Lecture 9: Constraint Satisfaction Programming and Scheduling

Viliam Lisý & **Branislav Bošanský**

Artificial Intelligence Center
Department of Computer Science, Faculty of Electrical Eng.
Czech Technical University in Prague

bosansky@fel.cvut.cz

February, 2021

Overview

What we have covered so far:

- (un)informed search
- reinforcement learning
- two-player games

In all these problems, we have not assumed that states of the world have some specific structure.

Question

What if we restrict the structure of the states?

Question

What if we restrict the structure of the states?

- – we lose generality (not every problem could be represented)
- + we gain performance (we will be able to solve much larger problems)

We can identify and solve (exactly!) instances of a subclass of problems and improve scalability by several orders of magnitude compared to standard search algorithms.

N-Queens Example

Consider an N-Queens problem: Place on a chessboard of size $N \times N$ squares N queens so that no two queens threaten each other. For $N = 4$:
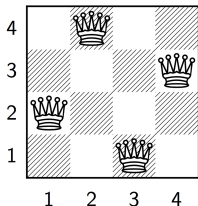
What would be the state representation?

- N coordinates (one tuple of coordinates for each queen)
- N numbers (every queen has to be in a different column, we can only represent rows)

Action changes position of one (or more) queen.

Differences from previous (general) problems:

- there is no start state (we can start from any state), hence
- the path to the goal state is not interesting, only the goal state itself

Constraint Satisfaction Problems (CSPs)

The class of problems that include the N-Queens problems are known as CSPs (subclass of NP-complete problems).

CSPs are defined by 3 finite sets:

- **variables** $(x_1, x_2, \ldots, x_n)$
- **domains** (D_i for each variable x_i)
- **constraints** $(c_1, c_2, \ldots, c_m)$

A constraint is specified as a tuple of

- subset of variables $x_{j_1}, \ldots, x_{j_l}$
- all allowed joint assignments (l-tuples from $D_{j_1}, \ldots, D_{j_l}$)

Goal: find such an assignment values to variables that satisfy all the constraints

Many problems can be represented as CPSs. These include known puzzles:

- Sudoku,
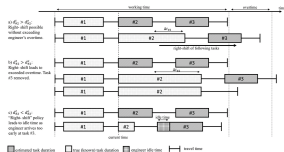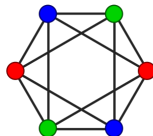- Cryptaithmetic,

essential NP problems:

- SAT,
- Graph Coloring,

and many practical problems:

- Scheduling

$$
\begin{array}{r}
S\ E\ N\ D \\
+\ \ M\ O\ R\ E \\
\hline
M\ O\ N\ E\ Y
\end{array}
$$

N-Queens Example as a CSP

We can formulate N-Queens problem as a CSP:

- **variables**: $x_1, \ldots, x_N$ (one variable for each queen, queen i is placed in the i-th column)
- **domains**: $D_i = \{1, \ldots, N\}$ (the row in which the queen is placed)
- **constraints**:
  - $x_i \neq x_j \qquad \forall i, j \in \{1, \ldots, N\}, i \neq j$
    (some solvers support global constraint
    alldifferent($x_1, \ldots, x_N$**)**)
  - $|x_i - x_j| \neq |i - j| \ \ \forall i, j \in \{1, \ldots, N\}, i \neq j$

Question

How do we search for a solution?

Search Tree for CSPs

We use (uninformed) search as we know it (for now) and represent the search space as a search tree.

What are the nodes and actions in the search tree for a CSP?

- **Nodes** in the search tree – (partial) assignment of values to variables,
- **Edges** – choosing an unassigned variable and assign a value to this variable.

During the assignment, the algorithm must check whether the assignment does not violate constraints.

If there is no satisfying assignment, the algorithm backtracks.

Standard Representation of CSPs
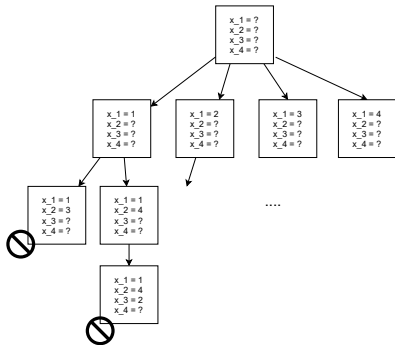
We now move to specific CSP algorithms. Many of them assume only **binary constraints**.

Question

Is it a problem? Is it a subclass of CSP problems?

Not really, we can reformulate any k-ary constraint as a set of binary constraints:

- Assume there is a constraint c involving k variables. Let Γ be the set of all k-tuples that satisfy this constraint.
- Create a new variable x_c with the domain Γ and create k binary constraints with involved k variables, such that i-th item of the value of x_c equals to value of the variable i.

Standard Representation of CSPs – Visualization

Having only binary constraints, we can visualize CSPs as graphs:

- variables are vertices in the graph,
- constraints are edges in the graph.

There is an edge connecting two vertices if there is a constraint between these variables.
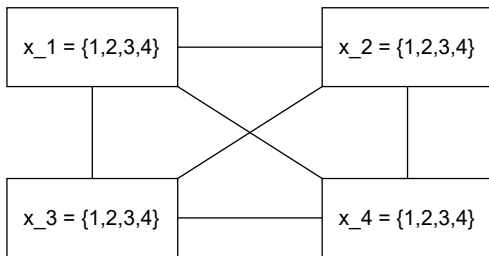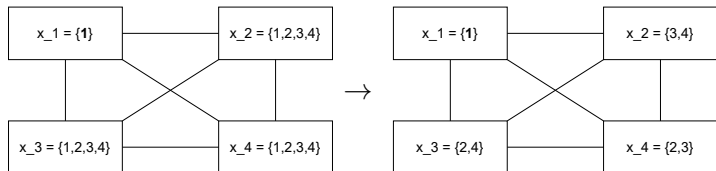
CSP Search and Propagating Constraints

Can we utilize the fact that we have a specific structure of the problem?

The simple search checks the constraints only in a passive way.

We can propagate the values to other variables. Every time we set a value for some variable, we can filter out values of other variables that do not satisfy constraints → **forward checking**:

CSP Search and Propagating Constraints

Assume the search algorithm selects a value for variable x_i. Now:

- for every other variable x_j such that there is a constraint c_{ij} between x_i and x_j, we evaluate all available values from D_j and keep only those that satisfy c_{ij}

How is the forward checking integrated into the search algorithm?

- the algorithm keeps available values for every variable
- if for any variable its domain is empty after the forward checking, the algorithm immediately backtracks

First heuristic $\rightarrow$ **minimal remaining value (MRV)**.

So far, there was no rule which variable to choose next in the search tree. MRV heuristics is a fail-fast heuristic that can quickly prune out dead-ends.

pseudocode of the search algorithm:

- **if** all variables are assigned **then return** current assignment (solution)
- $x_i \leftarrow$ ChooseVariable(X, D)
- for each $v \in D_i$
    - assign $x_i = v$
    - valid $=$ ForwardChecking(X, D, i, v)
    - **if** valid **then** search(X, D)
    - undo local assignments
- **return** false

Towards a Better Use of Constraints

Forward checking ensures that there are supporting values in domains of other involved constraints.

The algorithm removes those values that do not satisfy the constraints.

But this can violate some other constraints ... Is there a way we can ensure that every constraint **can be satisfied** (termed **consistent)**?

Yes! We can have an algorithm that makes every edge (constraint) consistent.

Arc Consistency

Making one edge (arc) c_{ij} consistent:

- deleted = false
- **for each** $v \in D_i$
    - supported = false
    - **for each** $v' \in D_j$
        if $c_{ij}(v, v')$ **then** supported = true
    - **if not** supported **then**
        remove v from D_i
        deleted = true

- **return** deleted

The procedure checks one constraint (in a directed manner) and returns true if some value was removed from domain D_i.

Arc Consistency – AC-3

Assume the algorithm has set value for variable x_i. We need to make consistent all incoming edges to node i (constraints that depend on this selected value). Next, if some value is removed from any variable x_j, we need to do the same for node j.
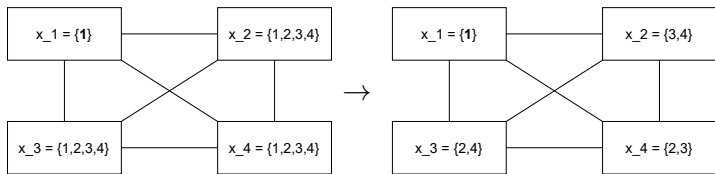
We will have a queue Q of all edges to make consistent:

- $Q = \{(j, i) \mid c_{ji} \in C, \ i \neq j\}$
- **while** Q is **not empty**
    - $(a, b) = \text{pop}(Q)$
    - **if** MakeConsistent(a, b) **then**
        append$(Q, \{(k, a) \mid c_{ka} \in C, \ k \neq a\})$
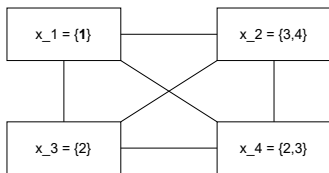
This algorithm is known as **AC-3**.

AC3 Algorithm – Example

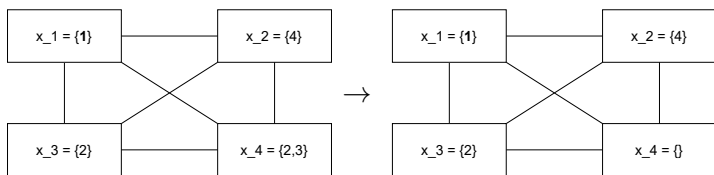Step 1: making consistent all edges $(n, 1)$ for $n = \{2, 3, 4\}$

Step 2: making consistent all edges $(n, 2)$ (AC3 deleted from D_2)

Value 4 is removed from D_3 since it is not supported by any value in D_2.

AC3 Algorithm – Example

Step 3: making consistent all edges $(n, 3)$

Value 3 is removed from D_2 since it is not consistent with $x_3 = 2$.

Next, all values in D_4 are removed since neither of them is consistent with $x_3 = 2 \rightarrow$ no solution!

The AC3 algorithm can determine after the first assignment $x_1 = 1$ that this action does not lead to goal.

AC3 Algorithm

Question

Does AC3 solve everything? Do we still need search?

Unfortunately, AC3 is not able to guarantee there exists a solution.
If AC3 prunes out some domain, the search algorithm can safely
backtrack. Otherwise, the search needs to continue.

Other Interesting Search Improvements in CSP
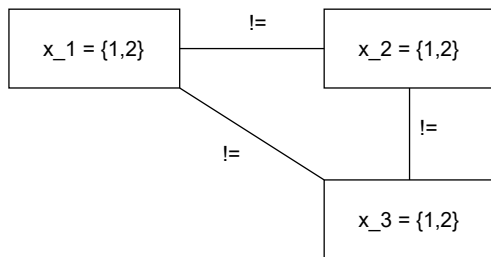
Least Constraining Value
Another heuristic for CSPs – among all the values to be assigned to a variable, choose such that supports the most other values.

Backjumping
Inability of choosing valid value for one variable can be caused by a choice of a variable up in the search tree. $\rightarrow$ The algorithm can identify which variables cause the conflict and can backtrack immediately to this conflicting variable (jumping back).

Dynamic Backtracking
In backjumping, the assignment between two conflicting variables is lost if we jump (even if it was a good one) $\rightarrow$ dynamic backtracking can dynamically choose which variable to assign (or re-assign) so that partially valid solutions are not lost.

Constraint Optimization Problems (COPs)

Constraints in CSPs are **hard constraints** – they need to be satisfied to find a solution.

Often, not all constraints have to be hard – we can combine CSPs with an objective function representing **soft constraints**.

For example in scheduling – we cannot plan execution of two jobs on a single machine (hard constrain) but we want to minimize time required to finish all the jobs (objective function).

After finding a solution, we can keep it and continue searching – current solution is a lower bound on the optimal value (if we maximize), hence we can add additional pruning – **branch and bound** (similar to alpha beta pruning).

There are many additional modifications and improvements regarding CSP.

There are many solvers that you can use.

Again, the ideas from CSP algorithms can be used also elsewhere (using specific structure to prune out not perspective branches, exploiting locality of partial solution in dynamic backtracking, etc.)

A good source for a quick reference – CSP course at MFF CUNI.