Constraint Programming (Solving)

Lecture Notes

Nadjib Lazaar
Université Paris-Saclay
lazaar@lisn.fr

Introduction

Constraint Programming (CP) is a declarative approach that allows modeling and solving complex combinatorial problems. By defining a set of variables, domains, and constraints, this approach facilitates the expression and automatic resolution of problems without explicitly specifying a particular algorithm.

1 Introduction to Constraint Programming (CP) Solving

CP is a declarative approach to solving combinatorial problems, in which a problem is modeled in terms of variables, domains (sets of possible values for each variable), and constraints (relations that must be satisfied between variables). One of the major advantages of this approach is that the solving algorithm does not need to be explicitly defined; instead, it relies on automated techniques that explore the space of possible solutions.

Solving a CP problem involves finding one or more values for each variable that satisfy all imposed constraints. This search is carried out using solvers that employ specialized algorithms to efficiently navigate the solution space and optimize the search for valid solutions.

2 Solving Methods

CP solving techniques vary depending on the nature of the problem and its constraints. The main methods include:

- **Backtracking:** This is one of the simplest yet effective methods for solving CP problems. The idea is to construct the solution incrementally,

verifying at each step whether the constraints are satisfied. If a constraint is violated, the algorithm backtracks to try another possibility.

- **Constraint Propagation:** This technique involves propagating the effects of constraints throughout the search space to reduce variable domains and proactively eliminate invalid solutions.
- **Local Search:** These techniques, such as descent algorithms or genetic algorithms, attempt to find solutions by exploring the neighborhood of a given solution, iterating locally, and seeking to improve the solution at each step.
- **Problem Decomposition:** This approach involves dividing a complex problem into simpler subproblems, allowing for faster and more targeted resolution.

3 Solving Methods

CP solving techniques vary depending on the nature of the problem and its constraints. The main methods include:

- **Backtracking:** This is one of the simplest yet effective methods for solving CP problems. The idea is to construct the solution incrementally, verifying at each step whether the constraints are satisfied. If a constraint is violated, the algorithm backtracks to try another possibility.
- **Constraint Propagation:** This technique involves propagating the effects of constraints throughout the search space to reduce variable domains and proactively eliminate invalid solutions.
- **Local Search:** These techniques, such as descent algorithms or genetic algorithms, attempt to find solutions by exploring the neighborhood of a given solution, iterating locally, and seeking to improve the solution at each step.
- **Problem Decomposition:** This approach involves dividing a complex problem into simpler subproblems, allowing for faster and more targeted resolution.

4 Solving with Backtracking

The backtracking algorithm is a search method used to solve combinatorial problems. It explores a solution space exhaustively and backtracks when a partial solution can no longer lead to a valid solution.

The fundamental idea is to construct a solution progressively, step by step, verifying at each stage whether the partial solution is valid according to the problem's constraints. If a step leads to a situation where the constraints are

no longer satisfied, the algorithm backtracks and tries another possibility. The process continues until a valid solution is found or all possibilities have been explored.

Algorithm 1: Backtracking (BT)

```
Input: ⟨X, D, C⟩ (constraint network), I (partial instantiation)
Output: true if a solution is found, otherwise false
if I is a complete instantiation then
    return true ;                          // A solution has been found
end
Select a variable X_i ∉ I;
foreach v ∈ D(X_i) do
    if I ∪ {X_i ← v} is locally consistent then
        if BT(⟨X, D, C⟩, I ∪ {X_i ← v}) then
            return true
        end
    end
end
return false ;                             // No solution was found
```

4.1 Algorithm Description

1. **Input:** The algorithm takes as input a constraint network $\langle X, D, C \rangle$, where X is a set of variables, D is the set of variable domains, and C is the set of constraints. The algorithm also takes a partial instantiation I of the variables, representing the current state of the solution under construction.

2. **Base Case:** If the instantiation I is complete (all variables are assigned), then a solution has been found, and the algorithm returns **true**.

3. **Variable Selection:** If the instantiation is incomplete, the algorithm selects a variable that has not yet been instantiated (unassigned) from the set of variables X.

4. **Value Exploration:** The algorithm iterates through all possible values v in the domain $D(X_i)$ of the selected variable X_i.

5. **Local Consistency Check:** Before assigning a value to X_i, the algorithm checks whether the instantiation $I \cup \{X_i \leftarrow v\}$ is locally consistent, meaning all constraints are satisfied under this partial assignment.

6. **Recursive Call:** If the instantiation is locally consistent, the algorithm recursively calls BT to solve the problem with this new assignment. If the recursion returns **true**, it means a solution has been found, and the algorithm returns **true**.

7. **Backtracking:** If no value leads to a solution, the algorithm backtracks and tries another value for the previous variable.

8. **Return:** If all possibilities have been explored without success, the algorithm returns **false**, indicating that no solution has been found.

5 Local Consistencies in Constraint Programming

In CP, **local consistency** refers to a property where a part of the constraint network is made consistent, without guaranteeing that the entire problem is globally solved. Different levels of local consistency exist, including **Arc Consistency (AC)** and **Bound Consistency (BC)**.

5.1 Arc Consistency (AC)

Arc Consistency (AC) is a form of local consistency, stronger than BC. A problem is arc-consistent if, for each binary constraint $C(X_i, X_j)$ and each value $v_i \in D(X_i)$, there exists at least one value $v_j \in D(X_j)$ such that the constraint is satisfied. If not, v_i is removed from $D(X_i)$.

The *AC-3* algorithm is often used to enforce arc consistency by iterating over the constraints and removing inconsistent values. Although more computationally expensive than BC, AC enables better pruning of the search space.

5.2 Bound Consistency (BC)

Bound Consistency (BC) is another form of local consistency that considers only the extreme values of the variable domains. A binary constraint $C(X_i, X_j)$ is bound-consistent if, for the minimum and maximum values of X_i, there exists at least one value of X_j satisfying the constraint.

Unlike AC, which checks all values in a domain, BC is limited to the bounds, making it faster but less effective in filtering out inconsistent values.

For Boolean variables, BC and AC become **equivalent**. Since a Boolean domain is limited to two values, verifying BC amounts to checking all possible values, which is precisely what AC does.

5.3 Other Levels of Local Consistency

Other forms of local consistency include:

- **SAC (Singleton Arc Consistency)**: Ensures that each value of a variable can be extended to a consistent partial solution.
- **RPC (Restricted Path Consistency)**: Strengthens arc consistency by considering constraints that indirectly link three variables.
- **k-consistency**: A problem is **k-consistent** if any valid instantiation of $k-1$ variables can be extended to a k-th variable while satisfying the constraints.

5.4 Global Consistency and Domain Consistency

Global consistency means that the entire problem is made locally consistent at all levels of propagation, maximizing the reduction of the search space.

Domain consistency is a weaker property that ensures each value in a domain satisfies at least one constraint in the problem.

5.5 Global Constraints and Propagators

Global constraints involve multiple variables and allow modeling complex relationships. For example, the `AllDifferent` constraint ensures that all variables take distinct values.

To maintain a certain level of consistency on these constraints, solvers use **propagators**, which apply specialized algorithms to enforce a specific level of local consistency (AC, BC, or others). These techniques improve filtering efficiency and significantly reduce search time.

6 Strategies

Once the problem has been modeled, it is necessary to choose effective strategies to explore the solution space. Among the most commonly used strategies in CSP, we find:

- **Variable and Value Selection Heuristics: Heuristics** are strategies used to guide the search in the CSP solution space. They help make more informed decisions regarding the order of variable selection and the values assigned to these variables. The goal is to reduce computation time and increase the chances of quickly finding a solution by prioritizing the most promising choices.

    1. **Variable Selection Heuristics:** Choosing the variable to assign is crucial in solving CSP problems. A poor selection can lead to an inefficient exploration of the search space. Here are some common heuristics for variable selection: **Minimum Domain (Min Dom)**: This heuristic selects the variable with the smallest domain; **Degree Heuristic**: Prefers the variable that is involved in the largest number of constraints.
    2. **Value Selection Heuristics:** Once a variable is selected, the next heuristic determines which value to assign to it. As with variable selection, choosing the right value can significantly influence search efficiency. Here are some value selection heuristics: **Most Constraining Value**: This heuristic selects the value that is the most restrictive for other variables, meaning the value that most reduces the domains of neighboring variables; **Random Value**: This heuristic randomly selects a value from the variable's domain.

- **No-good:** *No-goods* are sets of decisions that inevitably lead to a conflict or contradiction. When a *no-good* is detected, all branches of the search space containing this configuration can be immediately pruned. For example, in a Sudoku problem, a *no-good* could be a combination of values violating the uniqueness constraint for numbers in a row.
- **Conflict Detection:** Conflict detection identifies constraint violations, generating *no-goods* to prevent revisiting these configurations. As soon as a conflict is detected, the algorithm backtracks and uses *no-goods* to eliminate conflicting solutions in future iterations.
- **Consistency:** Applying different levels of consistency, such as domain consistency or arc consistency, reduces the domains of variables by eliminating invalid values, thereby reducing the number of possibilities to explore.