CP

April 10, 2024

1 Constraint Programming

1.1 Introduction

Constraint Programming (CP) is a subset of discrete optimization, where the goal is to find an assignment of values to a set of variables that adheres to a set of predefined constraints. These constraints express the rules and restrictions that must be satisfied for a solution to be considered valid.

CP problemss have found wide-ranging applications across various domains, including: Solving puzzles like Sudoku, scheduling tasks, and planning. CP is particularly useful in situations where the problem is non-convex, combinatorial, or discrete in nature. Some examples include: Solving puzzles like Sudoku, scheduling tasks, planning, resource allocation, network configuration, and many more.

The power of CP lies in its declarative nature. Rather than focusing on the step-by-step process of finding a solution, CP allows you to focus on defining the characteristics of a valid solution. This makes it easier to represent and reason about complex problems with intricate relationships between variables.

In this notebook, we'll explore the basics of CP, including the components of CP, how to model a CP, and techniques to solve constraint problems using Python.

1.2 Constraint Programming vs Integer Programming vs Combinatorial Optimization

Constraint Programming: CP focuses on solving optimization problems by specifying constraints that must be satisfied rather than optimizing an objective function. It employs a declarative programming paradigm where relationships between variables are expressed as constraints, aiming to find a solution that satisfies all constraints. CP is versatile and can handle various types of constraints, including logical, arithmetic, and global constraints.

Integer Programming (IP): In contrast, IP is a mathematical optimization technique where variables are constrained to take integer values. It's a subset of linear programming, where both the objective function and constraints are linear, but the variables must be integers. IP is particularly useful for optimization problems where decisions involve discrete quantities, such as production planning or scheduling.

Combinatorial Optimization: This branch deals specifically with optimizing discrete structures, such as graphs, matroids, or other combinatorial structures. It's concerned with finding the best arrangement, combination, or selection of elements from a finite set to achieve an optimal solution.

Problems in this category include the traveling salesman problem, graph coloring, and the knapsack problem.

While these branches of discrete optimization are related and often overlap, they each have their own distinct approaches and techniques for solving optimization problems. Combinatorial optimization deals with discrete structures and arrangements, integer programming focuses on optimizing with integer variables, and constraint programming emphasizes finding solutions that satisfy specified constraints.

1.3 Mathematical Formulation

CPs are built upon three fundamental elements:

- **Variables:** These are the unknown quantities or decision points within the problem. Each variable has a specific domain associated with it.
- **Domains:** Domains represent the set of possible values that a variable can take on. Domains can be discrete (e.g., colors for map coloring) or continuous (e.g., a range of temperatures in a scheduling problem).
- **Constraints:** Constraints are the heart of a CP. They define the relationships and restrictions between variables, dictating which combinations of values are acceptable. Constraints can be expressed mathematically, logically, or even graphically.
- **Solution Function:** This function takes the components of a CP and returns 1 if the assignment satisfies all constraints, and 0 otherwise. This is also called the Consistency Check or Constraint Check.
- **Objective Function:** In some cases, CPs may have an objective function that needs to be optimized. This function quantifies the quality of a solution based on certain criteria.

A CP can be formally represented as a triple (V, D, C) where: (Russell & Norvig, 2020; Mackworth, 2013)

- $V = \{V_1, V_2, ..., V_n\}$ is a finite set of variables.
- $D = \{D_1, D_2, ..., D_n\}$ is a set of domains, where each D_i is the domain of possible values for the corresponding variable V_i.
- $C = \{C_1, C_2, ..., C_m\}$ is a finite set of constraints, where each constraint C_i restricts the values that a subset of the variables can simultaneously take.
- $S((V, D, C)) = 1$ if the assignment satisfies all constraints, and 0 otherwise. This is the solution function.
- $f((V, D, C))$ is the objective function that needs to be optimized. CP problems don't always have an objective function, for example, in constraint satisfaction problems where the goal is to find any valid solution.

1.3.1 Example 1: Assigning Tasks to Workers

Let's consider a simple resource allocation problem. We have three tasks (T_1, T_2, T_3) that need to be assigned to three workers (W_1, W_2, W_3). Each task can only be assigned to one worker, and each worker can only be assigned one task. We can model this problem as a CP with the following components:

Variables:

- T_1, T_2, T_3 representing the start time of each task.

- W_1, W_2, W_3 representing the worker assigned to each task.
- (T_i, W_j) representing the assignment of task i to worker j.

Domains:

- $\{W_1, W_2, W_3\}$ for each task (the possible workers who can be assigned).
- $\{T_1, T_2, T_3\}$ for each worker (the possible tasks they can be assigned to).

Solution Function:

A solution function, denoted as $f(\cdot)$, takes an assignment of task-worker pairs and determines if it satisfies all constraints:

- $f((T_i, W_j)) = 1$ if the assignment is valid (when job i can be assigned to worker j).
- $f((T_i, W_j)) = 0$ if the assignment is invalid (violates one or more constraints).

Constraints:

1. **Set Notation**

   No two tasks assigned to the same worker: $\{(T_i, W_j) \neq (T_k, W_j)\}$ for all $i \neq k$

   No worker assigned to two tasks: $\{(T_i, W_j) \neq (T_i, W_k)\}$ for all $j \neq k$

2. **Summation Notation**

   $\sum_{i=1}^{3} f((T_i, W_j)) = 1$ for all j (for all workers, exactly one task is assigned)

   $\sum_{j=1}^{3} f((T_i, W_j)) = 1$ for all i (for all tasks, exactly one worker is assigned)

Objective Function:

In this case, we may not have an objective function to optimize. The goal is to find any valid assignment that satisfies all constraints.

1.3.2 Example 2: Assigning Tasks to Workers to Minimize Project Duration

Unlike the previous example which did not have an objective function, this problem has a function we need to optimize. Consider four tasks (T_1, T_2, T_3, T_4) that need to be completed within a specific timeframe by two workers (W_1, W_2). Each task has a duration, and the objective is to minimize the total duration of the project. We can model this problem as a CP with the following components:

Variables:

- T_1, T_2, T_3, T_4 representing the start time of each task.
- W_1, W_2 representing the worker assigned to each task.
- (T_i, W_j) representing the assignment of task i to worker j.

Domains:

- $\{W_1, W_2\}$ for each task (the possible workers who can be assigned).
- $[0, T_{\max} - \text{duration}(T_i)]$ for each task i (time intervals of each task).

Solution Function:

A solution function, denoted as $f(\cdot)$, takes an assignment of task-worker pairs and determines if it satisfies all constraints:

- $f((T_i, W_j)) = 1$ if the assignment is valid (when job i can be assigned to worker j).
- $f((T_i, W_j)) = 0$ if the assignment is invalid (violates one or more constraints).

Constraints:

I will be using summation notation to represent the constraints in this example.

1. **Temporal Constraints**

   Task durations must not overlap for the same worker: $\sum_{i=1}^{4} \sum_{k=i+1}^{4} f((T_i, W_j)) \times f((T_k, W_j)) = 0$ for all j

   The multiplication of the function f helps detect overlapping start times of two tasks assigned to the same worker: a product of 1 indicates a constraint violation, while 0 shows compliance, aiding in identifying temporal or resource constraint violations.

2. **Resource Constraints**

   Each worker can only handle one task at a time: $\sum_{i=1}^{4} \sum_{k=i+1}^{4} f((T_i, W_j)) \times f((T_k, W_j)) = 0$ for all j and $W_j = W_k$

   Similar ideas as the temporal constraints, but this time, we're checking for overlapping tasks assigned to the same worker.

Objective Function:

The objective function here is to minimize the total duration of the project. It's represented as:

Minimize $\sum_{i=1}^{4} \text{duration}(T_i)$

In this setup, the goal is to find an assignment of tasks to workers and start times for each task that satisfies all constraints while minimizing the overall project duration. The objective function guides the search towards solutions that optimize the project's timeline.

1.4 Techniques to Solve Constraint Programming Problems

Solving a CP involves finding an assignment of values to variables that satisfies all constraints. Several techniques can be used to solve CPs, including: (Brailsford et al., 1999; Mackworth, 2013)

- **Backtracking:** A systematic search algorithm that explores the solution space by incrementally assigning values to variables and backtracking when a constraint violation is encountered.
- **Constraint Propagation:** A technique that enforces constraints locally to reduce the search space. This can be done through techniques like arc consistency, forward checking, and constraint propagation.
- **Local Search:** An optimization technique that iteratively explores the neighborhood of a solution to find better assignments. Techniques like simulated annealing and genetic algorithms can be used for local search.

1.4.1 Backtracking Algorithm

The backtracking algorithm is a widely used technique for solving CP problems. It systematically explores the solution space by incrementally assigning values to variables and backtracking when a constraint violation is encountered. The algorithm follows a depth-first search strategy and prunes the search space by detecting constraint violations early. (Brailsford et al., 1999; Mackworth, 2013)

The backtracking algorithm can be summarized as follows:

1. **Choose a variable:** Select an unassigned variable.
2. **Order the domain:** Order the domain of the selected variable.
3. **Assign a value:** Assign a value from the domain to the variable.
4. **Check constraints:** Check if the assignment satisfies all constraints.
5. **Recursive call:** If the assignment is valid, recursively call the algorithm on the next variable.
6. **Backtrack:** If the assignment is invalid, backtrack to the previous variable and try a different value.

1.4.2 Constraint Propagation

Constraint propagation, a technique central to solving CP Problems, employs various methods to prune the search space effectively. Two prominent methods within constraint propagation are **arc consistency** and **forward checking**.

Arc Consistency: This method ensures that for every pair of variables linked by a constraint, all possible values of one variable are compatible with at least one value of the other variable. In simpler terms, it checks that the constraints are satisfied between every pair of variables. If a variable's domain is reduced due to the constraint on another variable, arc consistency iterates through the constraints to ensure that all remaining values are consistent. (Brailsford et al., 1999)

Forward Checking: This technique is applied during the search process. Whenever a variable is assigned a value, forward checking examines the remaining unassigned variables to see if any are left with no valid values. If a variable's domain becomes empty, indicating no valid assignments, the algorithm backtracks to the previous decision point. Forward checking helps in detecting dead-end paths early, potentially reducing the search space and improving efficiency. (Brailsford et al., 1999)

Both arc consistency and forward checking contribute to constraint propagation by progressively narrowing down the possible solutions. They work in tandem to eliminate inconsistent values from variable domains, guiding the search towards feasible solutions more efficiently. These methods are crucial components of CP solvers, enhancing their ability to find solutions for complex problems like Sudoku, scheduling, and planning.

1.4.3 Local Search Algorithms

Local search algorithms explore the solution space by iteratively moving from one solution to another, aiming to improve the current solution. Two common local search algorithms are **hill-climbing** and **simulated annealing**.

Hill-Climbing: This algorithm starts with an initial solution and iteratively explores neighboring solutions by making small improvements. At each step, it moves to the neighboring solution with the best value, essentially "climbing uphill" towards the peak of the solution space. However, hill-climbing can get stuck in local optima, where the current solution is better than its neighbors but not the globally optimal solution. It tends to halt its search prematurely if no better solutions are found nearby. (Russell & Norvig, 2020)

Simulated Annealing: Inspired by the physical process of annealing in metallurgy, simulated annealing introduces randomness to explore the solution space more effectively. It starts with an initial solution and allows for "bad" moves with a certain probability, akin to heating and cooling in metallurgy. This randomness helps simulated annealing escape local optima and continue exploring

the solution space more broadly. As the algorithm progresses, the probability of accepting worse solutions decreases, mimicking the cooling process in annealing. Simulated annealing is particularly useful for finding global optima in complex solution spaces where hill-climbing might get trapped in local optima. Adhikari (2017)

Both hill-climbing and simulated annealing are examples of local search algorithms that trade-off between exploration and exploitation. While hill-climbing focuses on exploiting nearby solutions by always moving to the best neighbor, simulated annealing balances exploration by accepting worse solutions with a decreasing probability, allowing it to escape local optima and search for better global solutions. These techniques are widely used in optimization problems where finding the best solution might be challenging or computationally expensive.

1.5 References

- Russell, S., & Norvig, P. (2020). Artificial Intelligence: A Modern Approach (4th ed.). Pearson.
- Brailsford, S. C., Potts, C. N., & Smith, B. M. (1999). Constraint satisfaction problems: Algorithms and applications. European Journal of Operational Research, 119(3), 557–581. https://doi.org/10.1016/s0377-2217(98)00364-6
- Adhikari, B. (2017, February 4). The simulated annealing algorithm explained with an analogy to a toy. YouTube. https://www.youtube.com/watch?v=eBmU1ONJ-os
- Mackworth, A. (2013). Solving Constraint Satisfaction Problems (CSPs) using Search. https://www.cs.ubc.ca/~mack/CS322/lectures/3-CSP2.pdf