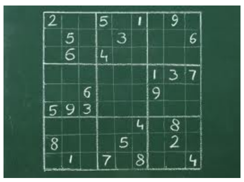
CSPs: examples



- In this module, I will walk through some examples of how to take problems and **model** them as constraint satisfaction problems.

Example: LSAT question

Three sculptures (A, B, C) are to be exhibited in rooms 1, 2 of an art gallery.

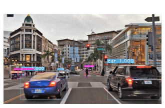
The exhibition must satisfy the following conditions:

- Sculptures A and B cannot be in the same room.
- Sculptures B and C must be in the same room.
- Room 2 can only hold one sculpture.

[demo]

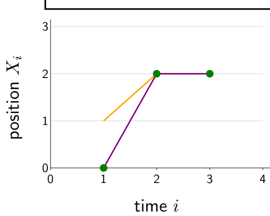
- The LSAT is a standardized test for law school which features questions that are logic puzzles. These can usually be formalized as a constraint satisfaction problem. CSPs offer a formulaic way of tackling these problems which could even be automated (though the hard part for computers is translating the English into the CSP, whereas the hard part for the human is actually solving the CSP!).
- Here is an example of an LSAT question. We will use Javascript inference demo to solve this problem.

Example: object tracking



Problem: object tracking

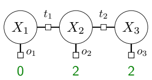
(O) Noisy sensors report positions: $0, 2, 2$.
(T) Objects can't teleport.
What trajectory did the object take?



- In this example, consider the problem of object tracking. For instance, for autonomous driving, objects such as cars and pedestrians must be tracked to know where not to drive.
- Here, at each discrete time step i, we are given some noisy information about where the object might be. For example, this noisy information could be the video frame at time step i. The goal is to answer the question: what trajectory did the object take?
- To simplify, suppose we consider an object moving in 1D and we have a sensor that tells us an approximate position at each time step. We observe 0, 2, 2 from this sensor.

Example: object tracking CSP

Factor graph:



x_1	$o_1(x_1)$
0	2
1	1
2	0

x_2	$o_2(x_2)$
0	0
1	1
2	2

x_3	$o_3(x_3)$
0	0
1	1
2	2

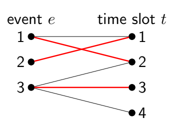
$\|x_i - x_{i+1}\|$	$t_i(x_i, x_{i+1})$
0	2
1	1
2	0

[demo]

- Variables $X_i \in \{0, 1, 2\}$: position of object at time i
- Observation factors $o_i(x_i)$: noisy information compatible with position
- Transition factors $t_i(x_i, x_{i+1})$: object positions can't change too much

- Let's try to model this problem. Always start by defining the variables: these are the quantities which we don't know. In this case, it's the positions of the object at each time step: $X_1, X_2, X_3 \in \{0, 1, 2\}$.
- Now let's think about the factors, which need to capture two things. First, transition factors make sure physics isn't violated (e.g., object positions can't change too much). Second, observation factors make sure the hypothesized positions X_i are compatible with the noisy information. Note that these numbers returned by the factors are just numbers, not necessarily probabilities.
- Having modeled the problem as a factor graph, we can now ask for the maximum weight assignment, which gives us the most likely trajectory for the object.
- Click on the the [track] demo to see the definition of this factor graph as well as the maximum weight assignment, which is [1, 2, 2]. Note that this trajectory is a smoothed version of the observations, which assumes that the first sensor reading was inaccurate.

Example: event scheduling



Problem: Event scheduling

Have E events and T time slots

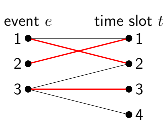
(C1) Each event e must be put in **exactly one** time slot

(C2) Each time slot t can have **at most one** event

(C3) Event e allowed in time slot t only if $(e, t) \in A$

- Scheduling is a broad class of problems for which CSPs are well suited. We will consider a simplified scheduling problem and show that there are sometimes multiple ways to cast the problem as a CSP.
- Consider a simple scheduling problem, where we have E events that we want to schedule into T time slots. There are three types of requirements: (C1) every event must be scheduled into a time slot; (C2) every time slot can have at most one event (zero is possible); and (C3) we are given a fixed set A of (event, time slot) pairs which are allowed.

Example: event scheduling (formulation 1)



Problem: Event scheduling

Have E events and T time slots

(C1) Each event e must be put in **exactly one** time slot

(C2) Each time slot t can have **at most one** event

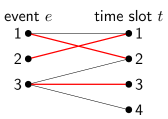
(C3) Event e allowed in time slot t only if $(e, t) \in A$

CSP formulation 1:

- Variables: for each event e, $X_e \in \{1, \dots, T\}$; satisfies (C1)
- Constraints (only one event per time slot): for each pair of events $e \neq e'$, enforce $[X_e \neq X_{e'}]$; satisfies (C2)
- Constraints (only scheduled allowed times): for each event e, enforce $[(e, X_e) \in A]$; satisfies (C3)

- The first formulation is perhaps the more natural one. We make a variable X_e for each event, whose value will be the time slot that the event is scheduled into. Since each variable can only take on one value, we automatically satisfy (C1), the requirement that every event must be put in exactly one time slot.
- However, we need to make sure no two events end up in the same time slot (C2). To do this, we can create a binary constraint between every pair of distinct event variables X_e and X'_e that enforces their values to be different ($X_e \neq X_{e'}$).
- Finally, to deal with the requirement that an event is scheduled only in allowed time slots (C3), we just need to add a unary constraint for each variable saying that the time slot X_e that's chosen for that event is allowed.
- Note that we end up with E variables with domain size T, and $O(E^2)$ binary constraints.

Example: event scheduling (formulation 2)



Problem: Event scheduling

Have E events and T time slots
(C1) Each event e must be put in **exactly one** time slot
(C2) Each time slot t can have **at most one** event
(C3) Event e allowed in time slot t only if $(e,t) \in A$

CSP formulation 2:

- Variables: for each time slot t, $Y_t \in \{1,\dots,E\} \cup \{\emptyset\}$; satisfies (C2)
- Constraints (each event is scheduled exactly once): for each event e, enforce $[Y_t = e$ for exactly one $t]$; satisfies (C1)
- Constraints (only schedule allowed times): for each time slot t, enforce $[Y_t = \emptyset$ or $(Y_t, t) \in A]$; satisfies (C3)

- Alternatively, we can take the perspective of the time slots and ask which event was scheduled in each time slot. So we introduce a variable Y_t for each time slot t which takes on a value equal to one of the events or none ($\emptyset$); this automatically takes care of (C2).
- Unlike the first formulation, we don't get for free the requirement that each event is put in exactly one time slot (C1). To add it, we introduce E constraints, one for each event. Each constraint needs to depend on all T variables and check that the number of time slots t which have event e assigned to that slot ($Y_t = e$) is exactly 1.
- Finally, we add T constraints, one for each time slot t enforcing that if there was an event scheduled there ($Y_t \neq \emptyset$), then it better be allowed according to A.
- With this formulation, we have T variables with domain size $E+1$, and E T-ary constraints. We will show shortly that each T-ary constraints can be converted into $O(T)$ binary constraints with $O(T)$ variables. Therefore, the resulting formulation has T variables with domain size $E+1$, $O(ET)$ variables with domain size 2 and $O(ET)$ binary constraints.
- Which one is better? Since $T \geq E$ is required for the existence of a consistent solution, the first formulation is better.
- But if we were to add another constraint relating adjacent time slots (e.g., the courses assigned two adjacent slots should have topic overlap), then the second formulation would make it easier.

Example: program verification

```
def foo(x, y):
    a = x * x
    b = a + y * y
    c = b - 2 * x * y
    return c
```

Specification: `c >= 0` for all `x` and `y`

CSP formulation:

- Variables: x, y, a, b, c
- Constraints (program statements): $[a = x^2]$, $[b = a + y^2]$, $[c = b - 2xy]$

Note: program (= is assignment), CSP (= is mathematicality equality)

- Constraint (negation of specification): $[c < 0]$

Program satisfies specification iff CSP has no consistent assignment

- In our next example, we consider formal verification of programs. You are probably used to the idea of writing unit tests to check whether a computer program is correct. However, just because your tests pass doesn't mean that your program is correct, and you're never sure if you've covered all the cases. The idea behind formal verification is to write down a **specification**, which you want to verify.
- In this example, we have a Python function `foo` that computes some value `c` based on two inputs `x` and `y`. We want to verify the specification that the return value will always be non-negative for `all` possible inputs. (With some simple algebra, you can see that `foo` actually computes $(x-y)^2$, which is indeed non-negative.)
- We can use CSPs to encode the verification problem as follows. First, we create variables for the inputs and intermediate steps of the Python program.
- Then we add a constraint for each program statement.
- Finally, we add a constraint which is the negation of the specification. This is because solving a CSP only looks for the existence of an assignment. So here we are asking the CSP to look for a counterexample to the specification. If a consistent assignment is found, then we say that the program fails to satisfy the specification. If no consistent assignment is found, then the program satisfies the specification.
- It is important to note that these constraints look like the the assignment statements in Python, but they are mathematically different operations. In Python, "$=$" is the assignment operator and is executed to set the variable on the left-hand-side. In the CSP, "$=$" is the mathematical equality operator that, given a value for the variables on both the left-hand-side and the right-hand-side, returns whether this valid is or not.
- The ramification of this is that while you can only run the Python program forward, the CSP factors have no directionality: they just relate the variables on the left-hand-side to the variables on the right-hand-side. That means the CSP solver can even "work backwards" from the specification (which is a constraint on the final program output).

Summary

- Decide on variables and domains
- Translate each desideratum into a set of factors
- Try to keep CSP small (variables, factors, domains, arities)
- When implementing each factor, think in terms of checking a solution rather than computing the solution

- We have seen a few examples of taking a real-world problem and creating a CSP to solve this problem, which is the process of modeling.
- Generally, you want to first nail down the variables and domains, and make sure that an assignment to these variables provides the result of interest.
- Then we examine the desiderata and convert them into factors. One nice thing about CSPs is that this process can often done in parallel: each desideratum maps onto to a set of factors, which are just thrown into the set of all factors.
- There are sometimes multiple ways of creating a CSP that will do the job, but the different CSPs might differ in terms of computational and memory efficiency. It's generally a good idea to keep the CSP small (though there isn't really any rigorous characterization of smallness that translates directly to computational efficiency).
- Finally, modeling with CSPs requires a different mindset than normal programming, which is most salient in the program verification example. While the factors look like mini-programs, they need to check any given solution rather than computing the right solution. It is the job of the inference algorithm to compute the solution.