Review of constraint satisfaction problems (CSPs)

We have:

- A set of n *variables* $V_1, V_2, \ldots, V_n$.
- For each V_i a *domain* D_i specifying the values that V_i can take.
- A set of m *constraints* $C_1, C_2, \ldots, C_m$.

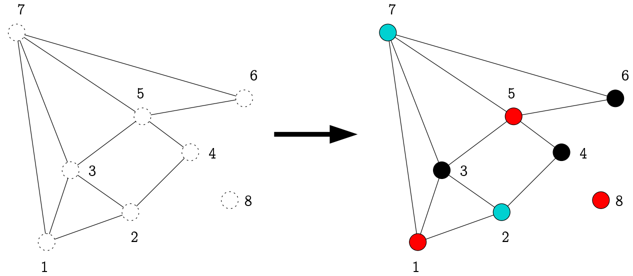
Each constraint C_i involves a set of variables and specifies an *allowable collection of values*.

- A *state* is an assignment of specific values to some or all of the variables.
- An assignment is *consistent* if it violates no constraints.
- An assignment is *complete* if it gives a value to every variable.

A *solution* is a consistent and complete assignment.

Example

We will use the problem of *colouring the nodes of a graph* as a running example.



Each node corresponds to a *variable*. We have three colours and directly connected nodes should have different colours.

Caution required: later on, edges will have a different meaning.

Example

This translates easily to a CSP formulation:

- The variables are the nodes

$$V_i = \text{node } i$$

- The domain for each variable contains the values black, red and cyan

$$D_i = \{B, R, C\}$$

- The constraints enforce the idea that directly connected nodes must have different colours. For example, for variables V_1 and V_2 the constraints specify

$$(B, R), (B, C), (R, B), (R, C), (C, B), (C, R)$$

- Variable V_8 is unconstrained.

Different kinds of CSP

This is an example of the simplest kind of CSP: it is *discrete* with *finite domains*. We will concentrate on these.

We will also concentrate on *binary constraints*; that is, constraints between *pairs of variables*.

- Constraints on single variables—*unary constraints*—can be handled by adjusting the variable's domain. For example, if we don't want V_i to be *red*, then we just remove that possibility from D_i.
- *Higher-order constraints* applying to three or more variables can certainly be considered, but...
- ...when dealing with finite domains they can always be converted to sets of binary constraints by introducing extra *auxiliary variables*.

How does that work?

The state-variable representation

Another planning language: the *state-variable representation*.

Things of interest such as people, places, objects *etc* are divided into *domains*:

$D_1 = \{\texttt{climber1}, \texttt{climber2}\}$
$D_2 = \{\texttt{home}, \texttt{jokeShop}, \texttt{hardwareStore}, \texttt{pavement}, \texttt{spire}, \texttt{hospital}\}$
$D_3 = \{\texttt{rope}, \texttt{inflatableGorilla}\}$

Part of the specification of a planning problem involves stating which domain a particular item is in. For example

$$D_1(\texttt{climber1})$$

and so on.

Relations and functions have arguments chosen from unions of these domains.

$$\texttt{above}(x, y) \subseteq \mathcal{D}_1^{\text{above}} \times \mathcal{D}_2^{\text{above}}$$

is a relation. The $\mathcal{D}_i^{\text{above}}$ are unions of one or more D_i.

The state-variable representation

The relation above is in fact a *rigid relation (RR)*, as it is unchanging: it does not depend upon *state*. (Remember *fluents* in situation calculus?)

Similarly, we have *functions*

$$\texttt{at}(x_1, s) : \mathcal{D}_1^{\text{at}} \times S \to \mathcal{D}^{\text{at}}.$$

Here, $\texttt{at}(x, s)$ is a *state-variable*. The domain $\mathcal{D}_1^{\text{at}}$ and range $\mathcal{D}^{\text{at}}$ are unions of one or more D_i. In general these can have multiple parameters

$$\texttt{sv}(x_1, \ldots, x_n, s) : \mathcal{D}_1^{\text{sv}} \times \cdots \times \mathcal{D}_n^{\text{sv}} \times S \to \mathcal{D}^{\text{sv}}.$$

A state-variable denotes assertions such as

$$\texttt{at}(\texttt{gorilla}, s) = \texttt{jokeShop}$$

where s denotes a *state* and the set S of all states will be defined later.

The state variable allows things such as locations to change—again, much like *fluents* in the situation calculus.

Variables appearing in relations and functions are considered to be *typed*.

The state-variable representation

Note:

- For properties such as a *location* a function might be considerably more suitable than a relation.
- For locations, everything has to be *somewhere* and it can only be in *one place at a time*.

So a function is perfect and immediately solves some of the problems seen earlier.

The state-variable representation

Actions as usual, have a *name*, a *set of preconditions* and a *set of effects*.

- *Names* are unique, and followed by a list of variables involved in the action.
- *Preconditions* are expressions involving state variables and relations.
- *Effects* are assignments to state variables.

For example:

$\texttt{buy}(x, y, l)$	
Preconditions	$\texttt{at}(x, s) = l$ $\texttt{sells}(l, y)$ $\texttt{has}(y, s) = l$
Effects	$\texttt{has}(y, s) = x$

The state-variable representation

Goals are sets of *expressions* involving *state variables*.

For example:

Goal:
$\texttt{at}(\texttt{climber}, s) = \text{home}$
$\texttt{has}(\texttt{rope}, s) = \texttt{climber}$
$\texttt{at}(\texttt{gorilla}, s) = \texttt{spire}$

From now on we will generally suppress the state s when writing state variables.

The state-variable representation

We can essentially regard a *state* as just a statement of what values the state variables take at a given time.

Formally:

- For each state variable $\texttt{sv}$ we can consider all ground instances such as—$\texttt{sv}(\texttt{climber}, \texttt{rope})$—with arguments that are *consistent* with the *rigid relations*.

  Define X to be the set of all such ground instances.
- A state s is then just a set

$$s = \{(v = c) | v \in X\}$$

  where c is in the range of v.

This allows us to define the *effect of an action*.

A planning problem also needs a *start state* s_0, which can be defined in this way.

The state-variable representation

Considering all the *ground actions consistent with the rigid relations*:

- An action is *applicable in s* if all expressions v=c appearing in the set of preconditions also appear in s.

Finally, there is a function γ that maps a state and an action to a new state

$$\gamma(s, a) = s'$$

Specifically, we have

$$\gamma(s, a) = \{(v = c) | v \in X\}$$

where either c is specified in an effect of a, or otherwise $v = c$ is a member of s.

Note: the definition of γ implicitly solves the *frame problem.*

The state-variable representation

A *solution* to a planning problem is a sequence $(a_0, a_1, \dots, a_n)$ of actions such that...

- a_0 is applicable in s_0 and for each i, a_i is applicable in $s_i = \gamma(s_{i-1}, a_{i-1})$.
- For each goal g we have

$$g \in \gamma(s_n, a_n).$$

What we need now is a method for *transforming* a problem described in this language into a CSP.

We'll once again do this for a fixed upper limit T on the number of steps in the plan.

Converting to a CSP

Step 1: encode *actions* as *CSP variables*.

For each time step t where $0 \leq t \leq T - 1$, the CSP has a variable

$$\texttt{action}^t$$

with domain

$$D^{\texttt{action}^t} = \{a | a \text{ is the ground instance of an action}\} \cup \{\text{none}\}$$

Example: at some point in searching for a plan we might attempt to find the solution to the corresponding CSP involving

$$\texttt{action}^5 = \texttt{attach}(\texttt{inflatableGorilla}, \texttt{spire})$$

WARNING: be careful in what follows to distinguish between *state variables, actions etc* in the planning problem and *variables* in the CSP.

Converting to a CSP

Step 2: encode *ground state variables* as *CSP variables*, with a complete copy of all the state variables *for each time step*.

So, for each t where $0 \leq t \leq T$ we have a CSP variable

$$\texttt{sv}_i^t(c_1, \ldots, c_n)$$

with domain $\mathcal{D}^{\texttt{sv}_i}$. (That is, the *domain* of the CSP variable is the *range* of the state variable.)

Example: at some point in searching for a plan we might attempt to find the solution to the corresponding CSP involving

$$\texttt{location}^9(\texttt{climber1}) = \texttt{hospital}.$$

Converting to a CSP

Step 3: encode the *preconditions for actions in the planning problem* as *constraints in the CSP problem*.

For each time step t and for each ground action $a(c_1, \ldots, c_n)$ with arguments *consistent with the rigid relations in its preconditions*:

For a precondition of the form $\texttt{sv}_i = v$ include constraint pairs

$$(\texttt{action}^t = a(c_1, \ldots, c_n),$$
$$\texttt{sv}_i^t = v)$$

Example: consider the action $\texttt{buy}(x, y, l)$ introduced above, and having the preconditions $\texttt{at}(x) = l$, $\texttt{sells}(l, y)$ and $\texttt{has}(y) = l$.

Assume $\texttt{sells}(y, l)$ is only true for

$$l = \texttt{jokeShop}$$

and

$$y = \texttt{inflatableGorilla}$$

(it's a very strange town) so we only consider these values for l and y. Then for each time step t we have the constraints...

Converting to a CSP

$\texttt{action}^t = \texttt{buy}(\texttt{climber1}, \texttt{inflatableGorilla}, \texttt{jokeShop})$ paired with $\texttt{at}^t(\texttt{climber1}) = \texttt{jokeShop}$
$\texttt{action}^t = \texttt{buy}(\texttt{climber1}, \texttt{inflatableGorilla}, \texttt{jokeShop})$ paired with $\texttt{has}^t(\texttt{inflatableGorilla}) = \texttt{jokeShop}$
$\texttt{action}^t = \texttt{buy}(\texttt{climber2}, \texttt{inflatableGorilla}, \texttt{jokeShop})$ paired with $\texttt{at}^t(\texttt{climber2}) = \texttt{jokeShop}$
$\texttt{action}^t = \texttt{buy}(\texttt{climber2}, \texttt{inflatableGorilla}, \texttt{jokeShop})$ paired with $\texttt{has}^t(\texttt{inflatableGorilla}) = \texttt{jokeShop}$
and so on...

Converting to a CSP

Step 4: encode the *effects of actions in the planning problem* as *constraints in the CSP problem.*

For each time step t and for each ground action $a(c_1, \dots, c_n)$ with arguments *consistent with the rigid relations in its preconditions*:

For an effect of the form $\mathtt{sv}_i = v$ include constraint pairs

$$\begin{gathered}(\mathtt{action}^t = a(c_1, \dots, c_n), \\ \mathtt{sv}_i^{t+1} = v)\end{gathered}$$

Example: continuing with the previous example, we will include constraints

$\mathtt{action}^t = \mathtt{buy}(\mathtt{climber1}, \mathtt{inflatableGorilla}, \mathtt{jokeShop})$ paired with $\mathtt{has}^{t+1}(\mathtt{inflatableGorilla}) = \mathtt{climber1}$
$\mathtt{action}^t = \mathtt{buy}(\mathtt{climber2}, \mathtt{inflatableGorilla}, \mathtt{jokeShop})$ paired with $\mathtt{has}^{t+1}(\mathtt{inflatableGorilla}) = \mathtt{climber2}$
and so on...

Converting to a CSP

Step 5: encode the *frame axioms* as *constraints in the CSP problem.*

An action must not change things not appearing in its effects. So:

For:

1. Each time step t.
2. Each ground action $a(c_1, \dots, c_n)$ with arguments *consistent with the rigid relations in its preconditions*.
3. Each $\mathtt{sv}_i$ that *does not appear in the effects of* a, and each $v \in \mathcal{D}^{\mathtt{sv}_i}$

include in the CSP the ternary constraint

$$\begin{gathered}(\mathtt{action}^t = a(c_1, \dots, c_n), \\ \mathtt{sv}_i^t = v, \\ \mathtt{sv}_i^{t+1} = v)\end{gathered}$$

Finding a plan

Finally, having encoded a planning problem into a CSP, we solve the CSP.

The scheme has the following property:

A solution to the planning problem with at most T *steps exists if and only if there is a a solution to the corresponding CSP.*

Assume the CSP has a solution.

Then we can extract a plan simply by looking at the values assigned to the $\mathtt{action}^t$ variables in the solution of the CSP.

It is also the case that:

There is a solution to the planning problem with at most T *steps if and only if there is a solution to the corresponding CSP from which the solution can be extracted in this way.*

For a proof see:

Automated Planning: Theory and Practice

Malik Ghallab, Dana Nau and Paolo Traverso. Morgan Kaufmann 2004.