Dipartimento di elettronica e informazione
Politecnico di Milano

Artificial Intelligence 2010-11

8. Constraint Satisfaction Problems (CSPs)

8.1 Main points

State space search is:

- either uninformed
- or informed by *domain-dependent* heuristics.

(Domain-dependent heuristics are often called *problem-dependent*, but this term is inaccurate: the heuristic function depends on the structure of the state space, but does *not* depend on specific problems formulated in a given state-space.)

CSPs allow one to perform search guided by *domain-independent* heuristics, based on the formal structure of such problems. This implies that useful heuristics can be defined a-priori as part of a CSP solver. Moreover, CSPs are very common in practical applications. Examples:

- Sudoku
- map colouring
- timetables
- job scheduling.

8.2 Definition

A (discrete) CSP is defined by:

- a finite set $X = \{x_1, ..., x_n\}$ of *variables*
- for every variable x_i, a set D_i of possible *values*, called the *domain* of x_i
- a finite set $C = \{c_1, ..., c_m\}$ of *constraints* on possible assignments of values to variables (see below).

An *assignment* is a possibly empty, functional set of ordered pairs $\langle x_i,v_j\rangle$, called *bindings*, where $x_i \in X$ and $v_j \in D_i$ ("functional" means that every variable appears in at most one binding).

Assignment *A'* *extends* assignment *A* if every binding in *A* also belongs to *A'*: $A \subseteq A'$.

A *complete assignment* is an assignment that binds every variable.

A *consistent assignment* is an assignment that satisfies all constraints (in general the empty assignment is assumed to be consistent).

An *initial assignment* is any consistent assignment (often the initial assignment is empty).

A *solution* of a CSP is a consistent and complete assignment that extends the initial assignment.

Solutions may be ordered according to an *objective function*.

Example 1: 2-*Sudoku*

1	2	3	2
2	2	3	1
2	3	2	**1**
1	3	2	**3**

1	4	3	2
2	3	1	4
3	2	4	**1**
4	1	2	**3**

Variables: $x_{11}, x_{12}, ..., x_{44}$

Domains: $D_{ij} = \{1,2,3,4\}$

Constraints: $x_{ij} \neq x_{ik}$,

$x_{ij} \neq x_{kj}$,

$x_{ij} \neq x_{kh}$ if both variables belong to the same square box.

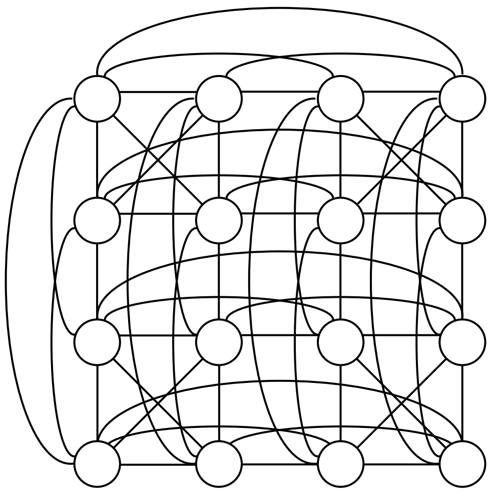
All constraints are *binary*, in the sense that each of them relates the values of two variables.

An initial assignment (see above): $\{x_{11} = 1, x_{21} = 2, x_{34}= 1, x_{44} = 3\}$.

A solution: see above.

The constraint graph

Every node of the graph represents a 2-Sudoku cell. Two nodes are connected by an edge if and only if the corresponding cells cannot contain the same value.



The *degree* of a variable is the number of (binary) constraints it has with other variables. Here every variable has degree 7.

Example 2: Timetable

5 courses, each with two two-hour lessons.

5 days, three two-hour time slots per day.

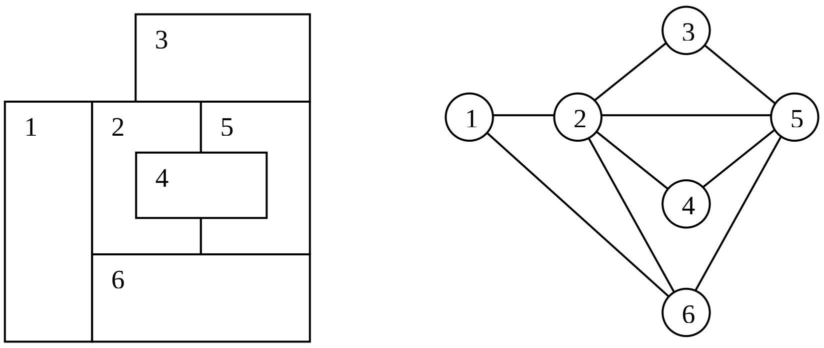
Variables: the lessons, not the time slots! (every lesson must have an associated time slot, a time slot may be empty)

Domains: the time slots

Constraints: the two lessons of a course must be in different days (*hard constraint*); minimise the costs of violating various *soft constraints* (e.g., professor X does not like to lecture on Fridays; the lesson of two difficult courses should not be adjacent; and so on).

Example 3: Map colouring

It is well known that every map can be coloured with 4 colours, so that no adjacent areas are coloured with the same colour. For example:



Variables: the 6 areas. Domains (the same for every variable): four colours (A, B, C, D).

8.3 Constraints

Constraints can be regarded as Boolean functions. *Binary constraints* take as input two bound variables, and return *true* if the respective values satisfy the constraint (and *false* otherwise). We can also consider *unary constraints*: for example, one may want to exclude a colour from the domain of a variable. In principle, one may have constraints of every *arity* 1, 2, 3, and so on. It can be shown, however, that every CSP with constraints of arity higher than 2 can be transformed into another CSP which is equivalent to the original one and has only unary and binary constraints.

8.4 CSPs and state spaces

Every CSP can be viewed as a state space problem (SSP):

– set of states: all consistent assignments
– actions: adding a new binding (i.e., variable-value pair) to a consistent assignment, so that the result is again a consistent assignment
– action costs: irrelevant, so they can be assumed to be identically equal to 1
– initial state: any state (often the empty assignment)
– goal set: the set of all CSP solutions (in the sense specified in sections 8.2)
– a solution: ??? (the sequence of actions is irrelevant, all relevant information is in the goal set – but be careful with the terminology, what is called a goal in SSPs, is called a solution in CSPs!).

If we view a CSP as a case of SPP, the resulting search tree has certain special features:

– if there are *n* variables and the initial state contains *k* variable-value pairs, then all nodes representing solutions are at depth *n*–*k* (if the depth of the search tree is less then *n*–*k*, then there is no solution to the problem)
– moreover, *every* node at depth *n*–*k* represents a solution, because:
  – every node of the search tree represents a consistent extension of the initial assignment
  – every node at level *n*–*k* represents a complete assignment.

Therefore, DF search is complete, because the search tree is finite, and optimal, because all solutions are at the same level of the search tree; it is memory efficient, but it can be time inefficient for large problems.

Indeed, the branching factor can be huge. For example, if we consider the 2-Sudoku problem of Example 1 we see that the root node has $12 \cdot 4 = 48$ successors, because there are 12 unbound variables, each with 4 possible values. We can easily reduce the number of successors by removing from the domain of every unbound variable those values that would violate a constraint: this operation is an instance of *constraint propagation*. (Note that constraint propagation is a case of *inference*, not of *search*.) This way the number of successors of the root is reduced to 26:

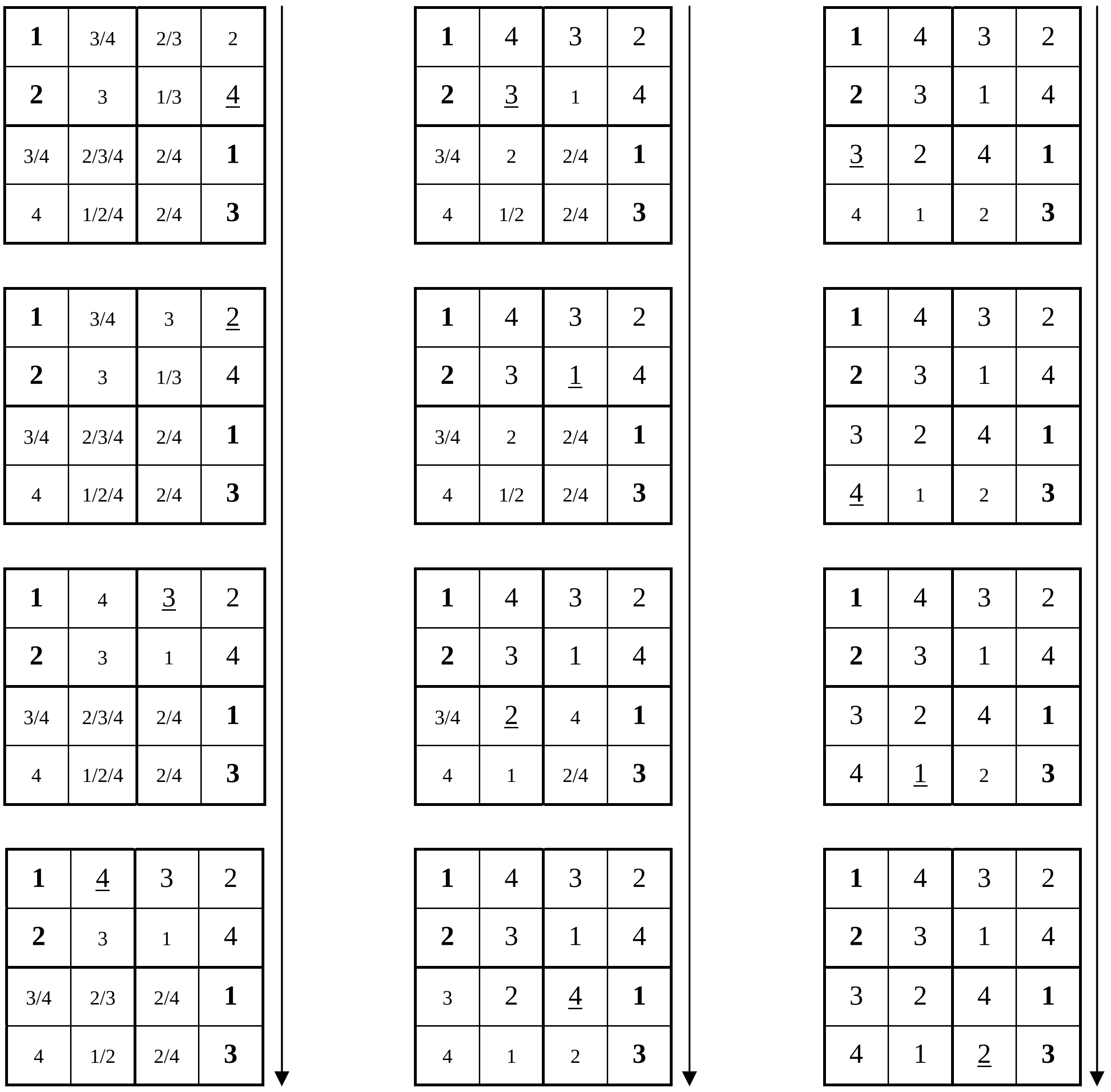
1	3/4	2/3/4	2/4
2	3/4	1/3/4	4
3/4	2/3/4	2/4	**1**
4	1/2/4	2/4	**3**

An important consideration, now, is that in CSPs actions are *commutative*, in the sense that starting from any assignment and assigning value *v* to variable *x* and then value *v'* to variable *x'*, or doing the opposite (i.e., assigning value *v'* to *x'* first, and then *v* to *x*), gives the same result. This makes the order in which actions are performed completely irrelevant to reaching a specific solution.

We can try to exploit this fact to find a solution more efficiently by DF search. That is, we shall try to order the variables to which we assign a value, and/or the values to be assigned to variables, so that we minimise failures, and reach level $n–k$ in the search tree as quickly as possible. (Do not forget that all and only the nodes at level $n–k$ in the search tree represent solutions, and thus we are only interested in reaching level $n–k$ in the search tree, irrespective of the order of actions.)

The above 2-Sudoku puzzle can be solved easily by applying the MRV (*Minimum Remaining Values*) heuristics, which orders variables according to the size of the their domains, after removing from the domain of every unbound variable those values that would violate a constraint (constraint propagation). MRV is a domain-independent heuristics, because it only depends on the structure of the problem (as expressed by the variables, their domains, and the constraints). In the current example, applying MRV makes search superfluous (i.e., to each variable, no alternative bindings of variables have to be explored).

In the scheme below, we start from the array reported above. All variables in the array are scanned from left to right and from the top to the bottom. The first variable that is found whose domain is reduced to a single value is bound to that value (underlined). Then the domains of all variables are updated (small numbers).



1	3/4	2/3	2
2	3	1/3	$\underline{4}$
3/4	2/3/4	2/4	**1**
4	1/2/4	2/4	**3**

1	3/4	3	$\underline{2}$
2	3	1/3	4
3/4	2/3/4	2/4	**1**
4	1/2/4	2/4	**3**

1	4	$\underline{3}$	2
2	3	1	4
3/4	2/3/4	2/4	**1**
4	1/2/4	2/4	**3**

1	$\underline{4}$	3	2
2	3	1	4
3/4	2/3	2/4	**1**
4	1/2	2/4	**3**

1	4	3	2
2	$\underline{3}$	1	4
3/4	2	2/4	**1**
4	1/2	2/4	**3**

1	4	3	2
2	3	$\underline{1}$	4
3/4	2	2/4	**1**
4	1/2	2/4	**3**

1	4	3	2
2	3	1	4
3/4	$\underline{2}$	4	**1**
4	1	2/4	**3**

1	4	3	2
2	3	1	4
3	2	$\underline{4}$	**1**
4	1	2	**3**

1	4	3	2
2	3	1	4
$\underline{3}$	2	4	**1**
4	1	2	**3**

1	4	3	2
2	3	1	4
3	2	4	**1**
$\underline{4}$	1	2	**3**

1	4	3	2
2	3	1	4
3	2	4	**1**
4	$\underline{1}$	2	**3**

1	4	3	2
2	3	1	4
3	2	4	**1**
4	1	$\underline{2}$	**3**

8.5 Domain independent heuristics

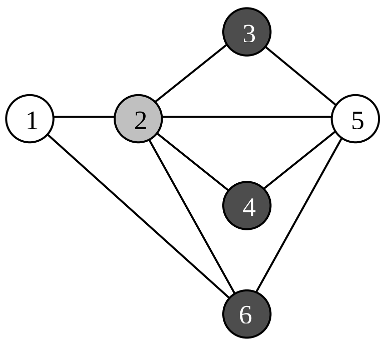
In general, domain independent heuristics involve:

- ordering *variables*: which variable should be assigned a value first?
- ordering *values*: which value should be assigned to a variable first?

Example: 3-colouring of the graph (colours: A, B, C).

Pure BT (i.e., DF with back-tracking), without special orderings

```
1 2 3 4 5 6
– – – – – –

        1 2 3 4 5 6
        A– – – – –

                1 2 3 4 5 6
                AB – – – –

                        1 2 3 4 5 6
                        ABA – – –

                                1 2 3 4 5 6
                                ABAA – –

                                        1 2 3 4 5 6
                                        ABAAC –
                                                fail: no possible value for 6
                                        fail: no possible value for 5

                                1 2 3 4 5 6
                                ABAC – –
                                        fail: no possible value for 5
                                fail: no possible value for 4

                        1 2 3 4 5 6
                        ABC – – –

                                1 2 3 4 5 6
                                ABCA – –
                                        fail: no possible value for 5

                                1 2 3 4 5 6
                                ABCC – –

                                        1 2 3 4 5 6
                                        ABCCA –

                                                1 2 3 4 5 6
                                                ABCCAC
```



Static ordering of variables

Failures can be minimized by adopting suitable heuristics. The MD (*Maximum Degree*) heuristics involves a *static re-ordering of variables* by *decreasing degree*.

BT + MD:

2 5 6 1 3 4
– – – – – –

2 5 6 1 3 4
A– – – – –

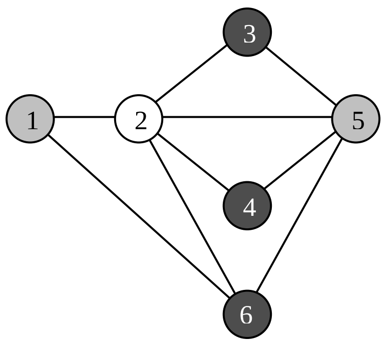
2 5 6 1 3 4
A**B** – – – –

2 5 6 1 3 4
AB**C** – – –

2 5 6 1 3 4
ABC**B** – –

2 5 6 1 3 4
ABCB**C** –

2 5 6 1 3 4
ABCBC**C**



This heuristics cannot be applied to problems like Sudoku, where the degree is the same for every variable.

Dynamic ordering of variables

The *Minimum Remaining Values* (MRV) heuristic orders the variables according to increasing cardinalities of the updated domains. Note: if a variable has zero remaining values, the path fails immediately. This heuristic allowed us to solve the 2-Sudoku puzzle without failures. MRV can also be applied to our map-colouring problem:

BT + MRV (remaining values are in *italics*):

1	2	3	4	5	6
ABC	*ABC*	*ABC*	*ABC*	*ABC*	*ABC*

1	2	6	3	4	5
A	*BC*	*BC*	*ABC*	*ABC*	*ABC*

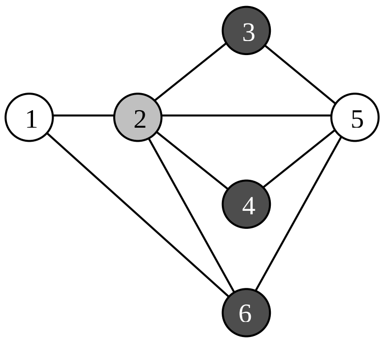
1	2	6	3	4	5
A	**B**	*C*	*AC*	*AC*	*AC*

1	2	6	5	3	4
A	B	**C**	*A*	*AC*	*AC*

1	2	6	5	3	4
A	B	C	**A**	*C*	*C*

1	2	6	5	3	4
A	B	C	A	**C**	*C*

1	2	6	5	3	4
A	B	C	A	C	**C**



Dynamic ordering of values

It may be useful to *order the values* to be assigned to the selected variable according to the *Least Constraining Value* heuristics (LCV): prefer the assignments that constrain less the remaining values of the other variables. In such a way, the probability of reaching a solution is maximized.

Consider for example the previous map, with 3 colours (A, B, C) and the following domains:

- areas 1, 5, and 6: {A, B};
- areas 2, 3, and 4: {A, C}.

BT + LCV (number of areas constrained by each value are in parentheses);

1	2	3	4	5	6
BA	*AC*	*AC*	*AC*	*AB*	*AB*
(12)					

1	2	3	4	5	6
B	*CA*	*AC*	*AC*	*AB*	*AB*
	(24)				

1	2	3	4	5	6
B	**C**	*A*	*A*	*AB*	*AB*

1	2	3	4	5	6
B	C	**A**	*A*	*B*	*AB*

1	2	3	4	5	6
B	C	A	**A**	*B*	*AB*

1	2	3	4	5	6
B	C	A	A	**B**	*A*

1	2	3	4	5	6
B	C	A	A	B	**A**