploring the leftmost branches exclusively.
- During the 1^{th} iteration, progressively introduce discrepancies, limiting the exploration of left branches to correct a small number of heuristic mistakes.
- On the i^{th} iteration, LDS visits all leaf nodes featuring precisely i discrepancies.

The underlying motivation is rooted in the expectation that the branching heuristic may have made a few errors, and LDS offers an efficient mechanism to rectify a limited number of mistakes at a relatively low cost. In contrast, traditional Depth-First Search (DFS) necessitates extensive exploration of the tree before addressing early mistakes.

To illustrate the process, consider the following diagram:



9.1.1 Problems with LDS

Despite its effectiveness, LDS encounters certain problems:

All discrepancies are treated uniformly, regardless of their depth in the search tree. Heuristics, being less informed, tend to make more mistakes at the top of the tree. There is a recognized need to explore discrepancies at the upper levels of the tree before delving into those at the lower levels for a more informed correction process.

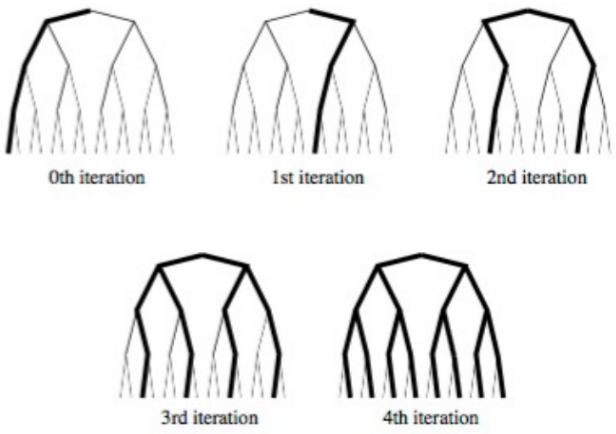
9.2 Depth-bounded Discrepancy Search (DDS)

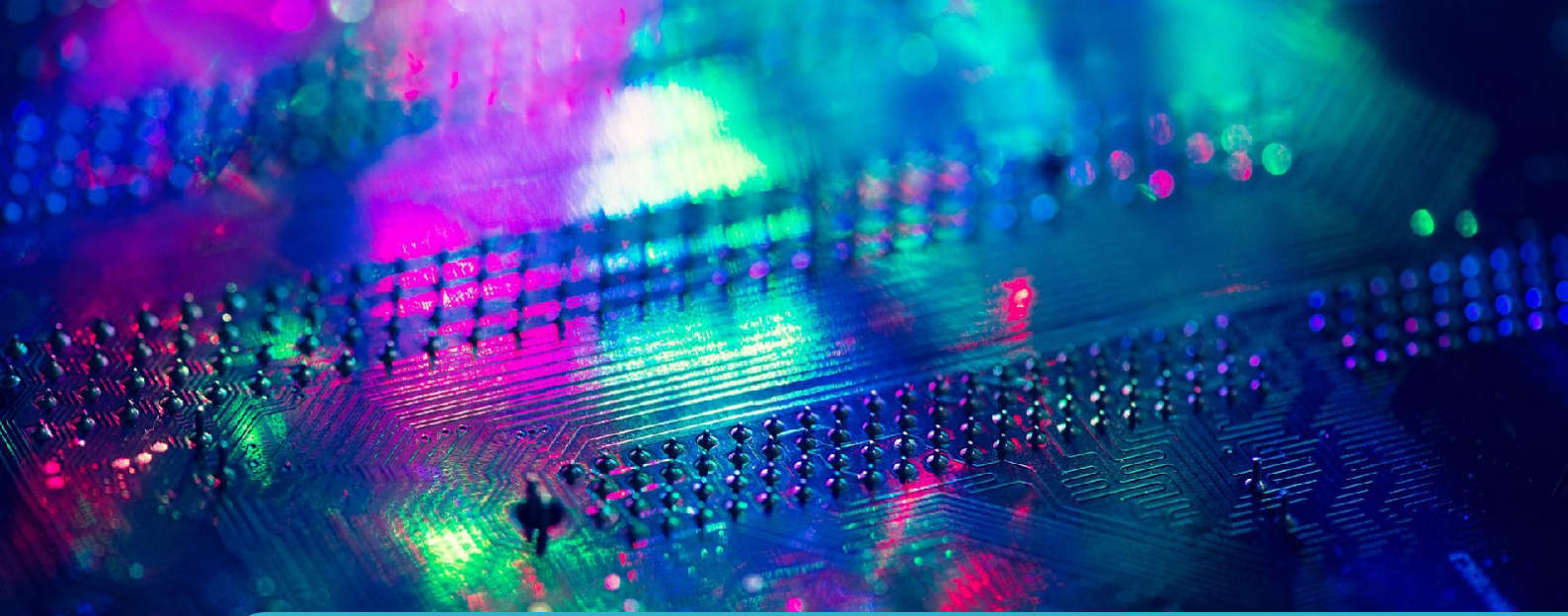
Depth-bounded Discrepancy Search (DDS) strategically biases the search towards discrepancies occurring at higher levels in the tree by imposing an iteratively increasing depth bound. The fundamental principle underlying DDS is to restrict discrepancies below a certain depth, beyond which the heuristic is trusted to guide the search.

Key characteristics of DDS:

- During the 0^{th} iteration, DDS aligns with the behavior of LDS.
- On the i^{th} iteration, DDS specifically explores branches featuring discrepancies at a depth of i or less. This ensures a focused exploration of discrepancies at shallower depths, allowing DDS to address a varying number of discrepancies at different levels.
- At depths lower than the current iteration, DDS widens its exploration to encompass more discrepancies.
- Conversely, at depths exceeding the current iteration, DDS relies on the heuristic to guide the search.

To visually illustrate the process, refer to the diagram below:





10. Constraint Optimization Problems

Constraint Optimization Problems (COPs) introduce a layer of sophistication by incorporating an optimization criterion to CSP. This criterion could be diverse, ranging from minimizing costs and finding the shortest distance to determining the fastest route or maximizing profit.

Formally expressed as $\langle X, D, C, f \rangle$, where f encapsulates the optimization criterion as an objective function or variable, the primary goal is to minimize f or, equivalently, maximize $-f$.

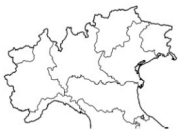
There are some methods for Solving COPs:

- **Enumeration**: the simplest approach involves enumerating solutions, systematically exploring the solution space. However, this method faces scalability challenges when dealing with a vast number of potential solutions.
- **Search over** $D(f)$: an alternative method involves searching over the domain of the optimization criterion ($D(f)$). This approach allows for a more targeted exploration of potential solutions.
- **Branch & Bound**: method employed for COPs is the Branch & Bound algorithm. This algorithm tackles a sequence of Constraint Satisfaction Problems (CSPs) through a unified search tree while incorporating bounding techniques to guide the exploration.

10.1 Searching over D(f)

10.1.1 Destructive lower bound

The **Destructive Lower Bound** strategy iteratively traverses the values v in $D(f)$, beginning from the $\min(D(f))$. At each iteration, the constraint $f \leq v$ is posted and the CSP is solved. The first feasible solution encountered is guaranteed to be optimal. Despite its effectiveness, this method discards intermediate computation results.



- Solve with 1 colour → fail
- Solve with 2 colours → fail
- Solve with 3 colours → success (optimal)

10.1.2 Destructive Upper Bound

Conversely, the **Destructive Upper Bound** approach iteratively explores values v in $D(f)$ from the $\max(D(f))$. At each iteration, the constraint $f \leq v$ is posted, and the value v is updated. The last solution found before the problem becomes infeasible serves as the proven optimal solution.

- Solve with 8 colours → success with 5 colours
- Solve with 4 colours → success with 4 colours
- Solve with 3 colours → success with 3 colours
- Solve with 2 colours → fail (optimality with 3 colours proven)

10.1.3 Destructive Lower Bound vs. Destructive Upper Bound

Comparing the two approaches:

- Destructive Lower Bound
  - **Pros**: Tighter constraints leading to more propagation.
  - **Pros**:Provides lower bounds.
  - **Cons**: Not an any-time algorithm, involves small steps.
- Destructive Upper Bound:
  - Pros: An any-time algorithm with larger steps.
  - Cons: Less propagation, no lower bounds.

A potential solution is to combine the strengths of both approaches, leading to a binary search over $D(f)$.

10.1.4 Binary Search

The Binary Search strategy is rooted in maintaining both a feasible upper bound (ub) and an infeasible lower bound (lb) for the optimization criterion (f). This method operates by iteratively solving the CSP with the constraint $lb < f < (lb + ub)/2$. If the solution is feasible, ub is updated; if infeasible, lb is updated. The process continues until a solution with $f = lb + 1$ is found. This approach serves as a compromise between destructive lower and upper bounding.

Key Features:

- Anytime algorithm.
- Provides lower bounds.
- Tight(ish) constraints on f lead to effective propagation.
- Involves large steps in the search.

However, one notable drawback is that a significant portion of information is discarded between each attempt, resulting in a considerable amount of repeated work.

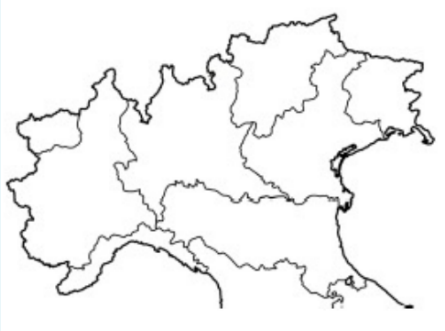
10.2 Branch & Bound Algorithm

The Branch & Bound algorithm presents a solution for tackling a sequence of CSPs within a unified search tree while incorporating bounding techniques. The key steps include:

1. Whenever a feasible solution is found, a new bounding constraint is posted. This constraint ensures that any future solution must surpass the current upper bound.
2. The algorithm then backtracks and searches for a new solution, integrating the additional bounding constraint within the same search tree.
3. This process is repeated until infeasibility is encountered, at which point the last solution found is deemed optimal.

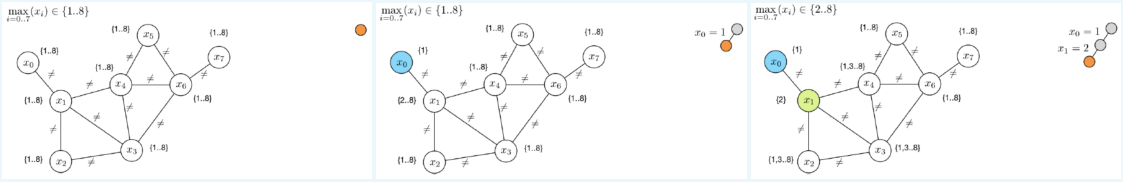
■ **Example 10.1 — Optimal Map Coloring.** Consider the problem of optimally coloring a map to ensure neighboring regions have different colors. The goal is to determine the minimum number of colors required for this task.

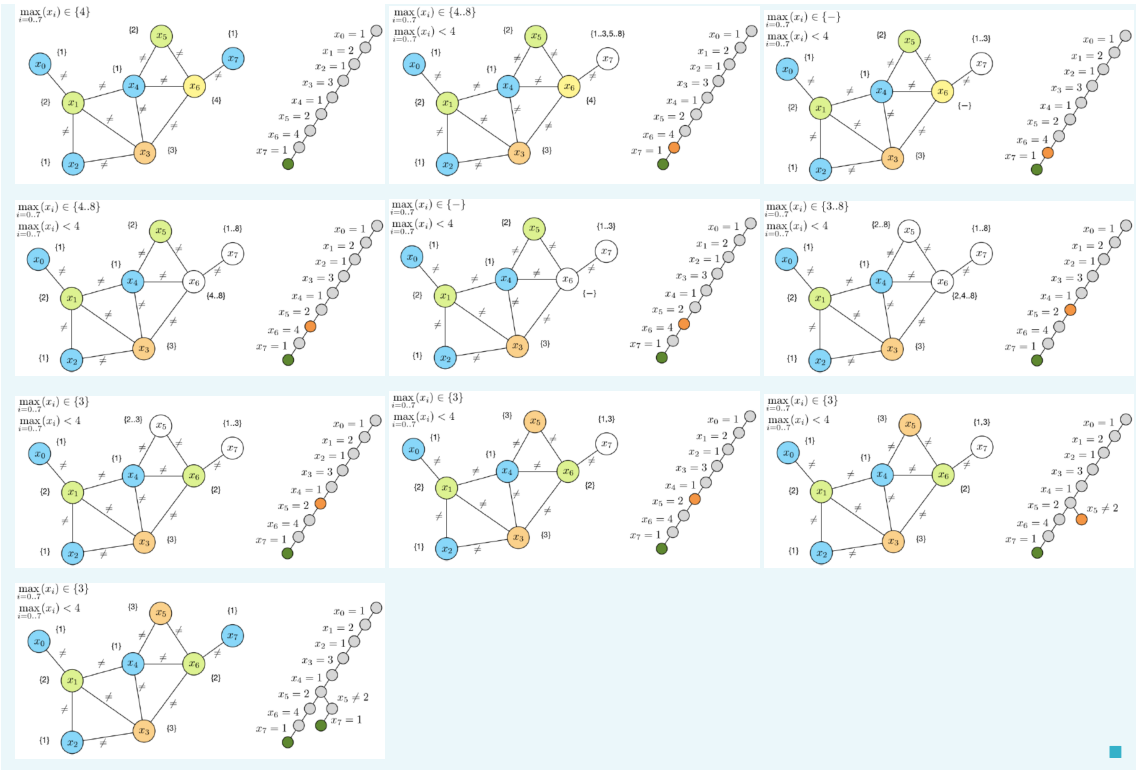
- **Variables and Domains**:
  - Introduce a variable X_i for each of the n regions, with a domain $[1..n]$ representing the available colors.
- **Constraints**:
  - Enforce the constraint $X_i \neq X_j$ for each pair of neighboring regions i and j to ensure adjacent regions have distinct colors.
- **Objective Function/Variable**:
  - Define an objective function f, where $f = \max(X_i)$. The objective is to minimize f by minimizing the maximum color assigned to any region.



The task is to find a coloring scheme that satisfies the constraints while minimizing the maximum color assigned, reflecting the optimal solution to the map coloring problem.

The following figures show how Branch&Bound algorithm solves the problem.



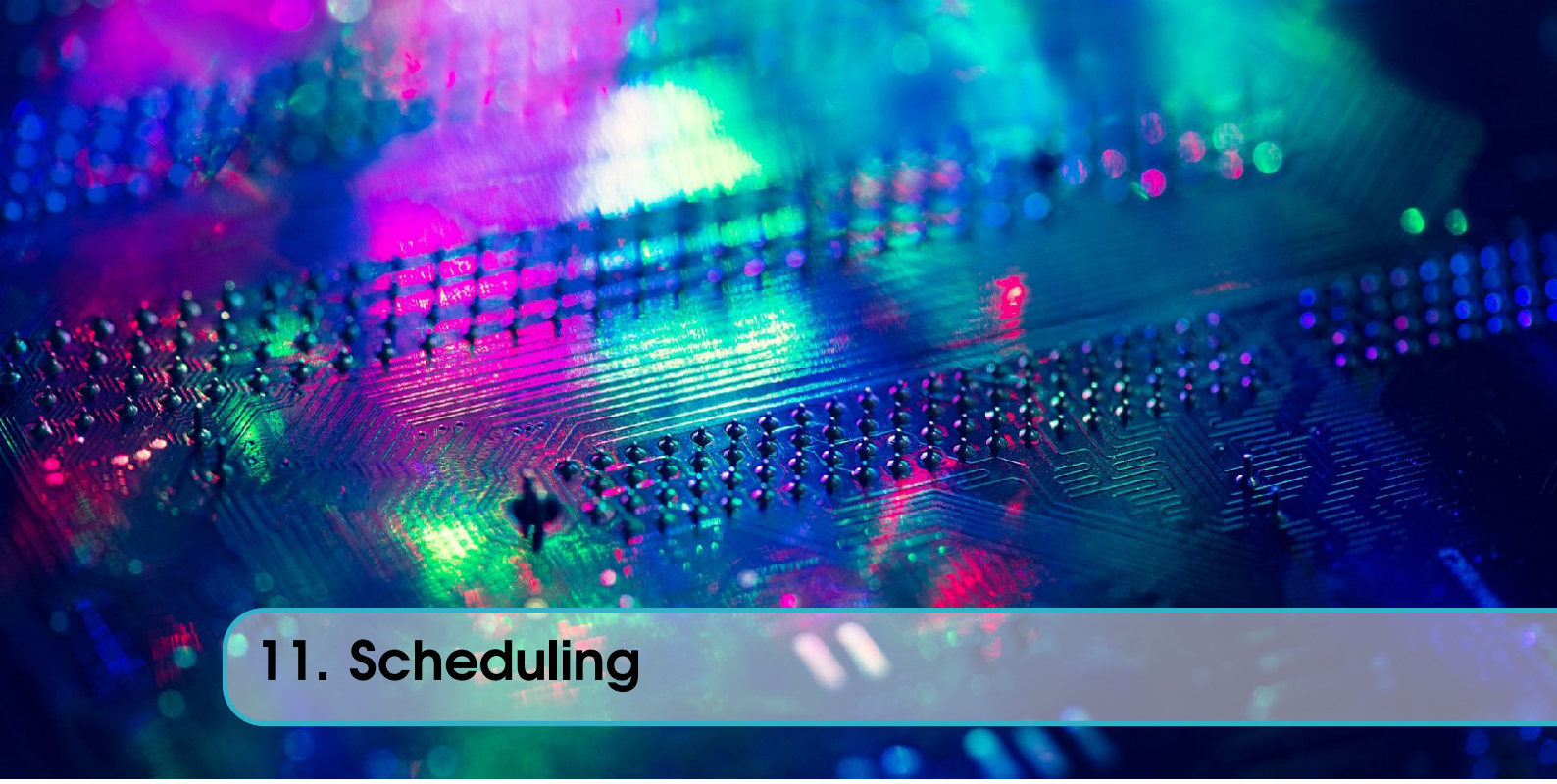


10.3 Conclusions on Optimization

In conclusion, addressing Constraint Optimization Problems (COPs) involves solving a sequence of Constrained Satisfaction Problems (CSPs). Two primary approaches are commonly employed:

1. Search over $D(f)$:
   - Destructive bounding and binary search.
   - Presents different trade-offs in terms of efficiency and computational effort.
2. Branch and Bound:
   - Pros:
     - Avoids wastage of information and promotes enhanced propagation.
     - An anytime algorithm, providing solutions at any point.
   - Cons:
     - Limited provision of lower bounds, compromising completeness.

IV Constraint-Based Scheduling



11. Scheduling

Scheduling involves the orderly arrangement of tasks that require resources over a specified time frame. This problem class is of utmost importance, often posing significant challenges, and finds applications in various practical scenarios.

Some Scheduling Problems are:

1. Project Planning and Scheduling: e.g software project planning.
2. Machine Scheduling: e.g. allocation of jobs to computational resources.
3. Scheduling of Flexible Assembly Systems: e.g. car production.
4. Employee Scheduling: e.g. nurse rostering.
5. Transport Scheduling: e.g. gate assignment for flights.
6. Sports Scheduling: e.g. schedule for NHL, World Cup, Olympics.
7. Educational Timetabling: e.g. Timetables at schools.

Resource Constrained Project Scheduling Problem (RCPSP)

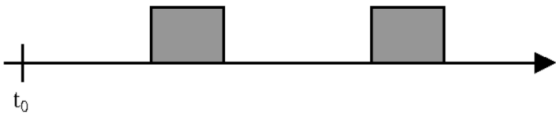
The RCPSP involves determining the optimal timing for executing a set of tasks while considering fixed-capacity resources, task durations, resource requirements, temporal constraints between tasks, and a specified performance metric.

A Constraint-Based Model for RCPSP includes:

- **Tasks**: Represented as variables.
- **Resource Constraints**: Modeled using unary, disjunctive, sequential, cumulative, or parallel resource constraints.
- **Temporal Constraints**: Specify relationships between tasks.
- **Performance Metric**: Captured through a schedule-dependent cost function.

11.1 Activity Variables

These variables correspond to various operations that need to be executed, such as processing an order, executing a job, working a shift, or performing a loading operation. The crux of the matter lies in determining the optimal positions for these operations within the timeline of the schedule.



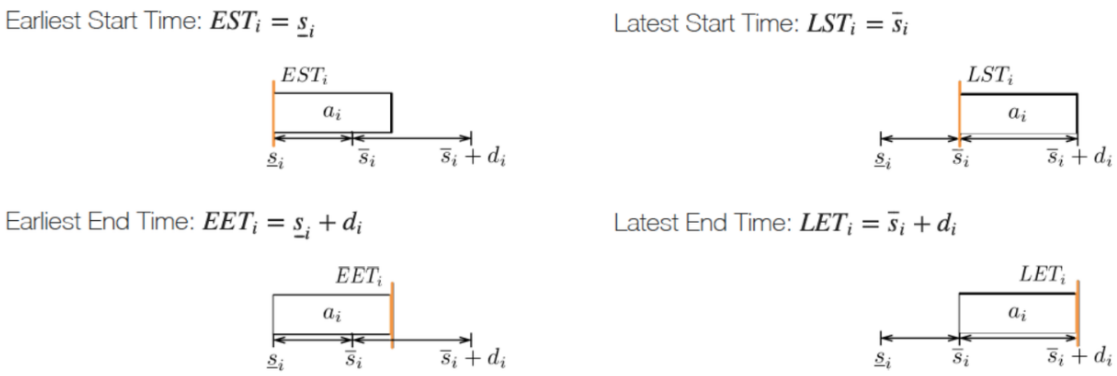
The principal variables in any scheduling problem revolve around the activities. For a given activity a_i, we have several associated variables:

- **Start Time** (S_i): This denotes the starting time variable for an activity a_i and is characterized by a domain $D(S_i)$. The earliest start time, or release date, is denoted as EST_i, while the latest start time is LST_i.
- **Duration** (d_i): The time it takes to complete the activity, typically assumed to be a known value.
- **End Time** (E_i): This represents the ending time variable for an activity a_i with a domain $D(E_i)$. The latest end time, or deadline, is LET_i, and the earliest end time is EET_i.

Activities can be categorized as preemptive or non-preemptive:

- **Preemptive Activity** a_i: Can be interrupted at any time, and its end time is determined by $S_i + d_i \leq E_i$.
- **Non-preemptive Activity** a_i: Cannot be interrupted and adheres to the relationship $S_i + d_i = E_i$.

Our focus predominantly lies on non-preemptive activities for the purpose of scheduling optimization.



11.2 Resources

Resources constitute the fundamental assets available to carry out various operations. These resources can take diverse forms:

- **Machine Capacity**: Represents the capability of a machine to execute operations.
- **Vehicle Volume**: Denotes the capacity of a truck, specifying the volume it can accommodate.
- **Classroom Seating**: Represents the number of seats available in a classroom.
- **Workforce Availability**: Signifies the number of available workers to perform tasks.

11.2.1 Cumulative/Parallel Resource

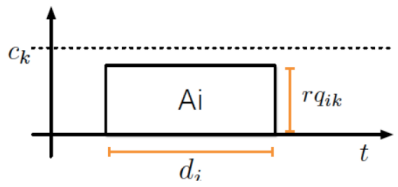
A cumulative or parallel resource possesses the capability to handle multiple activities concurrently. This means that activities can overlap in time, allowing for simultaneous execution. Examples include a group of identically skilled workers, a delivery truck, or a multi-core CPU.

For each resource r_k, there is an associated capacity c_k. Each activity a_i requires a certain amount $rq_{ik} \geq 0$ of the resource r_k during its execution. Throughout the schedule execution, the cumulative usage of r_k by all activities a_i should not exceed the capacity c_k.

Cumulative constraints, denoted by `cumulative`, ensure that the total resource usage during the execution of the schedule adheres to the given capacity constraints:

$$\texttt{cumulative}([S_1, S_2, \ldots, S_n], [d_1, d_2, \ldots, d_n], [rq_{1k}, rq_{2k}, \ldots, rq_{nk}], c_k)$$
$$\text{iff} \sum_{i|S_i \leq u < S_i + d_i} rq_{ik} \leq c_k, \ \forall u \in D$$

In this equation, S_i represents the start time of activity a_i and D signifies the domain of time.



In Resource-Constrained Project Scheduling Problems (RCPSP), resources predominantly adhere to cumulative constraints.

11.2.2 Unary/Disjunctive/Sequential Resource

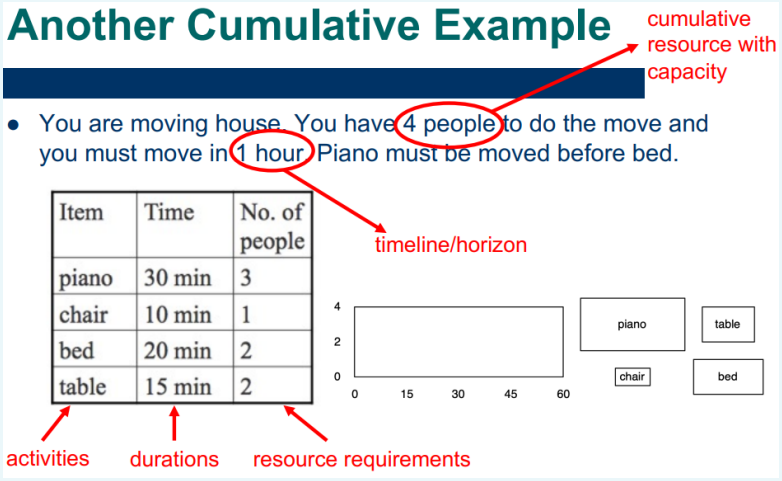
In contrast, a unary, disjunctive, or sequential resource can execute only one activity at a time. This implies that activities cannot overlap independently of the resource capacity. Examples include a classroom, a segment of train track, or a construction site crane.

Activities governed by a disjunctive constraint, denoted by `disjunctive`, ensure that no two activities overlap in time:

$$\texttt{disjunctive}([S_1, \ldots, S_n], [D_1, \ldots, D_n])$$
$$\text{iff } (S_i + D_i \leq S_j) \vee (S_j + D_j \leq S_i) \text{ for all } i < j$$

This constraint ensures that either the completion time of activity a_i precedes the start time of a_j or vice versa, preventing temporal overlap.

■ **Example 11.1** Cumulative Example Consider the scenario depicted in the scheduling diagram:



The associated scheduling problem can be formally defined as follows:

Define the domains of time for activities: $D(P) = D(C) = D(B) = D(T) = [0..60]$.

- Specify the temporal constraints for each activity: $P+30 \leq 60$, $C+10 \leq 60$, $B+15 \leq 60$, $T + 15 \leq 60$.
- Introduce precedence constraint: $P + 30 \leq B$.

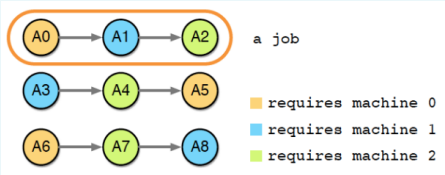
To capture the cumulative nature of the resources, we utilize the cumulative constraint:

$$\texttt{cumulative}([P, C, B, T], [30, 10, 20, 15], [3, 1, 2, 2], 4)$$

In this expression, the cumulative constraint ensures that the sum of resource requirements for each time point, considering activities P, C, B, and T, does not exceed the resource capacity of 4. The specific resource requirements for each activity are denoted by the arrays $[30, 10, 20, 15]$ and $[3, 1, 2, 2]$, respectively, representing the duration and resource consumption for each activity. ■

■ **Example 11.2 — Disjunctive Example.** Let's delve into the Job Shop Scheduling Problem, where the objective could be, for example, the scheduling a series of activities representing the manufacturing process of an automobile. In this scenario, each activity corresponds to a distinct machine, and there is a constraint that each machine can only handle one activity at a time.

Consider the scheduling diagram:



In this depiction, each horizontal line represents a job, and each circle represents an activity associated with a specific job. The disjunctive constraint ensures that the activities (machines) do not overlap in time, reflecting the restriction that each machine can handle only one activity at any given moment. ■

11.3 Temporal Constraints

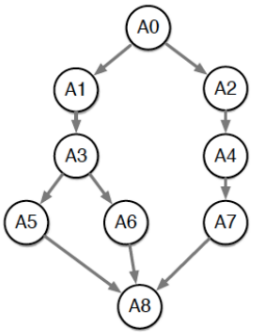
Temporal constraints play a pivotal role in shaping the dynamics of activities and their scheduling. Let's explore some fundamental temporal constraints that influence the sequencing of tasks.

11.3.1 Precedence Constraints

One of the essential temporal relationships is the precedence constraint, dictating the order in which activities unfold. Represented as $a_i \rightarrow a_j$, this constraint ensures that the completion of activity a_i precedes the commencement of activity a_j. In mathematical terms, it is expressed as $E_i \leq S_j$. This constraint finds applications in diverse scenarios, ranging from the meticulous task of house moving, where the piano must be relocated before setting up the bed, to complex problem-solving domains like the Resource-Constrained Project Scheduling Problem (RCPSP) and the Job Shop Scheduling Problem.

In the context of the RCPSP, the interplay of activities and precedence constraints gives rise to a

Directed Acyclic Graph (DAG) known as the Project Graph. Similarly, in the Job Shop Scheduling Problem, tasks within a job adhere to a sequential order, showcasing the ubiquity of precedence constraints in temporal management.

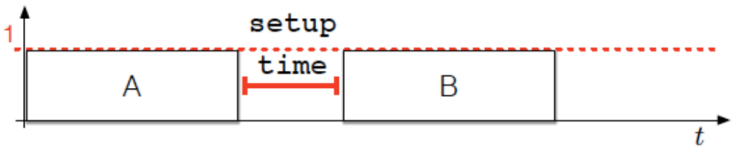


11.3.2 Time-legs & Time Windows

Time-legs and time windows introduce bounds on the temporal separation between the end time of one activity and the start time of another. Symbolically denoted as $a_i \xrightarrow{[l_{ij},u_{ij}]} a_j$, this constraint is mathematically expressed as $l_{ij} \leq S_j - E_i \leq u_{ij}$. Time windows, essentially time legs originating from a dummy activity a_0 with $S_0 = 0$ and $d_0 = 0$, further refine the temporal constraints to ensure activities occur within specified time intervals, adhering to $l_j \leq S_j \leq u_j$.

11.3.3 Sequence-dependent Set Up Times

Specifically relevant to unary resources, sequence-dependent set up times delineate constraints associated with the consecutive scheduling of activities. If a_i and a_j are arranged sequentially, a separation constraint governs their temporal relationship, expressed as $E_i \leq S \rightarrow E_i + d_{ij} \leq S_j$.



11.4 Cost Function

The determination of an optimal schedule hinges on the formulation of an appropriate cost function. Among all the cost functions, the **makespan** stands out as a prevalent metric, representing the completion time of the last activity within a schedule. Achieving an optimum makespan entails minimizing this metric.

For both the RCPSP and the job shop scheduling problem, the makespan serves as the primary cost function. Various models can be employed to represent the minimum makespan. One approach involves minimizing the maximum among the completion times of individual activities: $minimize \max([S_1 + d_1, \ldots, S_n + d_n])$. Alternatively, an auxiliary strategy introduces a dummy activity a_{n+1} with zero duration and establishes constraints to ensure its precedence over all other activities, transforming the objective to minimizing S_{n+1}.

Beyond makespan, other cost functions cater to specific optimization criteria. These include:

- **(Weighted) Tardiness Costs**: $\sum_{a_i \in A} w_i \cdot \max(0, E_i - LET_i)$
- **(Weighted) Earliness Costs**: $\sum_{a_i \in A} w_i \cdot \max(0, LET_i - E_i)$
- **Peak Resource Utilization**: Evaluating the highest resource utilization during the schedule.
- **Sum of Set Up Times and Costs**: Aggregating the set up times and associated costs.

11.5 Search Heuristics

In the quest to navigate the expansive solution space of scheduling problems, characterized by substantial variable domains such as the start times (S_i) and end times (E_i), the choice of effective search heuristics becomes paramount. Two fundamental questions arise in this context: which variable to select next, and which value to assign to the chosen variable?

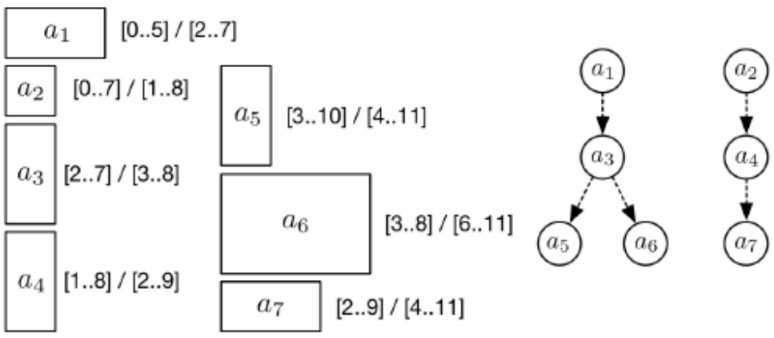
For the RCPSP, where the primary objective is to minimize the makespan, a judicious strategy is to focus on the earliest start time (EST_i). The rationale behind this lies in the fact that increasing the start time of a particular activity (S_i) without affecting the start times of other activities cannot lead to an improvement in the makespan. This principle holds not only for the RCPSP but extends to various scheduling problems characterized by **regular cost metrics**. Regular costs imply that modifying a single start time (S_i) in isolation cannot enhance the overall cost.

When faced with the decision of which variable to prioritize during the search, a criterion is to opt for the variable associated with the minimum earliest start time (EST_i). In instances where ties need resolution, the tie-breaking strategy involves selecting the variable with the tightest deadline, denoted by the minimum latest end time (LET_i). This approach ensures a systematic exploration of the solution space, with a focus on variables that contribute most significantly to the optimization objective.

11.5.1 Priority Rule-Based Scheduling

In the realm of scheduling problems, a straightforward yet effective solution approach is the Priority Rule-Based Scheduling, a form of greedy strategy that, while not guaranteeing optimality, often yields satisfactory solutions. Let's delve into this approach through an illustrative example.

Consider the scenario depicted in the following figure, assuming we have propagated the precedence constraints and obtained the start time and end time domains for each activity:

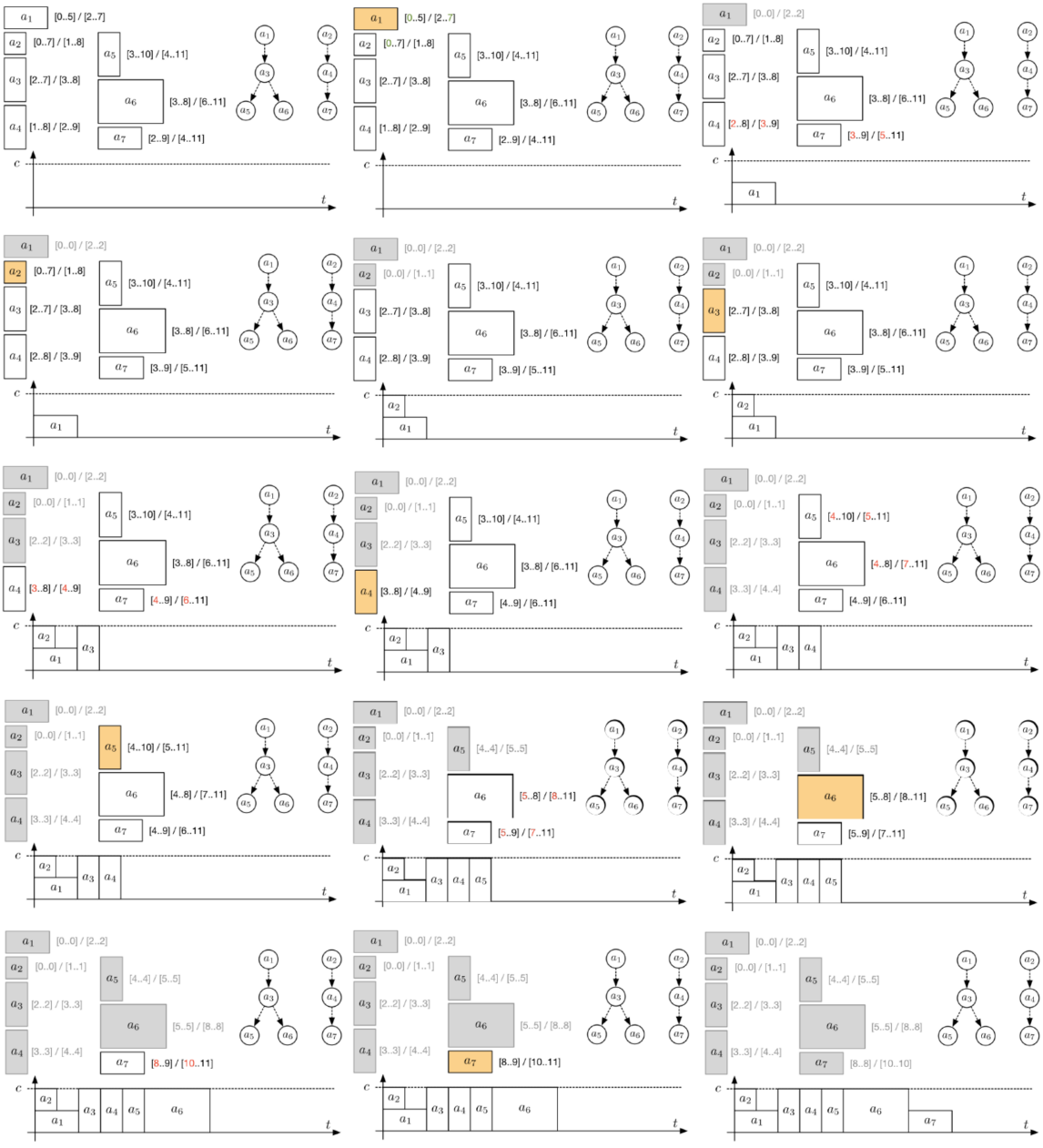


Here, the notation $[EST_i..LST_i]/[EET_i..LET_i]$ represents the earliest start time to latest start time and earliest end time to latest end time intervals for activity i.

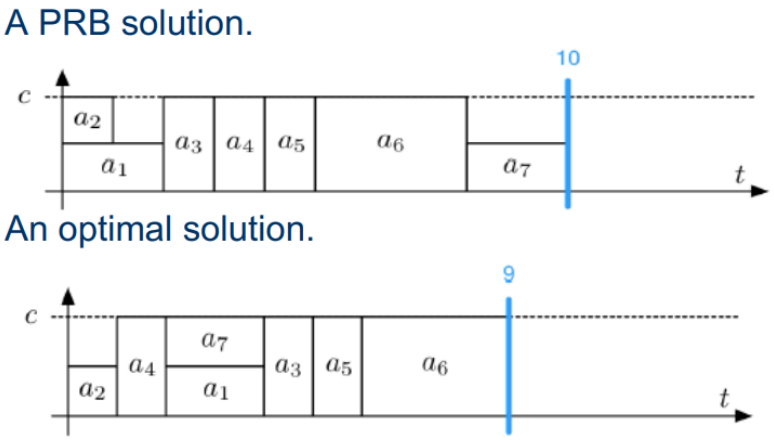
When determining the order in which to schedule activities, a sensible criterion is to prioritize the activity with the minimum earliest start time (EST_i). In cases where ties exist, the tie-breaking

strategy involves selecting the activity with the tightest deadline, represented by the minimum latest end time (LET_i).

With this heuristic, the problem is solved as follows:



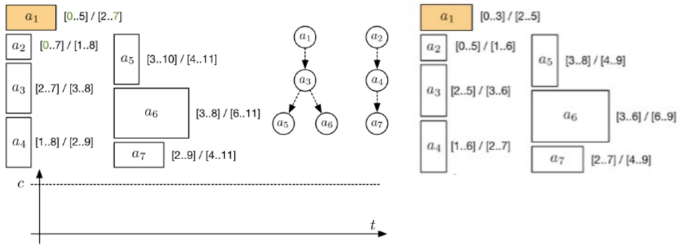
This method is efficient in finding a solution, but does not guarantee optimality:



Backtracking for Proving Optimality

To prove optimality, the backtracking mechanism plays a crucial role. Let's navigate through the process with a practical example.

Given the schedule depicted in the figure:



We encounter a decision point: Is $S_1 \neq 0$? However, this condition is weak due to the tendency of S_i domains to be excessively large.

11.5.2 SetTimes Search Strategy

The SetTimes approach offers a distinctive alternative marked by its focus on the notion of postponing activities. This approach introduces the concept of marking an activity, say i, as "postponed," implying that i cannot be selected for branching until its earliest start time (EST_i) undergoes a change. The underlying rationale is to explore different branching decisions, ensuring that activities are scheduled precisely at their earliest start times. The key insight is that the scheduling decision changes only when EST_i changes.

The primary steps of the SetTimes strategy can be outlined as follows:

1. On the first branch, schedule an activity a_i with the minimum EST_i, placing it at its EST_i.
2. Break ties according to any predefined rule.
3. On backtracking, mark a_i as postponed.
4. Upon propagation updating EST_i, schedule a_i again.

This strategy, rooted in Priority Rule-Based (PRB) scheduling, proves to be highly effective, particularly in finding good solutions early in the search process. It excels in making efficient branching choices, significantly outperforming alternatives like posting $S_i \neq v$. However, it's crucial to note that SetTimes is an incomplete search strategy. At choice points, it does not partition the search space; rather, it revolves around the decision of scheduling an activity at EST_i or making

it wait.

Why does it work?

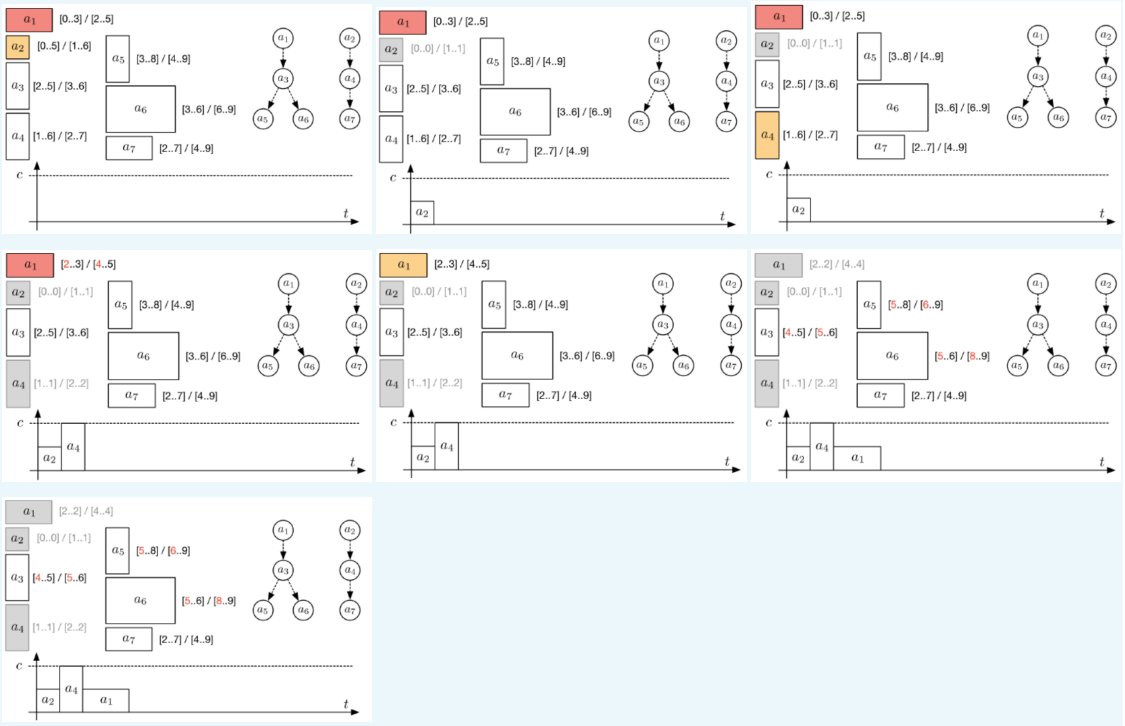
- The cost function is regular.
- There is no point in not scheduling activities at their EST_i unless they are delayed by previous activities.

When doesn't it work?

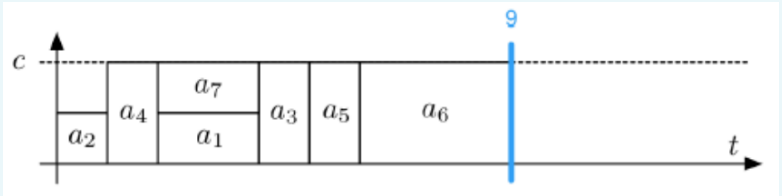
- Non-regular cost functions.
  - E.g., costs for starting activities too early.
- Side constraints that alter the problem structure.
  - E.g., maximal time legs.

Other strategies are becoming more popular. E.g., domain splitting.

■ **Example 11.3** Backtracking from the previous example

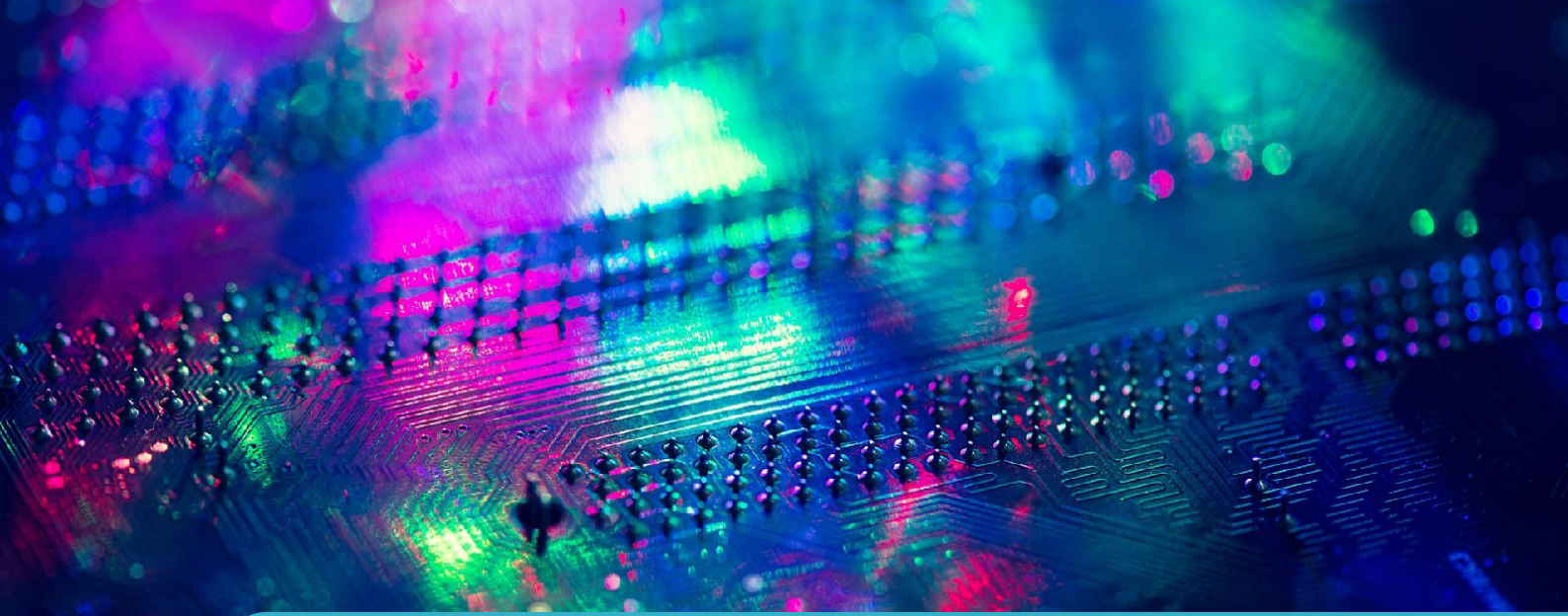


By proceeding along this branch, we will find the optimal solution.



■

V Heuristic Search



12. Complete and approximate Methods

In constraint programming, two overarching categories of methods emerge: complete methods and approximate methods. Each category serves distinct purposes, presenting unique characteristics and trade-offs.

12.1 Complete Methods

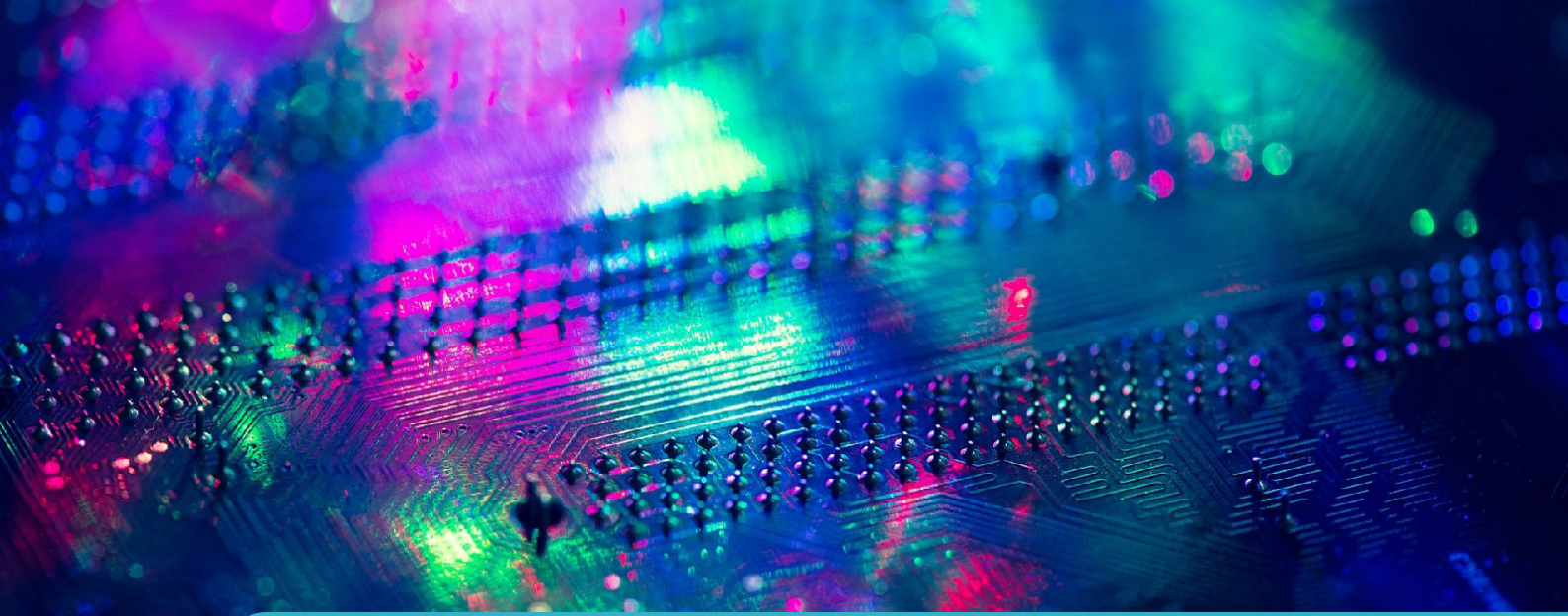
Complete methods are characterized by their guarantee to find a solution, including an optimal one, within a bounded time frame for every finite-sized instance. However, it's important to note that these methods might demand exponential computation time, particularly for larger or more intricate instances. Examples of complete methods span various domains, such as constructive tree search in Constraint Programming (CP), branch & bound, branch & cut in Integer Linear Programming (ILP), and several other tree search techniques.

12.2 Approximate Methods

Approximate methods, on the other hand, do not provide a guarantee of optimality, nor do they assure termination in cases of infeasibility. Despite this, they excel in delivering high-quality solutions within a considerably reduced time frame. These methods are particularly valuable when achieving an optimal solution is impractical or time-consuming.

Some types of approximate methods are:

- Constructive Heuristics
- Local Search
- Metaheuristics



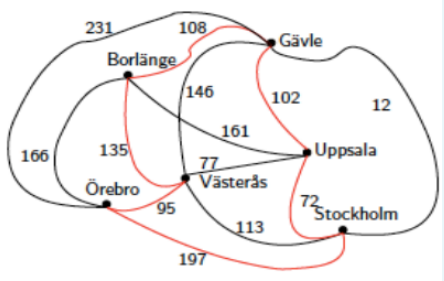
13. Constructive Heuristics

In the realm of approximate methods, constructive heuristics stand out as some of the fastest approaches to generating solutions. These methods operate by iteratively extending partial assignments until a solution is found or predetermined stopping criteria are met. Leveraging problem-specific knowledge, often in the form of heuristics, constructive heuristics are adept at building solutions from scratch.

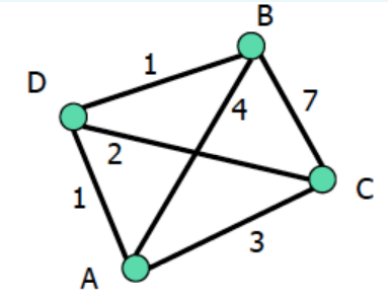
A prominent category within constructive heuristics is greedy heuristics, known for making locally optimal choices at each stage of the solution construction process. The Priority Rule-Based Scheduling method, introduced in the previous chapter, serves as an illustrative example of a greedy heuristic. These heuristics are characterized by their simplicity, speed, and the ability to yield reasonably good approximations. However, it's essential to note that solutions produced by constructive heuristics may deviate significantly from optimal solutions, as certain choices are committed to prematurely.

Constructive heuristics, despite their limitations, find widespread application when combined with other methods. They are frequently used for tasks such as initialization in local search algorithms and metaheuristics. By capitalizing on their efficiency and quick generation of solutions, constructive heuristics contribute to the broader landscape of approximate methods, demonstrating their utility in tackling real-world challenges.

■ **Example 13.1 — Travelling Salesman Problem (TSP).** The Travelling Salesman Problem (TSP) poses the question: Given a list of connected cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city? In graph-theoretic terms, this problem seeks the minimum cost (total distance) Hamiltonian tour in a graph where cities represent vertices and connections with distances are weighted edges.



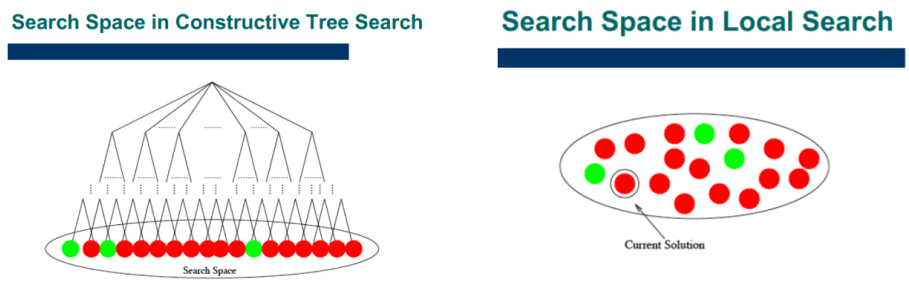
One specific greedy heuristic for the TSP involves visiting the next unvisited city nearest to the current city at each stage. This Nearest Neighbor approach, starting from city A, might yield a tour like A-D-B-C-A with a total distance of 12. However, it's important to recognize that such a solution is not guaranteed to be optimal. An alternative tour like A-C-D-B-A with a total distance of 10 demonstrates the heuristic's tendency to provide suboptimal solutions.



14. Local Search

In the realm of approximate methods, **local search** stands out for its ability to deliver solutions of superior quality compared to constructive heuristics. Unlike the latter, which builds solutions from scratch, local search starts from an **initial solution** and iteratively seeks to replace it with a better one within a defined **neighborhood**. This process involves applying **small**, **local** modifications to the current solution.

Local search is versatile and can commence with either a **feasible or unfeasible assignment** of all variables. The fundamental concept revolves around iteratively exploring the local space of solutions, seeking improvements through incremental modifications.



Given $\langle X, D, C, f \rangle$, Local search find a feasible solution $s^* \in S$ such that $f(s^*) \leq f(s)$ for all $s \in S$.

A critical component of local search is the definition of a **neighborhood structure**. This structure, denoted by $N : S \to 2^S$, assigns a set of neighbors $N(s) \subseteq S$ to every solution $s \in S$. The neighborhood is often implicitly defined by specifying modifications that must be applied to s to generate its neighbors $N(s)$. Each application of such an operator, producing a neighbor, is termed a move.

Definizione 14.0.1 — Local Minimum. A locally minimal solution, or local minimum, with respect to a neighborhood structure N, is a solution s' such that $f(s') \leq f(s)$ for all $s \in N(s')$.

14.0.1 A Simple Local Search Algorithm

One such Simple Local Search algorithm, illustrated in the pseudocode above, follows a straightforward approach:

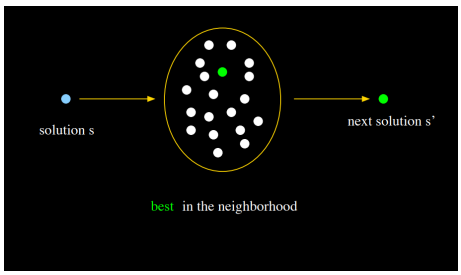
Algorithm 1 Iterative improvement local search

```
s ← GenerateInitialSolution()
while ∃ s' ∈ N(s) such that f(s') < f(s) do
    s ← ChooseImprovingNeighbor(N(s))
end while
```

1. **Initialization**: An initial solution is generated either randomly or through a heuristic method.
2. **Move Exploration**: A move is considered only if the resulting solution is better than the current solution, adhering to the principles of hill climbing. The choice of improving neighbors can follow strategies such as first improvement or best improvement.
3. **Stopping Criteria**: The algorithm terminates as soon as a local minimum is reached. The effectiveness of the algorithm heavily relies on the structure of the neighborhood explored.

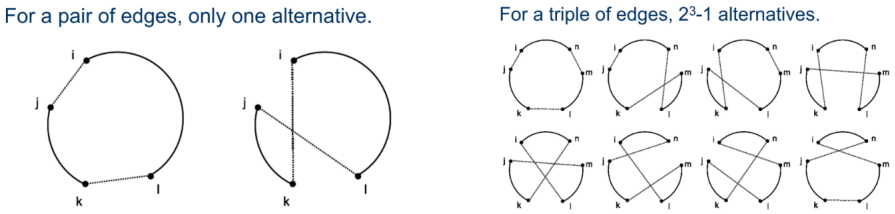
14.0.2 Iterative Improvement

The process of iterative improvement is depicted in the diagrams below, showcasing the evolution of solutions through successive iterations. For the Traveling Salesman Problem (TSP), an initial solution can be obtained using any Hamiltonian tour, often generated with a constructive heuristic like the nearest neighbor method.



14.0.3 K-exchange Neighbourhood

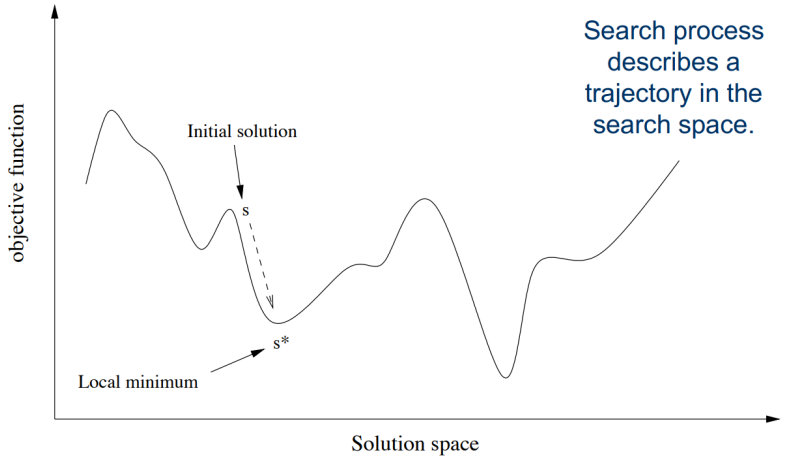
The K-exchange neighborhood, exemplified by 2-exchange and 3-exchange scenarios, offers alternatives for modifying edges in the tour. For a pair of edges, there is only one alternative in a 2-exchange, whereas a 3-exchange introduces $2^3 - 1$ alternatives. These alternative configurations are explored to find an improved solution.

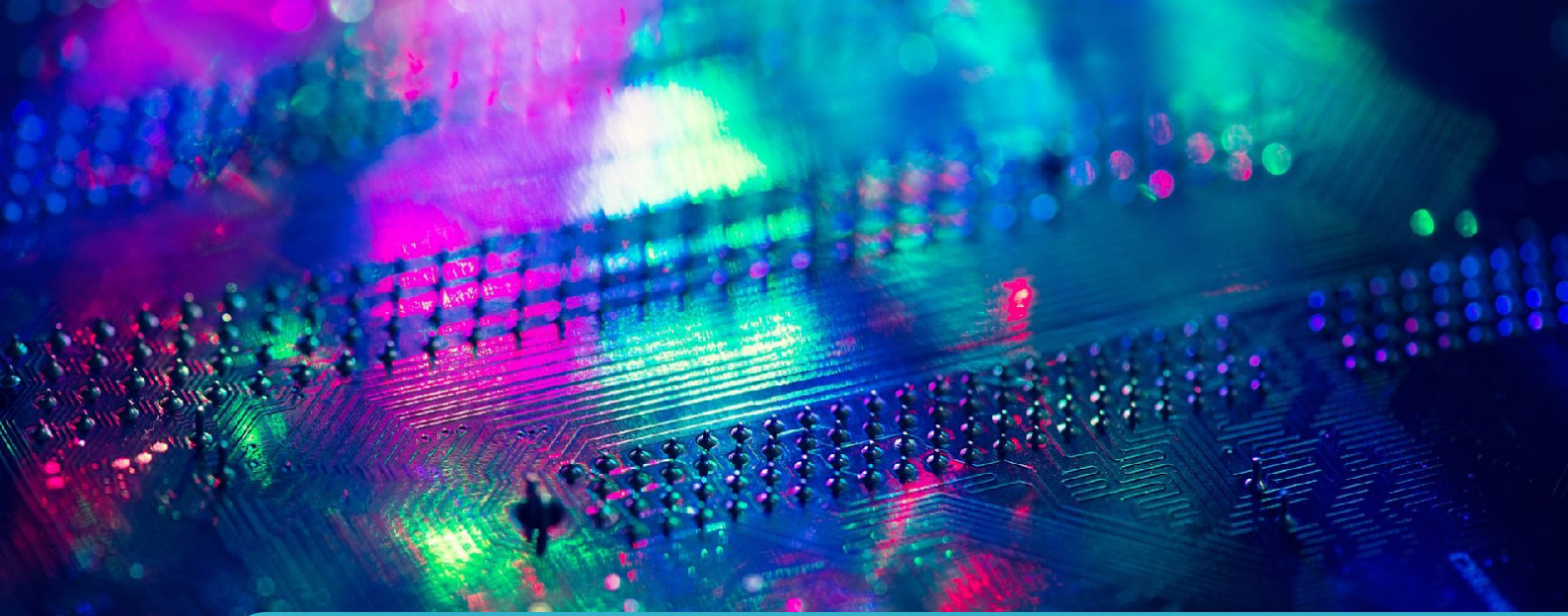


14.0.4 An Iterative Improvement Algorithm for TSP

1. **Initial Tour**: Build an initial tour using the nearest neighborhood heuristic.

2. **Random Edge Selection**: Randomly select an edge from the tour.
3. **2-exchange Move**: Perform a 2-exchange move with all other edges of the tour, selecting the best tour generated in the process.
4. **Update Solution**: If the new tour is better than the current tour, make it the current tour, and return to step 2.
5. **Termination**: If the new tour is not an improvement, the algorithm stops as the local minimum is reached.





15. Metaheuristics

Metaheuristics are high-level strategies designed to enhance the performance of optimization algorithms. These strategies leverage various techniques, including a priori knowledge in the form of heuristics, adaptation based on search history, and general approaches to balance intensification and diversification within the search process. The primary objective is to avoid getting trapped in local minima and instead explore and discover better local minima.

Two essential driving forces behind metaheuristic search are intensification and diversification:

Definizione 15.0.1 — Intensification. This involves exploiting accumulated search experience, often by focusing the search within a confined and small area of the search space. It aims to refine the current solution by exploiting the local structure.

Definizione 15.0.2 — Diversification. This focuses on exploration "in the large" of the search space. It aims to discover new regions and possibilities within the search space. Diversification is crucial for preventing premature convergence to suboptimal solutions.

Balancing intensification and diversification is critical, as they are contrary yet complementary forces. The challenge is to quickly identify high-quality solution regions without wasting too much time in explored or less promising areas.

Metaheuristics encompass and combine various optimization methods, including:

1. **Constructive Methods**: These involve the creation of solutions through random, heuristic, adaptive, or other strategies. Constructive methods lay the foundation for the search process.
2. **Local Search-Based Methods** (Trajectory): These methods refine solutions iteratively by exploring the local neighborhood. They are focused on improving the current solution by making small adjustments.
3. **Population-Based Methods**: These methods maintain a set of candidate solutions (population) and evolve them over time. They introduce diversity through the interaction of multiple solutions and their exploration of different regions in the search space.

15.1 LS-based Methods

Local Search-based Methods constitute a category of optimization algorithms that share similarities with Local Search (LS). In these methods, a single solution is utilized at each iteration of the algorithm, and the search process is characterized by a trajectory within the search space.

In line with traditional LS approaches, the algorithm maintains a trajectory that explores the search space iteratively. However, there are distinctive features that set Local Search-based Methods apart.

Unlike conventional LS methods, Local Search-based Methods introduce a **diversification** component to the iterative improvement process. This component serves the purpose of escaping local minima, allowing for moves that might worsen the objective function. The diversification component incorporates strategies such as:

- Allowing worsening moves.
- Changing the neighborhood structure during the search.
- Modifying the objective function during the search.

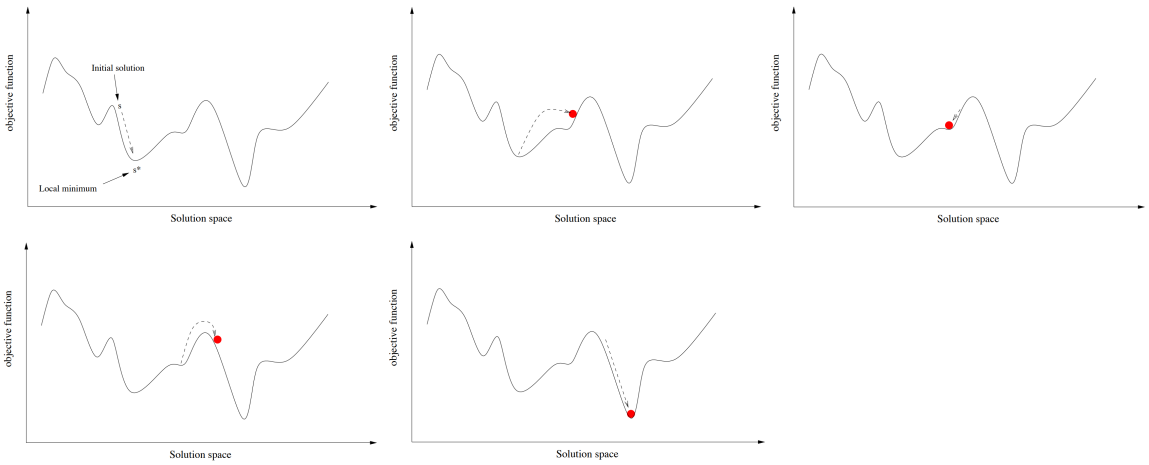
To control the algorithm's execution, **termination criteria** are established. These criteria may include maximum CPU time, a predefined number of iterations without improvement, achieving a solution of sufficient quality, and others.

Some notable Local Search-based Methods are:

- Simulated Annealing (SA)
- Variable Neighbourhood Search (VNS)
- Tabu Search (TS)
- Guided Local Search (GLS)
- Iterated Local Search (ILS)
- Greedy Randomized Adaptive Search Procedure (GRASP).

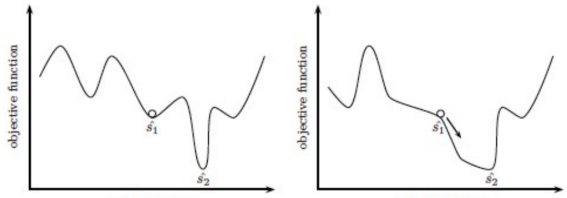
15.1.1 Simulated Annealing (SA)

Simulated Annealing is distinguished by its **acceptance of worsening moves during the search process**. The algorithm intuitively mimics the idea of climbing a hill and exploring downward in a different direction within the same search landscape. The probability of accepting worsening moves decreases as the search progresses, emphasizing a shift from diversification to intensification.



15.1.2 Variable Neighbourhood Search (VNS)

Variable Neighbourhood Search introduces the concept of **changing the neighborhood structure during the search**. The intuition behind this method is that different neighborhoods generate distinct search landscapes. As soon as a local minimum is reached, a neighborhood is substituted, enabling exploration of alternative search spaces.



(a) Search landscape 1 (b) Search landscape 2

15.1.3 Tabu Search

Tabu Search is a metaheuristic that **dynamically adapts its search strategy by changing the neighborhood structure based on the exploration history**. A key element of Tabu Search is the use of a Tabu list, which keeps track of recently visited solutions or moves and restricts revisiting them.

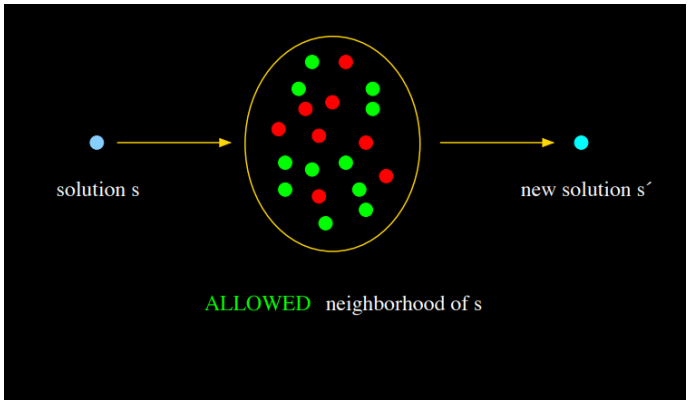
In the Tabu Search process, the algorithm maintains a Tabu list to forbid recently explored solutions or moves. The primary goal is to prevent the algorithm from revisiting the same areas of the search space too frequently. This helps to strike a balance between exploration and exploitation.

Storing entire solutions in the Tabu list might be inefficient, leading to a compromise. Tabu Search often opts to store individual moves instead of complete solutions. While this might eliminate some potentially good, unvisited solutions, it enhances the efficiency of the search process.

Aspiration criteria allow the algorithm to make an exception to the Tabu status and accept a forbidden move under certain circumstances. Specifically, a forbidden move can be accepted if it leads to a solution better than the current one.

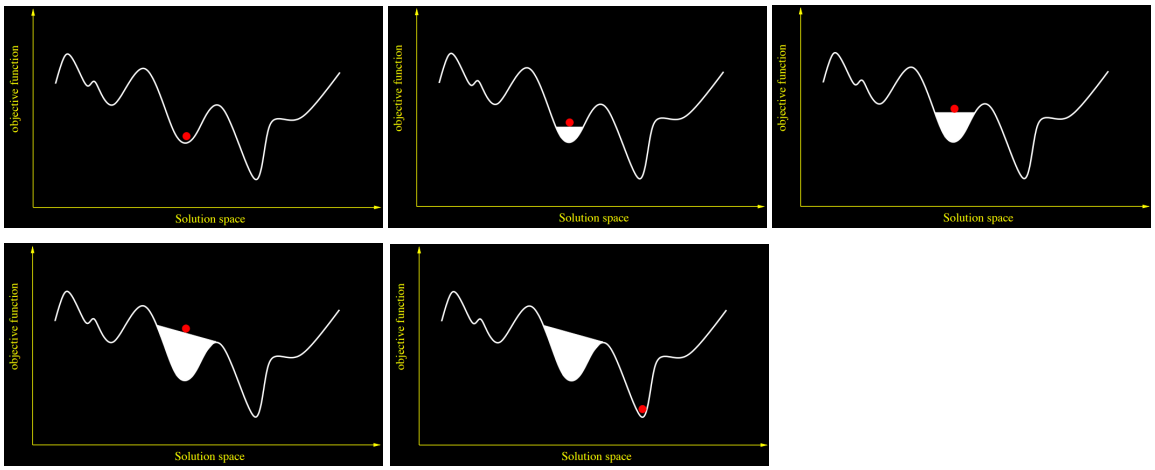
The Tabu list size is a crucial parameter in the exploration-exploitation balance. It determines the scope of exploration and influences the algorithm's preference for diversification over intensification. The dynamic adjustment of the Tabu list size is of particular interest:

- Increase the Tabu size in case of repetitions, indicating the need for **diversification**.
- Decrease the Tabu size in case of no improvements, boosting **intensification** efforts.



15.1.4 Guided Local Search (GLS)

Guided Local Search introduces the idea of changing the objective function during the search to navigate around local minima. The objective is to modify the search landscape and make the current local optimum less desirable. GLS penalizes solution features that frequently occur in visited solutions, thereby encouraging exploration of alternative regions.

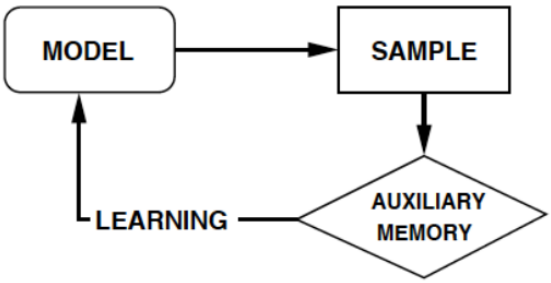


15.2 Population-based Methods

Population-based metaheuristics involve the use of a set, or population, of solutions at each iteration of the algorithm. In these approaches, the search process is characterized by the evolution of a set of points or a probability distribution within the search space. Many of these methods draw inspiration from natural processes, such as natural evolution and the foraging behavior of social insects.

The fundamental principle underlying population-based metaheuristics is to learn correlations between components of good solutions. Here's a general overview of their operation:

1. Probabilistic Model Generation:
   - Candidate solutions are generated using a parametrized probabilistic model.
   - This model defines the probability distribution for generating new solutions.
2. Model Update:
   - The model is updated based on the solutions encountered during the search process.
   - This updating mechanism aims to concentrate the search in regions of the search space that contain high-quality solutions.

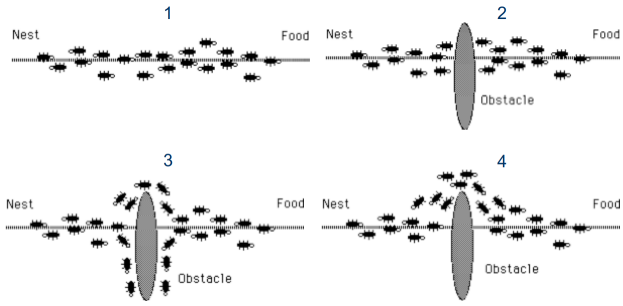


Examples of population-based metaheuristics include **Evolutionary Computation** (EC) and **Ant Colony Optimization** (ACO). These approaches leverage the collective behavior of a population

of solutions to explore and exploit the search space effectively.

15.2.1 Ant Colony Optimization (ACO)

Ant Colony Optimization (ACO) draws inspiration from the foraging behavior of ants, particularly their ability to find the shortest path between the nest and a food source. As ants move, they deposit a substance known as pheromone on the ground. When deciding on a direction, ants tend to choose paths with stronger pheromone concentrations. This cooperative interaction among ants leads to the emergence of shortest paths.



The simulation of pheromone trails in ACO involves a parametrized probabilistic model, the pheromone model. This model consists of parameters representing pheromone values, which act as memory to track the search process. The pheromone values intensify the search around the best solution components. For instance, a pheromone value $\tau(X_i, v_i)$ for variables $X_i \in X$ and values $v_i \in D(X_i)$ signifies the desirability of assigning v_i to X_i. By bounding pheromone values between τ_{min} and τ_{max}, a balance is achieved between intensification and diversification, with initial values set to τ_{max}.

Artificial ants in ACO utilize constructive heuristics to probabilistically construct solutions using pheromone values. They iteratively select a variable X_i based on the heuristic and a value $v_i \in D(X_i)$ using the probability formula:

$$p(x_i, v_i) = \frac{[\tau(x_i, v_i)]^\alpha \cdot [\eta(x_i, v_i)]^\beta}{\sum_{v_j \in D(x_i)} [\tau(x_i, v_j)]^\alpha \cdot [\eta(x_i, v_j)]^\beta}$$

Here, $[\eta(x_i, v_i)]^\beta$ represents the heuristic factor, and parameters α and β balance the influence of pheromone and heuristic factors.

The updating of pheromone values involves two steps:

1. **Evaporation**: All pheromone values decrease by an evaporation factor, allowing ants to progressively forget older solutions and emphasize more recent ones (diversification).
2. **Reinforcement**: Pheromone values associated with assignments in good solutions increase proportionally to the solution's quality. This aims to boost the probability of selecting such assignments in future constructions (intensification).

ACO Algorithm

1. Initialize pheromone values.
2. Ants construct solutions using heuristics and the pheromone model.
3. The constructed solutions update pheromone values to bias future sampling towards high-quality solutions.

Termination criteria may include maximum CPU time, maximum iterations without improvement, or achieving a solution of sufficient quality.

15.3 LS or Population-based Metaheuristics?

When deciding between Local Search (LS) and Population-based metaheuristics, it's essential to consider their strengths and suitability for specific scenarios.

LS-based methods are generally preferred under the following conditions:

1. neighbourhood structures create a correlated search graph;
2. inventing moves is easy;
3. computational cost of moves is low.

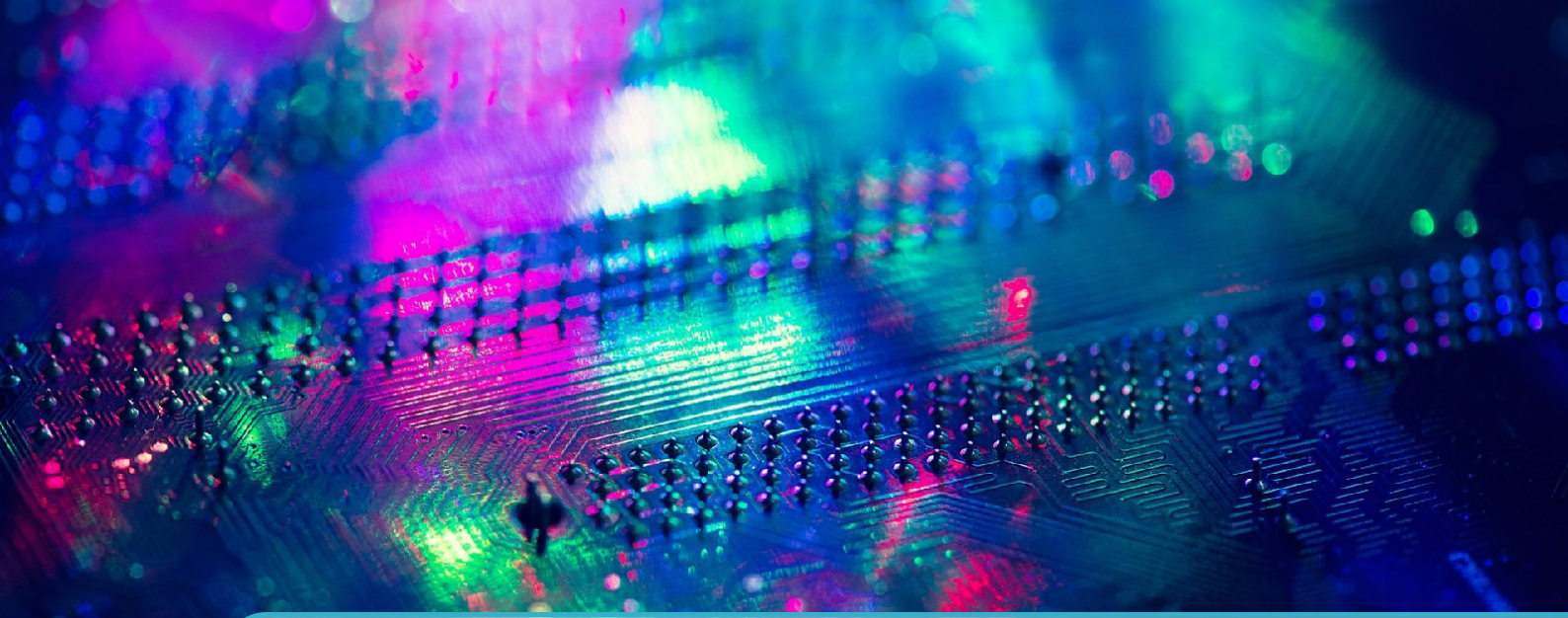
On the other hand, **Population-based methods** are more suitable in the following situations:

1. solutions can be encoded as composition of good building blocks;
2. computational cost of moves in LS is high;
3. it is difficult to design effective neighbourhood structures;
4. coarse grained exploration (e.g., huge search spaces) is preferable.

In essence, the strengths of these two approaches are complementary:

- **LS-based methods** focus on searching a promising area in the search space in a more structured manner, minimizing the risk of being close to good solutions but missing them. They exhibit intensification ability.
- **Population-based methods** generate new solutions by recombining earlier solutions, performing guided sampling of the search space. This often results in a coarse-grained exploration, providing diversification ability.

Hybrid methods have been developed to exploit the complementary strengths of LS-based and population-based methods.



16. Hybrid Metaheuristics

The primary objective of hybrid metaheuristics is to exploit the complementary strengths of LS-based and population-based methods, aiming for synergy in optimization processes. One approach is the incorporation of LS-based methods within population-based methods. For example, local search can be applied to solutions generated by Ant Colony Optimization (ACO). This hybridization leverages the strengths of both methods, combining the efficiency of population-based exploration with the focused refinement of LS-based local search.

Population-based iterated local search and multilevel techniques are other hybridization strategies employed in this context.

In the realm of Combinatorial Optimization, two main categories of methods exist:

1. Complete Methods:
   - Guarantee to find an optimal solution for every finite-sized instance within a bounded time.
   - Examples include constructive tree search in Constraint Programming (CP), branch & bound, branch & cut in Integer Linear Programming (ILP), and other tree search methods.
   - May require exponential computation time.
2. Approximate Methods:
   - Cannot guarantee optimality or termination in infeasible scenarios.
   - Focus on obtaining good-quality solutions quickly, even if not optimal.
   - Well-suited for handling large instances efficiently.

Considering their complementary strengths:

- **CP (Complete Method)**:
  - Focuses on constraints and feasibility.
  - Provides easy modeling and control of the search process.
  - Tends to be less effective in optimization tasks with loose bounds on the objective function.
- **Metaheuristics (Approximate Method)**:
  - Effective in rapidly finding good-quality solutions.
  - Handles constraints inefficiently, often resorting to penalizing infeasible assignments in the objective function.

16.1 Metaheuristics + Complete Methods

When combining metaheuristics with complete methods, several integration approaches are employed to leverage their respective strengths effectively:

1. Metaheuristics before complete methods:
   - Metaheuristics provide an initial solution or guide the search space exploration.
   - The output is then refined or optimized using a complete method.
2. Complete method applies metaheuristic:
   - A complete method utilizes a metaheuristic to enhance or optimize its solution.
   - This integration aims to exploit the efficiency of metaheuristics in refining solutions obtained through complete methods.
3. Metaheuristics using complete methods for exploration:
   - Metaheuristics incorporate a complete method to efficiently explore the solution space.
   - Examples include strategies like Large Neighbourhood Search (LNS) and the combination of Ant Colony Optimization (ACO) with Constraint Programming (CP).
4. Metaheuristic concepts for tree exploration:
   - Metaheuristic concepts are adapted to devise incomplete but efficient tree exploration strategies.
   - This approach helps in enhancing the exploration efficiency of complete methods.

In the context of LS-based methods, two key issues are crucial for effective optimization:

1. Defining Neighbourhood Structure:
   - The success of LS methods heavily depends on defining an appropriate neighbourhood structure.
   - A well-designed neighbourhood structure facilitates effective exploration of the solution space.
2. Neighbourhood Examination:
   - Choosing an efficient method to examine the neighbourhood of a solution is essential.
   - The examination process determines the quality of the local search and impacts the overall optimization performance.

The choice between small and large neighbourhoods in LS-based methods involves trade-offs:

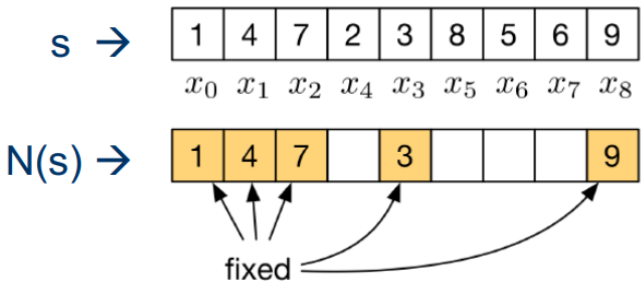
- Small Neighbourhoods:
  - Pros: Fast identification of improving neighbours (if any).
  - Cons: Local minima found in small neighbourhoods tend to have lower average quality.
- Large Neighbourhoods:
  - Pros: Local minima discovered in large neighbourhoods generally exhibit higher average quality.
  - Cons: Finding an improving neighbour might be more challenging due to the expansive search space.

The decision between small and large neighbourhoods depends on the specific characteristics of the optimization problem and the trade-off between exploration speed and solution quality.

16.2 Large Neighbourhood Search(LNS)

Large Neighbourhood Search (LNS) is a powerful optimization technique that combines the advantages of both Local Search (LS) and Constraint Programming (CP). The core idea behind LNS is to employ a generic and extensive neighbourhood and explore it efficiently using a complete method such as CP.

The fundamental concept involves treating the exploration of a neighbourhood as the solution to a sub-problem. This sub-problem is tackled through tree search, allowing for an exhaustive yet rapid exploration. Given an initial solution s, a part of the variables is fixed to the values they possess in s, creating what is referred to as a "fragment". The remaining variables are then relaxed for further exploration.



LNS brings several advantages over standalone LS and CP methods. Firstly, it ensures efficient neighbourhood exploration, leveraging the propagation and advanced search techniques of CP. Additionally, LNS is more straightforward to develop compared to LS, as it allows for an easy and problem-independent neighbourhood definition. There's no need to address the complexities of satisfying intricate constraints.

One of the key strengths of LNS is its scalability. It outperforms using only CP on the problem by breaking it down into smaller, more manageable subproblems. This scalability is achieved by controlling the size of subproblems, reducing domain sizes through fixed variables, and optimizing propagation efficiency in smaller domains.

When implementing LNS, several design decisions come into play:

1. Complete vs Incomplete Neighbourhood Exploration:
   - LNS often involves incomplete exploration, providing a balance between efficiency and solution quality.
2. How many variables to fix:
   - The size of the neighbourhood should be large enough for diversified search but with manageable complexity.
   - Tuning may be done manually for specific applications or through automatic/adaptive techniques.
3. Which variables to fix:
   - Variables can be fixed randomly for diversification.
   - Problem-specific approaches or automatic/adaptive techniques can be employed.

16.3 Ant Colony Optimization + CP

The combination of Ant Colony Optimization (ACO) and Constraint Programming (CP) brings together constructive techniques with complementary strengths, making it a powerful approach for tackling complex optimization problems.

ACO is known for its learning capability, particularly in finding paths in search spaces, while CP excels in efficiently handling constraints within a problem. When these two methodologies are combined, their synergy allows for more effective problem-solving.

In typical ACO + CP approaches, two main strategies are employed:

1. Use CP for Solution Construction in ACO
2. Use ACO for Variable and Value Ordering Heuristics in CP

16.3.1 CP followed by ACO

When implementing CP followed by ACO, the process unfolds in the following steps:

1. Artificial ants employ constructive heuristics for probabilistically constructing solutions using the pheromone values.
2. Iteratively choose a variable X_i according to the heuristic and a value $v_i \in D(X_i)$ according to the probability:

$$p(x_i, v_i) = \frac{[\tau(x_i, v_i)]^\alpha \cdot [\eta(x_i, v_i)]^\beta}{\sum_{v_j \in D(x_i)} [\tau(x_i, v_j)]^\alpha \cdot [\eta(x_i, v_j)]^\beta}$$

3. Pheromone values are updated as usual.

16.3.2 ACO followed by CP

When implementing ACO followed by CP, the process unfolds in the following steps:

1. ACO+CP is executed, and the final pheromone matrix is saved.
2. The resulting solution obtained from ACO+CP serves as an upper bound to the subsequent CP phase.
3. CP performs a complete search and uses the pheromone matrix as a heuristic information for value selection.