CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

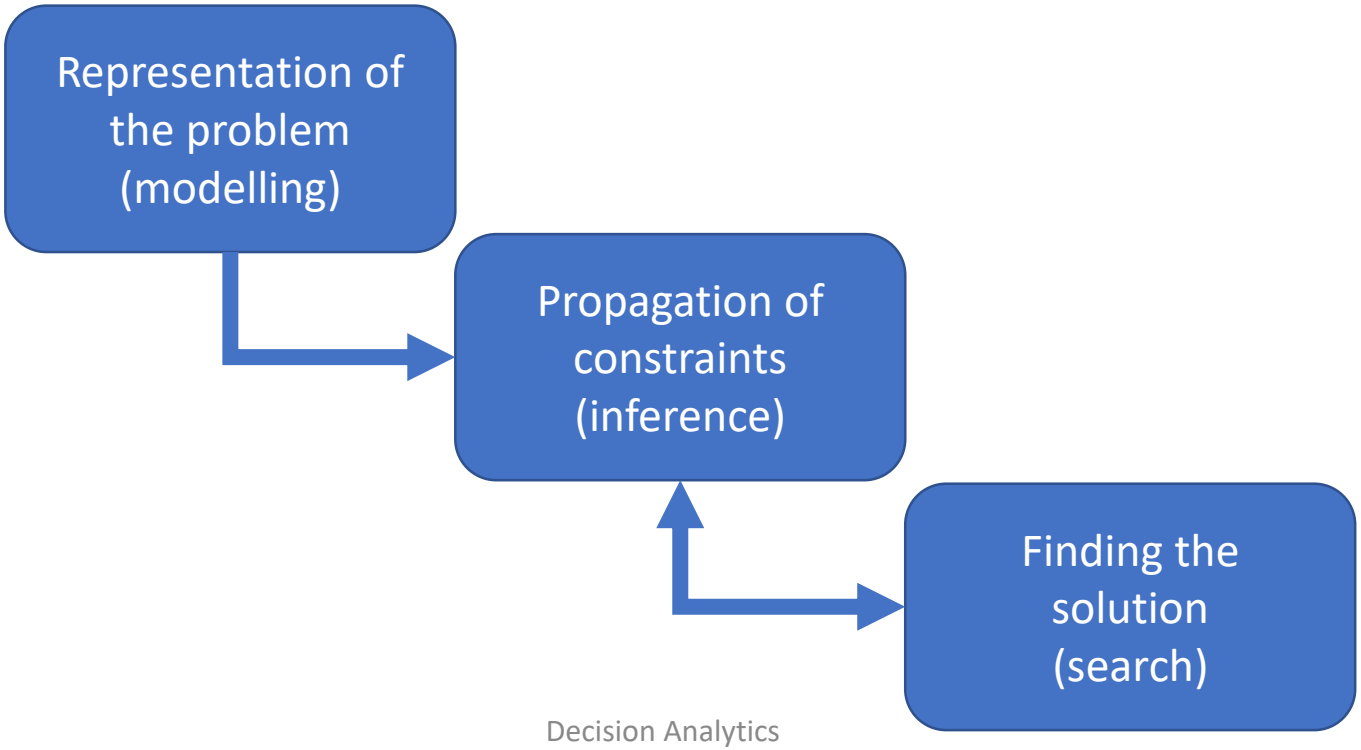
Computing@CIT

Decision Analytics

Lecture 17: Improving Backtracking Search

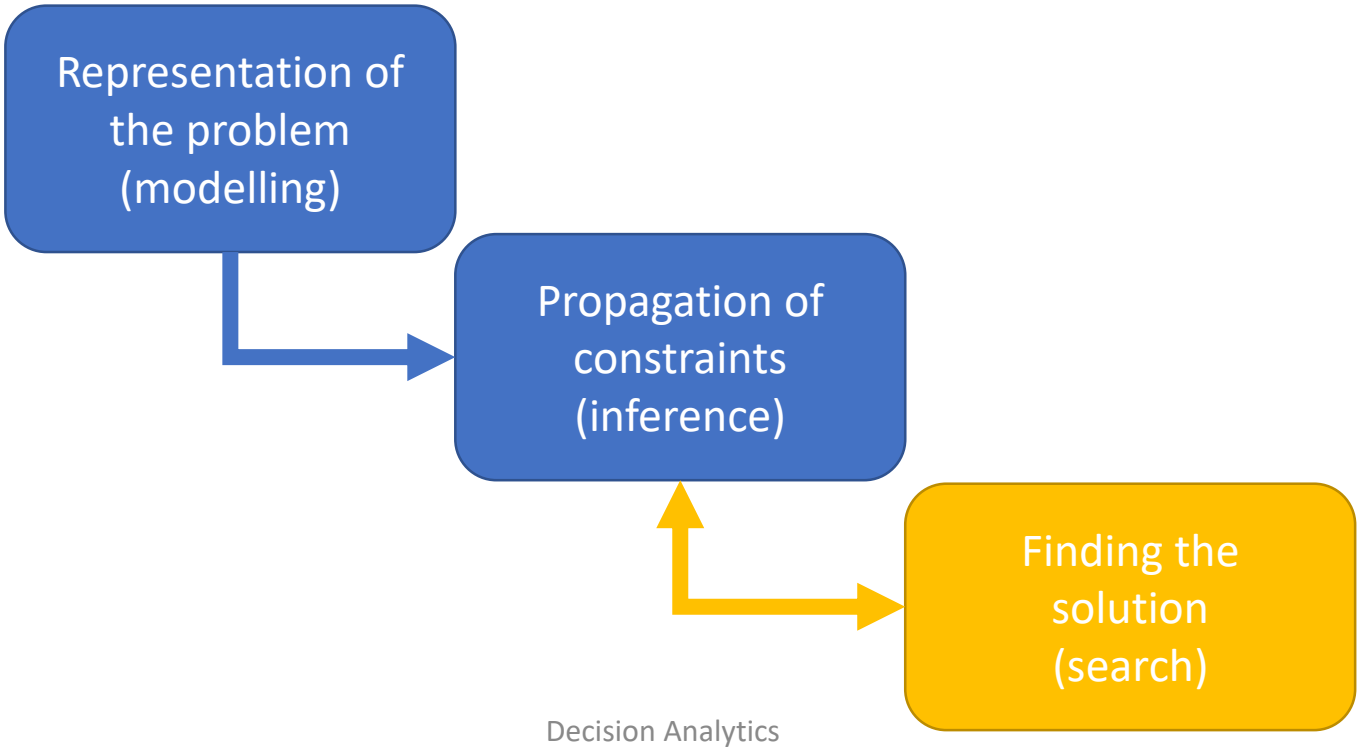
Constraint Programming

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



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 **backtracking search**, which very closely linked with constraint propagation



Branching constraints

- How can we formalise this consistent with the concepts from constraint propagation?

- In every node of the search tree we need to make a **decision** what variable we want to explore further and what value we want to assign during this exploration
- More generally, we need to decide on a **branching constraint** b that we want to explore by looking at the network $N' = (X, D, C \cup \{b\})$ and see if it can be tightened to a fully instantiated network
- Note, that a generic constraint does not need to be limited to just 1 variable, nor does it need to be assigning only 1 value to that variable

- This decision making process is called a **branching strategy**

Maintaining Arc Consistency (MAC)

- Search and constraint propagation are inherently linked in constraint programming
- The following algorithm propagates in each node to maintain arc consistency

```
Search(p, (X, D, C)):
        (X, D', C) = make arc consistent (X, D, C)
        if ∃D'(x_i): |D'(x_i)| = 0
                (X, D', C) is a no-good
                return
        else if ∀D'(x_i): |D'(x_i)| = 1
                (X, D', C) is a solution
                return
        else
                choose x_i so that |D'(x_i)| > 1
                choose v for b^1 = {x_i ≤ v}, b^2 = {x_i > v}
                Search(p ∪ {b^1}, (X, D', C ∪ {b^1}))
                Search(p ∪ {b^2}, (X, D', C ∪ {b^2}))
```

No-good recording

- When the MAC algorithm runs into a dead-end it simply backtracks one level up in the search tree
- This potentially results in discovering the same no-good network several times during the search procedure
- If we could identify the conditions that led to a branch of the search tree not leading to any solution, these conditions could be used to avoid repeating the same mistakes
- The idea of no-good recording is to discover no-good networks and add **implied constraints**, i.e. solution-preserving constraints that identify the no-good early on in the search tree

No-good recording

- Let the node $p = \{b_1, \dots, b_j\}$ be a dead-end node in the search tree and $|D'(x_i)| = 0$ the domain of the variable that collapsed

- Then an **eliminating explanation** of this collapse can be defined for each value $v \in D(x_i)$ that has been removed from the domain

- It is a subset of branching decisions sufficient for determining that assigning $(\mathrm{x_i}, v)$ will not lead to a solution anymore

$$e(x_i \neq v) = \{b_i \in p | sol(X, D, C \cup \bigcup_i b_i \cup \{x_i = v\}) \neq sol(X, D, C)\}$$

No-good recording

- Then the **jump-back no-good** for that node is defined as the union over all eliminating explanations for all values

$$J(p) = \bigcup_{v \in D(x_i)} e(x_i \neq v)$$

- We can extend this definition recursively to non-leaf nodes, where all sub-trees $\{b_{j+1}^1, \ldots, b_{j+1}^k\}$ failed as

$$J(p) = \bigcup_{i=1}^{k} J\left(p \cup b_{j+1}^i\right) - \{b_{j+1}^i\}$$

No-good recording

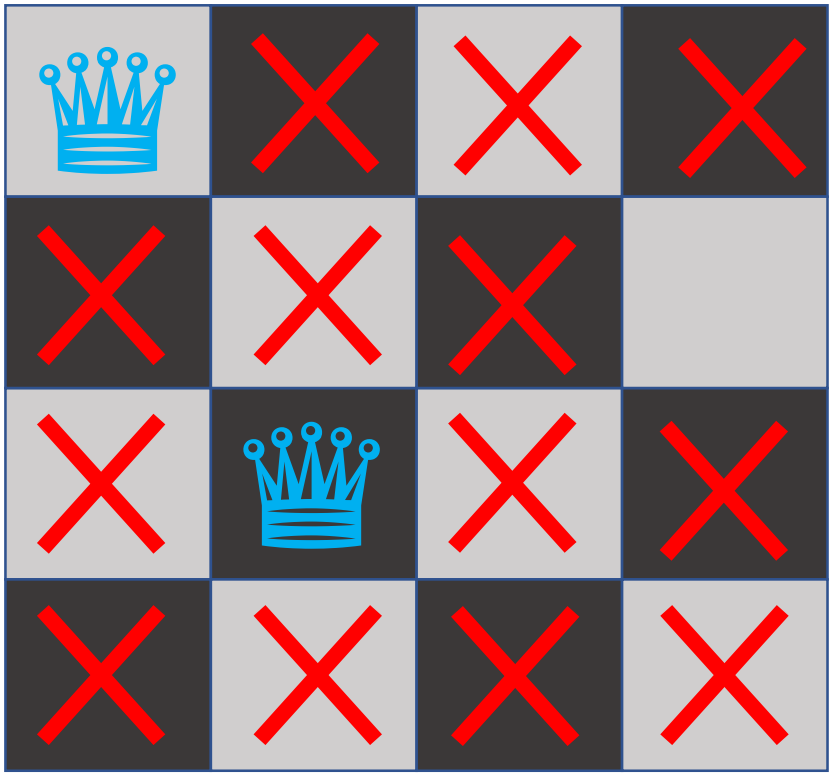
- The jump-back no-good $J(p)$ is a subset of branching constraints that led to a dead-end leaf node not being satifiable

- Therefore, the negated constraint $\neg J(p)$ is an implied constraint of the network and can be added

- This should help identifying and recording no-goods earlier in the search tree and avoid going into subtrees that have no chance of success

No-good recording

- Example: $\mathrm{p} = \{r_1 = 1, r_2 = 3\}$ is a no-good of the 4-queens problem, because $D(r_3) = \emptyset$
- The two values removed in this step are $(r_3, 2)$ and $(r_3, 4)$
- The eliminating explanations are
  $e(r_3 \neq 2) = \{r_1 = 1, r_2 = 3\}$
  $e(r_3 \neq 4) = \{r_1 = 1, r_2 = 3\}$
- The jump-back no-good for this node is therefore
  $J(p) = \{r_1 = 1, r_2 = 3\}$
- We can therefore add the implied constraint to our network
  $\neg \mathrm{J}(p) = r_1 \neq 1 \vee r_2 \neq 3$



Back-jumping

- Chronological backtracking is appending and removing branching constraints in order, i.e. when a path from the root of the search tree to a leaf $\{b_1, \dots, b_j\}$ is a dead-end, then the leaf node $\{b_j\}$ is removed and the search continued

- Removing any other node and "jump" to a different location in the search tree might be advantageous

- The jump-back no-good $J(p) \subset \{b_1, \dots, b_j\}$ comprises the constraints that "caused" the no-good and might not contain b_j
- The idea of back-jumping is to move up several levels in the search tree and continue the search at the node $\{b_1, \dots, b_i\}$ where b_i is the last constraint contained in $J(p)$

Randomisation and re-starts

- Backtracking search can get trapped in sub-trees that do not lead to any solution
- If these sub-trees are encountered early in the search, the search algorithm (which still is very likely exponential in its worst-case) might get trapped and will not result in any solution within a reasonable timeframe
- We can observe that different ordering heuristics sometimes lead to very different runtime behaviour, where some lead to a solution very quickly while others do not terminate in time
- In order to address this issue we could randomise the variable ordering heuristic and re-start the search after some time

Optimisation

- So far we have focused on the constraints and searched for a solution that satisfies all these constraints
- Sometimes we are solving an optimisation problem rather than a satisfiability problem, i.e. our ultimate goal is to find the minimum of an objective function

$$f[x_1, \dots, x_n] \rightarrow min$$

- In this case we need to traverse the whole search tree in order to make sure we have considered all satisfiable solutions and make sure that the one we select is the one that minimises the objective function

Branch-and-bound

- Sometimes we are able to identify a lower bound g based on the domains, i.e.

$$\forall x_1 \in D(x_1), \dots, x_n \in D(x_n): g[D(x_1), \dots, D(x_n)] \leq f[x_1, \dots, x_n]$$

- Then we can make branching decisions based on this heuristic

- If we already recorded a solution where the objective function evaluates to f^*, then we do not need to branch into sub-trees that cannot yield a solution, i.e. where $g > f^*$

- Further to that, g can be used to guide branching decisions for example by exploring sub-trees where g is smallest first

- As with all heuristics, there is no guarantee that this makes a difference

Summary

- The performance of a constraint programming problem crucially depends on the constraint propagation
- Integrating backtracking search with constraint propagation is therefore crucial
- Many heuristics can be tried to influence the branching strategy, which often has significant impact on performance
- The selection of the right model for a given problem is the most effective way to achieve good performance, in particular breaking symmetries is very important as it will prune symmetric sub-trees

Thank you for your attention!