CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

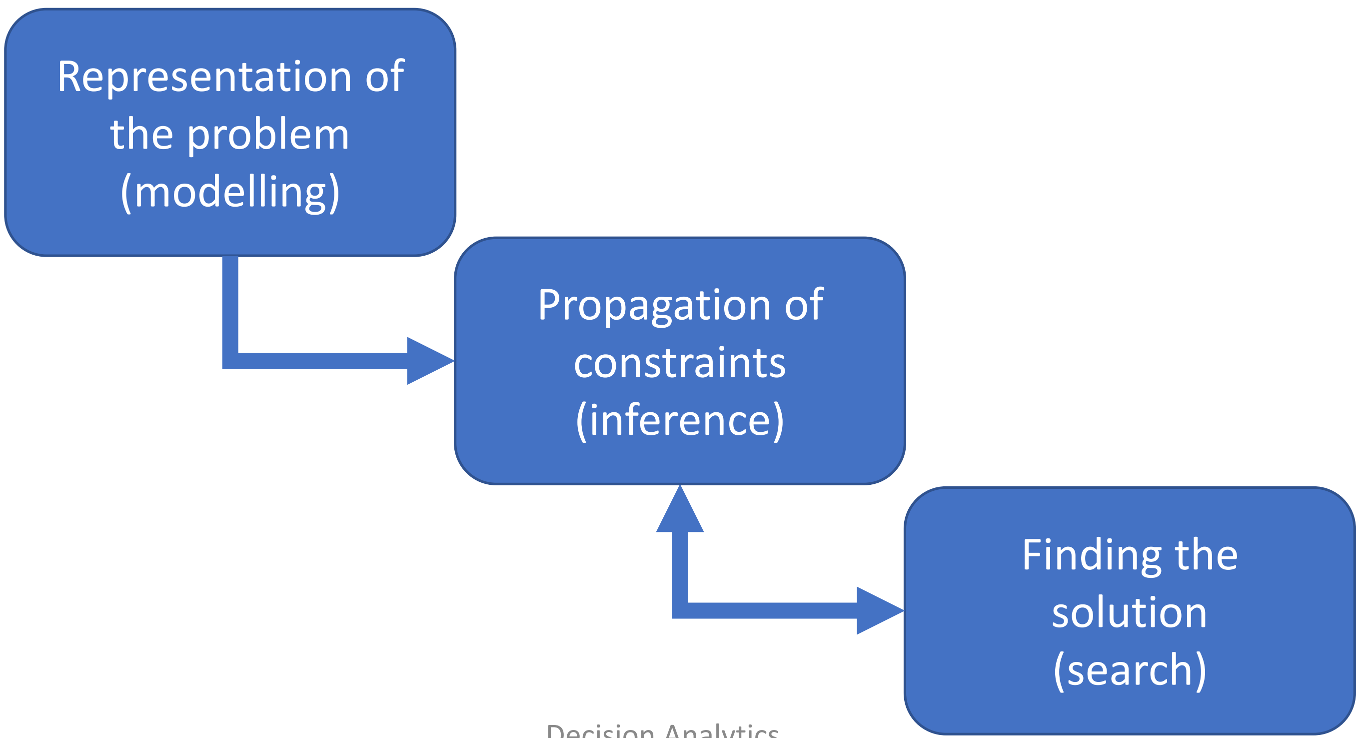
Computing@CIT

Decision Analytics

Lecture 15: Algorithmic view on Constraint Propagation

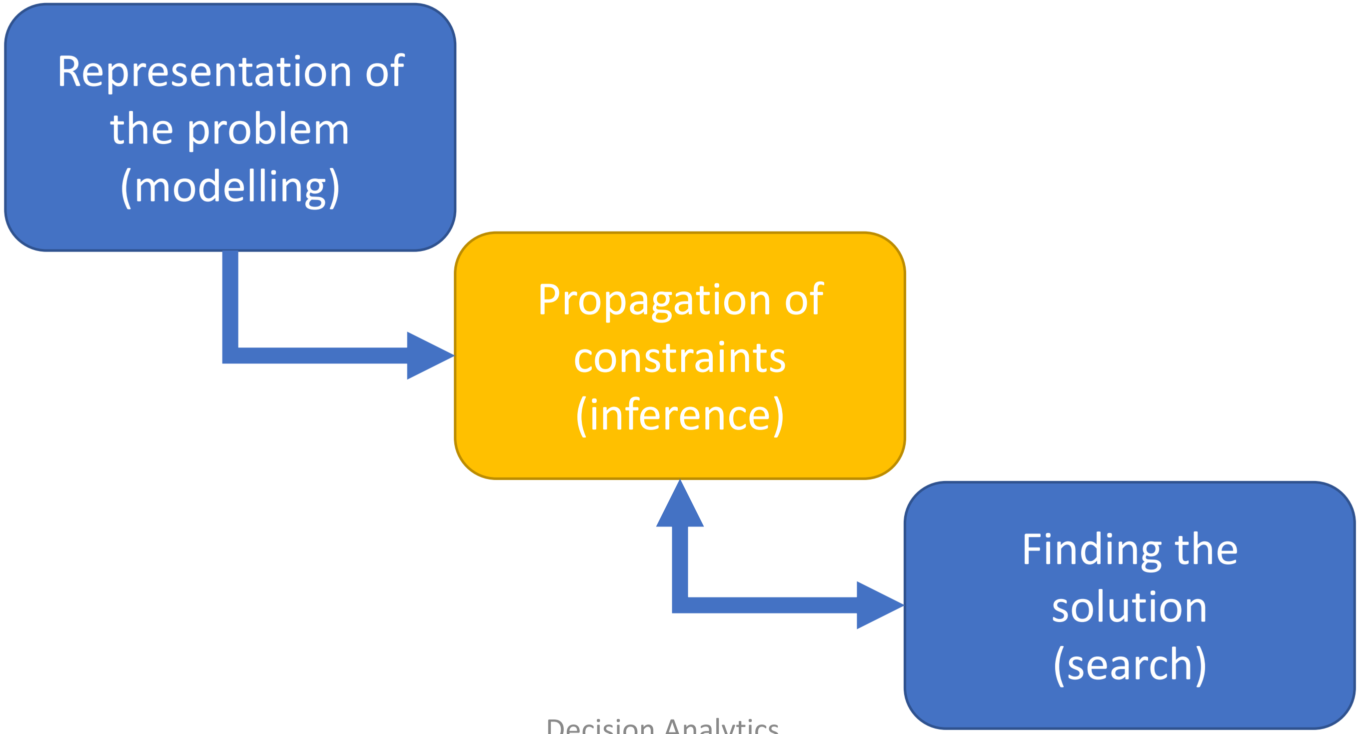
Constraint Programming

- Constraint Programming (CP) is a paradigm for solving combinatorial constraint satisfaction and constrained optimisation problems using a combination of modelling, propagation, and search



Constraint Programming

- Constraint Programming (CP) is a paradigm for solving combinatorial constraint satisfaction and constrained optimisation problems using a combination of modelling, propagation, and search
- This lecture is about **constraint propagation**



Constraint network

- A constraint network (X, D, C) is defined by
  - A sequence of n **variables**
    $$X = (x_1, \dots, x_n)$$
  - A **domain** for X defined by the domains of the individual variables
    $$D = D(x_1) \times \cdots \times D(x_n)$$
  - A set of **constraints**
    $$C = \{c_1, \dots, c_e\}$$

- A network is **normalised** if two different constraints do not contain exactly the same variables, i.e. $c_i \neq c_j \Rightarrow X(c_i) \neq X(c_j)$

AC3 Algorithm

function `Revise3`(**in** x_i: variable; c: constraint)**: Boolean** ;
  begin
    CHANGE $\leftarrow$ **false**;
    foreach $v_i \in D(x_i)$ **do**
      if $\nexists \tau \in c \cap \pi_{X(c)}(D)$ *with* $\tau[x_i] = v_i$ **then**
        remove v_i from $D(x_i)$;
        CHANGE $\leftarrow$ **true**;
    return CHANGE ;
  end

function `AC3/GAC3`(**in** X: set)**: Boolean** ;
  begin
    /* initalisation */;
    $Q \leftarrow \{(x_i, c) \mid c \in C, x_i \in X(c)\}$;
    /* propagation */;
    while $Q \neq \emptyset$ **do**
      select and remove (x_i, c) from Q;
      if `Revise`(x_i, c) **then**
        if $D(x_i) = \emptyset$ **then** return **false** ;
        else $Q \leftarrow Q \cup \{(x_j, c') \mid c' \in C \wedge c' \neq c \wedge x_i, x_j \in X(c') \wedge j \neq i\}$;
    return **true** ;
  end

Forward checking

- Constraint propagation is typically not called in isolation
- It is usually embedded in a search procedure, which creates partial instantiations of variables and needs to assess their consistency
- This consistency check, where we test for arc consistency between assigned and unassigned variables, is called **forward checking**

Forward checking

- A binary network $N = (X, D, C)$ is **forward checking consistent** according to an instantiation I on Y, if for all pairs of variables $x_i \in Y$ and $x_j \in X - Y$ the variable x_j is arc consistent for all constraints $c_{ij} \in C$ between the two

- Incrementally building a forward checking consistent network by adding instantiated variables then only requires to check arc consistency for the newly added variable

procedure $FC(N, Y, x_i)$;
1 **foreach** $c_{ij} \in C_N \mid x_j \in X \setminus Y$ **do**
2   **if** ***not*** $\texttt{Revise}(x_j, c_{ij})$ **then** return false

Consistencies on bounds

- AC3 checks for each value for consistency in the *Revise* call
- Instead of checking consistency value-by-value we could check the consistency only for the upper and lower bound of the domain
- Obviously this only makes a difference for integer (i.e. not Boolean) domains with more than two values

- A network $N = (X, D, C)$ is **bounds consistent**, if for all constraints $c \in C$ and all variables $x_i \in X(c)$

$$\min D(x_i) \in \pi_{\{x_i\}}(c \cap \pi_{X(c)}(D))$$
$$\max D(x_i) \in \pi_{\{x_i\}}(c \cap \pi_{X(c)}(D))$$

Reduction rules

- Up until now we have defined consistencies to characterise the result of constraint propagation

- Another way of approaching the problem is to characterise the procedure of constraint propagation and define the propagation algorithm instead

Reduction rules

- For a given network N a **reduction rule** is a function $f: \mathcal{P}_{\mathrm{N}} \rightarrow \mathcal{P}_{N'}$, that maps a network N' ∈ $\mathcal{P}_N$ (that is a tightening of N) to a network $f(\mathrm{N}') \in \mathcal{P}_{N'}$ (that is a tightening of N')

- The basic idea is that the application of reduction rules is tightening the network, thereby reducing the search space for potential solutions
- Consistencies are not necessarily considered in this approach

Propagator

- A propagator is a reduction rule that reduces variable domains based on a single constraint only (ignoring all other constraints, and only tightening domains)

- More formally, a **propagator** for the constraint c is defined as a reduction rule f for the single-constraint network $N_c = (X, D, \{c\})$, that maps networks $N' = (X, D', C') \in P_{N_c}$ to $f(N') = (X, D'', C')$ with $D'' \subset D'$

- A propagator f is **monotonic**, if $\mathrm{N}_1 \preccurlyeq N_2 \Rightarrow f(N_1) \preccurlyeq f(N_2)$
- A propagator f is **idempotent**, if $f\big(f(N)\big) = f(N)$
- Propagators f and g are **commutative**, if $f\big(g(N)\big) = g\big(f(N)\big)$

Propagator Iteration

- The question is, if the iterated application of a set of propagators will lead to a stable outcome, or if the network will simply keep on changing?

- More formally, we ask for a **fixpoint** of the network N with respect to a set of propagators $F = \{f_1, \dots, f_k\}$, which is defined by iteratively applying all propagators to the network $N_i = f_{k_i}(N_{i-1})$ until it is stable, i.e. until $N_j = f\ (N_{j-1})$ for all propagators $f \in F$

- If the all propagators are monotonic, them this fixpoint exists and is unique

Propagator Iteration

- Example: for a network $N = (X, D, C)$ we define the following propagators for each constraint $c_j \in C$ and each variable $x_i \in X(c_j)$

$$f_{ij}(X, D', C) = (X, D'', C)$$

- To only change the domain of that variable according to the projection of the constraint onto that variable

$$D''(x_i) = \pi_{\{x_i\}}(c_j \cap \pi_{X(c_j)}(D'))$$

- Repeated iteration of this propagators will terminate in a fixpoint, which is the arc consistent closure of N
- Therefore, this is another way of defining arc consistency and an algorithm to achieve it

AC3 as propagator

- The *Revise* call of the AC3 algorithm can be seen as a propagator
- Going through all values and checking all constraints is a costly operation
- Observation: if we remove a value from the middle of the domain, this does usually have not a great effect on the other constraint
- Therefore, can we distinguish different types of change events in order to improve performance (at the expense of achieving guaranteed arc consistency)?

AC3 as propagator

- We can distinguish different change events and consider the type of change in the calls to *Revise* that are triggered by the event

- There are four distinct events, that can be handled differently
  - A value is removed from the middle of the domain:
    - We don’t need to do anything here
  - The minimum of the domain boundary is increase
    - Remove all values from the domain below the new minimum
  - The maximum of the domain boundary is decreased
    - Remove all values from the domain above the new maximum
  - A domain becomes instantiated, i.e. the domain size becomes 1
    - Remove all values but the last remaining from the domain

Summary

- Constraint propagation tightens the constraint network so that less potential combinations for variable assignments are possible
- There are two ways to characterise the constraint propagation approach: by looking at the level of consistency that is achieved or by defining the constraint propagators algorithmically
- The three most commonly used forms of consistency are
    - Node consistency
    - Arc consistency
    - Path consistency
- We can also define constraint propagation through propagators
- The problem is NP-hard, therefore finding a solution through constraint propagation in theory requires exponential runtime (unless P=NP)
- We still need to search for the final solution using backtracking search

Thank you for your attention!