Artificial Intelligence

Roman Barták
Department of Theoretical Computer Science and Mathematical Logic

Introduction

- So far we assumed the world states as blackboxes (no internal structure was assumed) accessed via:
  - successor function
  - goal test
  - heuristic function (distance to goal)

- **Today** we will look inside the states:
  - representing problems as constraint satisfaction problems (CSPs)
    - state has a structure that can be exploited during problem solving
  - general constraint satisfaction techniques
    - depth-first search combined with inference via constraint propagation

Sudoku?

- Logic-based puzzle, whose goal is to enter digits 1-9 in cells of 9×9 table in such a way, that no digit appears twice or more in every row, column, and 3×3 sub-grid.

A bit of history

1979: first published in New York under the name „Number Place“
1986: became popular in Japan
Sudoku – from Japanese „Sudji wa dokushin ni kagiru“ „the numbers must be single“ or „the numbers must occur once“
2005: became popular in the western world

How to find out which digit to fill in?

- Use information that each digit appears exactly once in each row and column.

What if this is not enough?

- Look at columns
  - or combine information from rows and columns

Solving Sudoku

If neither rows nor columns provide enough information, we can note allowed digits in each cell.

The position of a digit can be inferred from positions of other digits and restrictions of Sudoku that each digit appears one in a column (row, sub-grid).

Sudoku – One More Step

• Find colours for countries (red, blue, green) such that no neighbours are coloured by the same colour.

   – **Constraint model**
     - variables: \{WA, NT, Q, NSW, V, SA, T\}
     - superdomain: \{r, b, g\}
     - constraints: WA ≠ NT, WA ≠ SA ...

   – Can also be represented as a **constraint network** (nodes = variables, arc=constraints)

   • **Problem solution**
     \[WA = r, NT = g, Q = r, NSW = g, V = r, SA = b, T = g\]
**Terminology**

**State** is a partial assignment of values to variables.

A **consistent state** is an assignment that does not violate any constraint.

A **complete state** is a state where each variable is assigned to some value.

The **goal** is a complete consistent state.

Sometimes, there is an **objective function** defined over the variables that evaluates the goal states by assigning them real numbers. Then we are looking for an **optimal goal state**, that is, a goal state with the minimal (or maximal) value of the objective function.

**How to solve a CSP?**

- So far we know various **search algorithms**, so we can apply them to CSPs too.
  - the **initial state**: an empty assignment
  - **applicable actions**: assigning a value to a certain variable such that no constraint is violated
  - the **goal**: a complete consistent assignment

**Some notes:**

- the same solving approach for all CSPs
- the goal state is always at depth n, where n is the number of variables
  - We can use DFS even with checking cycles!
- the order of actions is not important to reach the goal (a CSP is a **commutative problem**)
  - \langle \text{WA=r, NT=g} \rangle \text{ is the same as } \langle \text{NT=g, WA=r} \rangle
  - we can also use local search techniques
- it is possible to use different branching schemes to solve CSPs, for example domain splitting

**Backtracking**

The core uninformed algorithm to solve a CSP:

- gradually assigns values to variables
- if no value can be assigned to a variable then goes back to the previous variable and tries an alternative value for that variable

```python
function Backtracking-Search( cap) returns a solution, or failure
return Recursive-Backtracking({}, cap)
function Recursive-Backtracking( assignment, cap) returns a solution, or failure
if assignment is complete then return assignment
var ← Select-Unassigned-Variable( Variables[cap], assignment, cap)
for each value in Order-Domain-Values(var, assignment, cap) do
  if value is consistent with assignment according to Constraints[cap] then
    add \{ var = value \} to assignment
    result ← Recursive-Backtracking(assignment, cap)
    if result \neq failure then return result
    remove \{ var = value \} from assignment
  return failure
```

**Backtracking - an example**

This is in fact **depth-first search**. The real **backtracking** keeps only the states on a single path, more precisely, it keeps the partial assignment of variables!
How to influence efficiency of search?

- assigning the “right” values
  - this is usually problem dependent
- the choice of variable for assignment
  - at the end, we need to assign values to all the variables, but
    the order of variables influences the size of the search tree
  - problem independent heuristics (such as first-fail)
- early detection of “wrong” branches
  - deducing extra information (for example via constraint propagation)
- exploiting a problem structure
  - some problems can be solved using backtrack-free search
    (for example tree-structured CSPs)

Backtracking – variable ordering

- The most restricted variable first
  - a variable with the smallest number of actions
  - i.e. variable with the smallest current domain
  - so called dom heuristic

- The most constrained variable first
  - participates in the largest number of constraints
  - so called deg heuristic
  - frequently used when dom heuristic does not select a single variable
    (dom+deg heuristic)

These are instances of the fail-first principle – assign first a variable whose assignment will probably lead to a failure.

Backtracking – value ordering

- When selecting a value for the variable, we prefer values probably belonging to a solution – a succeed-first principle.
- How to recognize such a value?
  - for example a value that restricts least the other variables
    (keeps the largest flexibility in the problem)
  - the value can also be found by relaxing the problem, finding the solution of the relaxed problem, and using values from this solution (recall construction of heuristics)
  - finding the generally best value is frequently computationally expensive and hence problem-dependent heuristics are usually preferred

Forward checking

- Can we guess in advance that a given path does not lead to the goal?
  - After assigning a value to the variable we can check the future constraints – constraints between the current variable and not-yet instantiated variables – forward checking.
  - constraint check = remove values violating the constraint
• Can we exploit the constraints even more?

– we can check the constraints even between the future variables;
– then we can find that blue cannot be used for NT and SA and this is the only colour in their domains
– because the assigned value is propagated through the constraints, this method is called constraint propagation or look ahead
– this is implemented via maintaining consistency of constraints

Arc consistency

• each constraint is used to filter out values that violate the constraint
• usually implemented in a directional way – remove values from the domain of X that have no support (a consistent value) in the domain of Y for the binary constraint \((X,Y)\);
• of course do it also in the reverse direction

Stronger consistency

• We can generally define \(k\)-consistency, as the consistency check where for a consistent assignment of \((k-1)\) variables we require a consistent value in one more given variable.
  – arc consistency (AC) = 2-consistency
  – path consistency (PC) = 3-consistency

– if the problem is \(i\)-consistent \(\forall i=1,\ldots,n\) \((n\) is the number of variables\), then we can solve it in a backtrack-free way.
  – DFS can always find a value consistent with the assignment of previous variables
– Unfortunately, the time complexity of \(k\)-consistency is exponential in \(k\).
Global constraints

- Instead of stronger consistency techniques (expensive) usually **global constraints** are used – a global constraint encapsulates a sub-problem with a specific structure that can be exploited in the ad-doc domain filtering procedure.

  **Example:**
  
  global constraint all_different({X_1, ..., X_k})
  
  - encapsulates a set of binary inequalities X_1 = X_2, X_1 ≠ X_3, ..., X_{k-1} ≠ X_k
  - all_different({X_1, ..., X_k}) = {{d_1, ..., d_k} | \( \forall i \in \mathbb{D}_i, \forall i \neq j \), d_i ≠ d_j}
  - the filtering procedure is based on matching in bipartite graphs

  Bipartite graph:
  - variables on one side, values on the other side
  - arcs connect a variable with values in its domain

Final notes

- **A declarative approach** to problem solving
  - construct a **model** (variables, domains, constraints)
  - use a **general constraint solver**

- **Possible extensions**
  - **optimisation problems**
    - applying branch-and-bound
  - **soft constraints**
    - the constraint describes a preference rather than a restriction
    - optimisation methods are applied there

- **Other solving approaches**
  - **local search** (the path to the goal is not important)
  - integer programming (for linear constraints)

Application areas

**Bioinformatics**
- DNA sequencing
- determining 3D structures of proteins

**Planning**
- autonomous action planning for space probes
  (Deep Space 1)

**Manufacturing scheduling**
- savings after applying CSP:
  US$ 0.2-1 million per day

More information

- **Constraint Solvers**
  - SICStus Prolog (available in labs)
  - ECLiPSe (Open Source, http://eclipse.crosscoreop.com/)
  - GECODE (Open Source C++, http://www.gecode.org/)
  - Choco (Open Source Java, http://www.emn.fr/z-info/choco-solver/)
  - ...

- **Course Constraint Programming**
  - also taught in English