Constraint Satisfaction Problems

Daniel Gigliotti

1 Constraint Satisfaction Problems

Constraint Satisfaction Problems (CSPs) involve finding a solution that satisfies a set of constraints or conditions. These problems are commonly used in various domains, including scheduling, planning, optimization, and decision-making.

Constraint Satisfaction Problem: find an **assignment**
of values for each variable from its respective domain
such that every constraint is satisfied.

Constraint Satisfaction Problems consist of:

- A set of **variables**
$$V = \{v_1, v_2, ..., v_n\}$$
- A set of **domains**, one for each variable
$$D = \{D_{v_1}, D_{v_2}, ..., D_{v_n}\}$$
- A set of **constraints** (binary relations), conditions that the solution must satisfy
$$C = \{C_{\{u,v\}}\}$$
$$u, v \in V, u \neq v, C_{\{u,v\}} \in C$$

We can visualize a CSP with a **constraint graph**, that is an undirected graph with variables as vertices and constraints as edges (the graph has an arc $\{u, v\}$ iff $C_u v \in C$.

A **partial assignment** assigns **some** variables to values from their respective domains.
A **total assignment** is an assignment defined on all the variables.
We say that a partial assignment is **consistent** if it does not violate any constraint.
A total assignment that is consistent is a **solution** for the problem.
A problem γ is **solvable** if there is a total consistent assignment α that is a solution for γ; if such solution does not exists, the problem is **inconsistent**.
Let α be a partial assignment; we say that α can be **extended to a solution**

if there exists a solution α' that agrees with α on the variables where α is defined.

An algorithm is **complete** if it terminates with a solution when one exists. An algorithm is **sound** if it does not yield any results that are untrue.

2 Backtracking

Backtracking is a general algorithmic technique used to solve constraint satisfaction problems very efficiently. It is much better than enumerating and solution-checking all total assignments because a partial assignment that is already inconsistent is excluded and no longer clutters when computing the solution.

The naive form of backtracking is the following:

```
def naive_backtracking(γ, α):

    if α is inconsistent with γ:
        return "inconsistent"

    if α is a total assignment:
        return α

    select some variable v for which α is not defined
    for each d ∈ D_v in some order do:
        α' := α ∪ {v = d} (instantiate the variable v with d)
        α'' := naive_backtracking(γ, α')
        if α'' is not inconsistent:
            return α''

    return "inconsistent"
```

Backtracking recursively instantiate variables one by one, backing up out of a search branch if the current partial assignment is already inconsistent. Backtracking does not recognize α that cannot be extended to a solution, unless α is already inconsistent. We can employ **inference** for this and speed up the algorithm. Other potential improvement points are highlighted in bold.

Backtracking is **complete**: it will find the solution.

- A widely used **variable ordering strategy** is to **choose the most constrained variable first**: this way we reduce the branching factor (number of sub-trees generated for the variable) and thus reduce the size of the search tree.
- A widely used **value ordering strategy** is to **choose the least constraining value first**: this way we increase the chances not to rule out the solutions below the current node.

3 Inference

The backtracking algorithm can be extended with inference techniques. Inference reduces the search space.

Inference: adding constraints without losing solutions;
we obtain an equivalent network with a tighter description
and hence a smaller number of consistent partial assignments.

We deduce additional constraints (unary or binary) that follows from the already known constraints and hence are satisfied in all solutions.

We have two inference methods:

- Forward checking;
- Arc consistency;

Two networks γ and γ' are equivalent if γ has the same solutions as γ'. A network γ is tighter than γ' if γ has the same constraints as γ' plus some others.

- $\gamma \subset \gamma' \rightarrow \gamma$ is strictly tighter than γ'
- $\gamma \subseteq \gamma' \rightarrow \gamma$ is tighter than γ'

Equivalence + tightness = inference. Let γ and γ' be constraint networks such that $\gamma \equiv \gamma'$ (γ is equivalent to γ') and $\gamma' \subset \gamma$ (γ' is tighter than γ). Then γ' has the same solutions as γ (definition of equivalence), but less consistent partial assignments than γ:

γ' is a better encoding of the underlying problem.

Given a total assignment α:

- if α is consistent with γ, it could be inconsistent with γ'.
- if α cannot be extended to a solution in γ, the same thing holds for γ' ($\gamma \equiv \gamma'$).

```
def naive_backtracking(γ, α):

    if α is inconsistent with γ:
        return "inconsistent"

    if α is a total assignment:
        return α

    # inference
    γ' := a copy of γ
    γ' := inference(γ')
    if exists v with D'_v = θ:
        return "inconsistent"

    select the most constrained variable v for which α is not defined
    for each d ∈ D_v (least constraining value):
        α' := α ∪ {v = d}
        α'' := naive_backtracking(γ, α')
        if α'' is not inconsistent:
            return α''

    return "inconsistent"
```

Inference(γ) is any procedure that delivers a tighter equivalent network.

- **forward checking**: from assigned to unassigned variables; after an assignment to a variable, look at the constraints of that variable, take the other end of each constraint (since it is a binary constraint) and remove the values of that end that would not comply with the assignment just made.
- **arc consistency**.

  Consistency: for every constraint and every domain value, at least one value on the other side of the constraint needs to make the constraint valid.

  - **Revise(γ, u, v)** is an algorithm enforcing consistency of u relative to v ($O(k^2)$), for just a pair of variables.
  - **AC-1**: apply Revise(γ, u, v) up to a fixed point with redundant computations.
  - **AC-3**: apply Revise(γ, u, v) up to a fixed point remembering the potentially inconsistent variable pairs.

Forward Checking

```
def ForwardChecking(γ, a):

    for each v where a(v) = d' is defined do:
        for each u where a(u) is undefined and C_{u,v} ∈ C do:

            # only leave valid values in D_u, values that satisfy
            # the constraints of u with v
            D_u := {d|d ∈ D_u, (d, d') ∈ C_{u,v}}

    return γ
```

Revise

```
def Revise(γ, u, v):

    for each d ∈ D_u:
        if there is no value ∈ D_v for which C_{u,v} is satisfied:
            remove d from D_u
            D_u := D_u/{d}

    return γ
```

AC-1

```
def AC1(γ):

    changes_made = False
    while !changes_made:

        for each constraint C_{u,v}:

            γ' = Revise(γ, u, v)
            if D_u reduces:
                changes_made = True

            γ' = Revise(γ, v, u)
            if D_v reduces:
                changes_made = True

    return γ
```

AC-3

def AC3(γ):

$M := \emptyset$
for each constraint $C_{u,v} \in C$ do:
    $M := M \cup \{(u,v),(v,u)\}$

while $M \neq \emptyset$ do:
    remove any element (u,v) from M
    Revise(γ, u, v)
    if D_u has changed in the call to Revise, then:
        for each constraint $C_{w,u} \in C$ where $w \neq u$ do:
            $M := M \cup \{(w,u)\}$

return γ

AcyclicCG

def AcyclicCG(γ):

```
# get a directed tree from γ picking an arbitrary
# variable v as root and directing arcs outwards
γ' ← GetDirectedTree(γ, v)

# order the variables topologically i.e. such that each
# vertex is ordered before its children; denote that
# by v_1, v_2, ..., v_n
vertex_list = Sort(γ')

# for i := n, n-1, ..., 2
for i in range(len(vertex_list), 2, -1):
    Revise(γ, v_parent(i), v_i)
    if D_p = ∅:
        return "inconsistent"

# now every variable is arc consistent relative to its children.
# Run backtracking with inference with forward checking using the
# variable order v_1, v_2, ..., v_n
Backtrack(vertex_list)
```