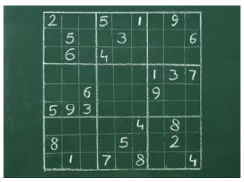
Constraint Satisfaction Problems (CSPs)



Lecture

CSPs: Overview

CSPs: Definitions

CSPs: Examples

- In this module, we will introduce constraint satisfaction problems (CSPs).

Course plan

Search problems
Markov decision processes
Adversarial games

Constraint satisfaction problems
Markov networks
Bayesian networks

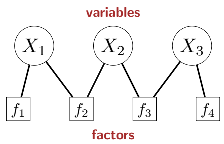
Reflex **States** **Variables** **Logic**

Low-level → High-level

Machine learning

- We started with machine learning and reflex-based models, which simply produce a single output or action (classification or regression).
- Then we looked at state-based models, where we thought in terms of states, actions, and costs/rewards.
- Now we embark on our journey through **variable-based models**, a different modeling language, in which we will think in terms of variables, factors, and weights.

Factor graphs



Objective: find the best assignment of values to the variables

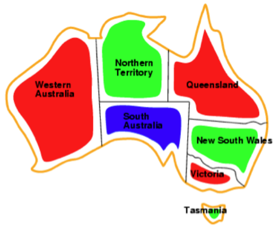
- All variable-based models have an underlying **factor graph**. Before formally defining what a factor graph is, let me first provide some intuition.
- A factor graph contains a set of **variables** (circle nodes), which represent unknown values that we seek to ascertain, and a set of **factors** (square nodes), which determine how the variables are related to one another.
- The objective of a constraint satisfaction problem is to find the best assignment of values to the variables.

Map coloring

Question: how can we color each of the 7 provinces {red,green,blue} so that no two neighboring provinces have the same color?

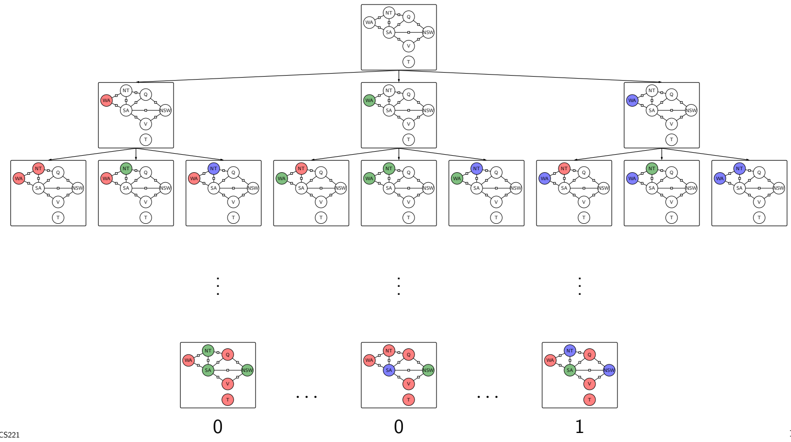
- Let us consider an example problem: map coloring.
- Here's Australia. It has 7 provinces, which might be hard to see, so let's color the provinces. How can we color the provinces with three colors so that no two neighboring provinces have the same color?

Map coloring



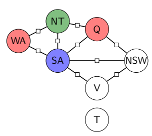
(one possible solution)

- Here is one solution.



- How do we solve this problem algorithmically? Let's use the hammer that we know: casting it as a search problem.
- We start with the state in which no colors are assigned. The possible actions from this state are to color one of the variables (WA) some color.
- In general, each state contains an assignment of colors to a subset of the provinces (a **partial assignment**), and each action corresponds to choosing a color for the next unassigned province.
- The leaves of the search tree are complete assignments, where every province has a color.
- Each leaf is either consistent — i.e., all neighboring provinces have different colors (1), or not (0).
- We then simply return any leaf that is consistent.

As a search problem



- State: partial assignment of colors to provinces
- Action: assign next uncolored province a compatible color

What's missing? There's more problem structure!

- Variable ordering doesn't affect correctness, can optimize
- Variables are interdependent in a local way, can decompose

- This is a fine way to solve this problem, and in general, it shows how powerful search problems are: we don't actually need any new machinery to color Australia. But the question is: can we do better?
- First, **the order in which we assign variables doesn't matter for correctness**. This gives us the flexibility to dynamically choose a better ordering of the variables. That, with a bit of lookahead will allow us to dramatically improve the efficiency over naive tree search.
- Second, it's clear that Tasmania's color can be any of the three colors regardless of the colors on the mainland. This is an instance of **independence**, and later we'll see how to exploit this observation.

Variable-based models

Special cases:

- Constraint satisfaction problems
- Markov networks
- Bayesian networks

Key idea: variables

- Solutions to problems ⇒ assignments to variables (modeling).
- Decisions about variable ordering, etc. chosen by inference.

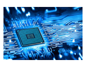
Higher-level modeling language than state-based models

- Variable-based models allow us to capture this additional structure. Variable-based models is an umbrella term that includes constraint satisfaction problems (CSPs), Markov networks, and Bayesian networks.
- Aside: The term graphical models can be used interchangeably with variable-based models, and the term probabilistic graphical models (PGMs) generally encompasses both Markov networks (also called undirected graphical models) and Bayesian networks (directed graphical models).
- The unifying theme is the idea of thinking about solutions to problems as assignments of values to variables (this is the modeling part). All the details about how to find the assignment (in particular, which variables to try first) are delegated to the inference algorithm. So the advantage of using variable-based models over state-based models is that it's making the algorithms do more of the work, freeing up more time for modeling.
- An (imperfect) analogy is programming languages. Solving a problem directly by implementing an ad-hoc program is like using assembly language. Solving a problem using state-based models is like using C. Solving a problem using variable-based models is like using Python. By moving to a higher language, you might forgo some amount of ability to optimize manually, but the advantage is that (i) you can think at a higher level and (ii) there are more opportunities for optimizing automatically.
- Once a new modeling framework become second nature, it is almost as if it was invisible. It's like when you master a language, you can "think" in it without thinking about the language.

Applications

Delivery/routing: how to assign packages to trucks to deliver to customers

Sports scheduling: when to schedule pairs of teams to minimize travel



Formal verification: ensure circuit/program works on all inputs

- Constraint satisfaction problems appear in many applications, most of which involve large-scale logistics, scheduling, and supply-chain management.
- Companies such as Amazon have to figure out how to put packages on vehicles to **deliver** them to customers to minimize cost and meet delivery times promised to the customer. Here, the variables include the assignment of packages to vehicles, and the factors encode travel times and costs. Ride-sharing services such as Uber and Lyft also have to figure out how to best assign drivers to riders. There are all extensions of the classic vehicle routing problem (VRP).
- Each year, the NFL has to make a **schedule** of which teams play what other teams and when. The schedule should minimize travel, fit into TV broadcast slots, be fair across teams, etc. Other scheduling problems involve assigning the courses that are offered one quarter to various classrooms at various time slots.
- A final application is **formal verification** of circuits and programs. Here, the variables are the unknown inputs to a program, and the factors encode the program/circuit execution. Then you can ask the question of whether there exists any program inputs that produce an error or incorrect result.

Roadmap

Modeling

Definitions

Examples

Backtracking (exact) search	**Approximate search**
Dynamic ordering	Beam search
Arc consistency	Local search

- Here's the roadmap for the rest of the modules on CSPs. First we will define constraint satisfaction problems and factor graphs formally, and give a few examples of CSPs.
- We then talk about backtracking search, which solves the problem exactly, though it takes exponential time in the worst case. To speed up search, we can take advantage of the fact that we can assign variables in any order to do dynamic ordering, where we heuristically figure out which variables to assign first. Arc consistency provides an lookahead algorithm called AC-3 to eagerly prune the search space, so that dynamic ordering can be more effective.
- Sometimes, you might not want to wait an exponential amount of time. If a crude solution suffices, one can apply approximate search algorithms. **Beam search** heuristically explores a small fraction of the exponentially-sized search tree, while **local search** takes an initial assignment and iteratively tries to improve it by changing one variable at a time.

Lecture

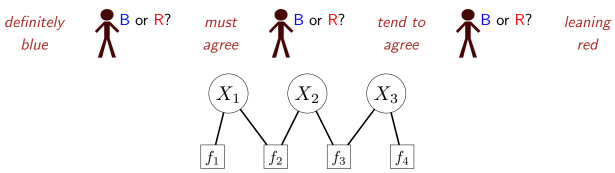
CSPs: Overview

CSPs: Definitions

CSPs: Examples

- In this module, we will formally define constraint satisfaction problems as well as the more general notion of a factor graph.

Factor graph example: voting



x_1	$f_1(x_1)$
R	0
B	1

x_1	x_2	$f_2(x_1, x_2)$
R	R	1
R	B	0
B	R	0
B	B	1

x_2	x_3	$f_3(x_2, x_3)$
R	R	3
R	B	2
B	R	2
B	B	3

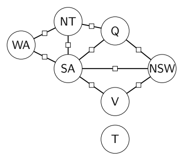
x_3	$f_4(x_3)$
R	2
B	1

$f_1(x_1) = [x_1 = \text{B}]$ $f_2(x_1, x_2) = [x_1 = x_2]$ $f_3(x_2, x_3) = [x_2 = x_3] + 2$ $f_4(x_3) = [x_3 = \text{R}] + 1$

[demo]

- Let us provide an example of a factor graph.
- Suppose there are three people, each of which will vote for a color, red or blue. We know that Person 1 is dead set on blue, while Person 3 is leaning red. Person 1 and Person 2 are close friends and must vote on the same color, while Person 2 and Person 3 are acquaintances who only weakly prefer to have the same color. The question is how each person will vote given their influences on each other?
- We can model this situation as a factor graph consisting of three **variables**, X_1, X_2, X_3, each of which must be assigned red (R) or blue (B).
- We encode each of the constraints/preferences as a **factor**, which assigns a non-negative number based on the assignment to a subset of the variables.
- We can either describe the factor as an explicit table, or via a function (e.g., $[x_1 = x_2]$).
- Notation: we use [condition] to represent the indicator function which is equal to 1 if the condition is true and 0 if not. Normally, this is written $\mathbf{1}$[condition], but we drop the $\mathbf{1}$ for succinctness.

Example: map coloring



Variables:

$X = (\text{WA}, \text{NT}, \text{SA}, \text{Q}, \text{NSW}, \text{V}, \text{T})$

$\text{Domain}_i \in \{\text{R}, \text{G}, \text{B}\}$

Factors:

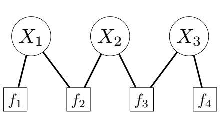
$f_1(X) = [\text{WA} \neq \text{NT}]$

$f_2(X) = [\text{NT} \neq \text{Q}]$

...

- Let's revisit the map coloring example.
- For each province, we have a variable, whose domain is the three colors.
- We have one factor for each pair of neighboring provinces which returns 1 (okay) if the two colors are not equal and 0 otherwise.

Factor graph



Definition: factor graph

Variables:

$X = (X_1, \dots, X_n)$, where $X_i \in \text{Domain}_i$

Factors:

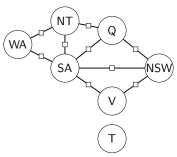
$f_1, \dots, f_m$, with each $f_j(X) \geq 0$

- Now we proceed to the general definition. A factor graph consists of a set of variables and a set of factors: (i) n variables $X_1, \dots, X_n$, which are represented as circular nodes in the graphical notation; and (ii) m factors (also known as potentials) $f_1, \dots, f_m$, which are represented as square nodes in the graphical notation.
- Each variable X_i can take on values in its **domain** Domain_i. Each factor f_j is a function that takes an assignment x to all the variables and returns a non-negative number representing how good that assignment is (from the factor's point of view). Usually, each factor will depend only on a small subset of the variables.

Factors

Definition: scope and arity

Scope of a factor f_j: set of variables it depends on.
Arity of f_j is the number of variables in the scope.
Unary factors (arity 1); **Binary** factors (arity 2).
Constraints are factors that return 0 or 1.



Example: map coloring

Scope of $f_1(X) = [\text{WA} \neq \text{NT}]$ is $\{\text{WA}, \text{NT}\}$
f_1 is a binary constraint

- The key aspect that makes factor graphs useful is that each factor f_j only depends on a subset of variables, called the **scope**.
- The arity of the factors is generally small (think 1 or 2).
- Factors that return 0 or 1 are called constraints. A constraint is satisfied iff a constraint returns 1.

Assignment weights example: voting

x_1	$f_1(x_1)$
R	0
B	1

x_1	x_2	$f_2(x_1, x_2)$
R	R	1
R	B	0
B	R	0
B	B	1

x_2	x_3	$f_3(x_2, x_3)$
R	R	3
R	B	2
B	R	2
B	B	3

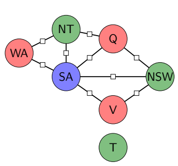
x_3	$f_4(x_3)$
R	2
B	1

x_1	x_2	x_3	Weight
R	R	R	$0 \cdot 1 \cdot 3 \cdot 2 = 0$
R	R	B	$0 \cdot 1 \cdot 2 \cdot 1 = 0$
R	B	R	$0 \cdot 0 \cdot 2 \cdot 2 = 0$
R	B	B	$0 \cdot 0 \cdot 3 \cdot 1 = 0$
B	R	R	$1 \cdot 0 \cdot 3 \cdot 2 = 0$
B	R	B	$1 \cdot 0 \cdot 2 \cdot 1 = 0$
B	B	R	$1 \cdot 1 \cdot 2 \cdot 2 = 4$
B	B	B	$1 \cdot 1 \cdot 3 \cdot 1 = 3$

[demo]

- An **assignment** specifies a value for each variable, which is a candidate solution.
- Recall that the factors specify local interactions between variables.
- For each assignment, we get its weight, which is defined to be the product over each factor evaluated on that assignment.
- Each factor makes a contribution to the weight. Note that any factor has veto power: if it returns zero, then the weight of the entire assignment is irrecoverably zero.
- Think of all the factors chiming in on their opinion of x. We multiply all these opinions together to get the global opinion.
- In this setting, the maximum weight assignment is (B, B, R), which has a weight of 4. This is the assignment we wish to return.

Example: map coloring



Assignment:

$x = \{\text{WA}: \text{R}, \text{NT}: \text{G}, \text{SA}: \text{B}, \text{Q}: \text{R}, \text{NSW}: \text{G}, \text{V}: \text{R}, \text{T}: \text{G}\}$

Weight:

$\text{Weight}(x) = 1 \cdot 1 \cdot 1 \cdot 1 \cdot 1 \cdot 1 \cdot 1 \cdot 1 \cdot 1 = 1$

Assignment:

$x' = \{\text{WA}: \text{R}, \text{NT}: \text{R}, \text{SA}: \text{B}, \text{Q}: \text{R}, \text{NSW}: \text{G}, \text{V}: \text{R}, \text{T}: \text{G}\}$

Weight:

$\text{Weight}(x') = 0 \cdot 0 \cdot 1 \cdot 1 \cdot 1 \cdot 1 \cdot 1 \cdot 1 \cdot 1 = 0$

- Consider the map coloring example. Here we are writing an assignment as a dictionary from variable (name) to value.
- For the first assignment, all the constraints (factors) are satisfied and evaluates to 1.
- For the second assignment, WA and NT have the same color (red), so $[\text{WA} \neq \text{NT}] = 0$. This zeros out the weight for the entire assignment.

Assignment weights

Definition: assignment weight

Each **assignment** $x = (x_1, \ldots, x_n)$ has a **weight**:

$$\text{Weight}(x) = \prod_{j=1}^{m} f_j(x)$$

An assignment is **consistent** if $\text{Weight}(x) > 0$.

Objective: find the maximum weight assignment

$$\arg\max_x \text{Weight}(x)$$

A CSP is **satisfiable** if $\max_x \text{Weight}(x) > 0$.

- Formally, the **weight** of an assignment x is the product of all the factors applied to that assignment ($\prod_{j=1}^{m} f_j(x)$). We say that an assignment is consistent if it has a non-zero weight.
- The objective in constraint satisfaction problem (what it means to solve a CSP) is to find the **maximum weight assignment**. A CSP is satisfiable if there exists a consistent assignment.
- Note: strictly speaking, a CSP only contains factors which are constraints (that return 0 or 1), but we consider a more general version of CSPs where weights can be arbitrary.
- Note: do not confuse the term "weight" in the context of factor graphs with the "weight vector" in machine learning.

Constraint satisfaction problems

Boolean satisfiability (SAT):

variables are booleans, factors are logical formulas $[X_1 \vee \neg X_2 \vee X_5]$

Linear programming (LP):

variables are reals, factors are linear inequalities $[X_2 + 3X_5 \leq 1]$

Integer linear programming (ILP):

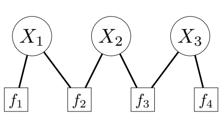
variables are integers, factors are linear inequalities

Mixed integer programming (MIP):

variables are reals and integers, factors are linear inequalities

- Constraint satisfaction problems are a general umbrella term that captures several important special cases, which are widely studied in the mathematical programming community.
- In SAT, all variables are boolean-valued and factors (constraints) are logical formulas. The goal is just to find any consistent assignment. While SAT is NP-complete, there has been extraordinary progress in SAT solving, and we can routinely solve SAT instances much larger than theory would predict.
- In linear programming, the variables are real-valued, and factors are linear inequalities. These problems can be solved efficiently using specialized methods (e.g., the simplex algorithm)
- ILPs and MIPs are hard to solve in general because they include integer values.

Summary



Variables, factors: specify locally

$$\text{Weight}(\{X_1 : \text{B}, X_2 : \text{B}, X_3 : \text{R}\}) = 1 \cdot 1 \cdot 2 \cdot 2 = 4$$

Assignments, weights: optimize globally

- In summary, we have formally defined factor graphs, where variables represent unknown quantities, and factors specify preferences for partial assignments. These allow us to specify preferences in a modular way: just "throw in" any desiderata you have.
- The weight of an assignment is the product of all the factors. The objective in solving a CSP is to find the maximum weight assignment, which is a global notion that must take into account all the factors at once.

Lecture

CSPs: Overview

CSPs: Definitions

CSPs: Examples

- In this module, we 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.

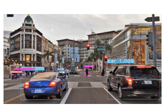
The exhibition must satisfy the following conditions:

- 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]

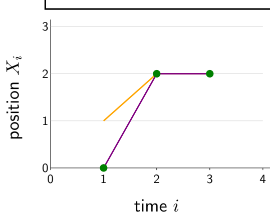
- The LSAT is a standardized test for law school which features questions that are logic puzzles. These can usually be formalized as a constraint satisfaction problem. CSPs offer a formulaic way of tackling these problems which could even be automated (though the hard part for computers is translating 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

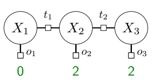
(O) Noisy sensors report positions: $0, 2, 2$.
(T) Objects can't teleport.
What trajectory did the object take?



- 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

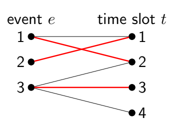
$\|x_i - x_{i+1}\|$	$t_i(x_i, x_{i+1})$
0	2
1	1
2	0

[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

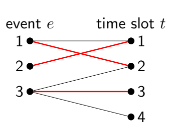


Problem: Event scheduling

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

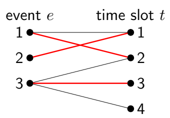
Have E events and T time slots
(C1) Each event e must be put in **exactly one** time slot
(C2) Each time slot t can have **at most one** event
(C3) Event e allowed in time slot t only if $(e, t) \in A$

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, $Y_t \in \{1, \ldots, E\} \cup \{\emptyset\}$; 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. One can show that each T-ary constraints can be converted into $O(T)$ binary constraints with $O(T)$ variables. After this transformation, the resulting formulation has T variables with domain size $E + 1$, $O(ET)$ variables with domain size $O(1)$ 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 mathematical 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 values computed in the Python program.
- Then we add a constraint for each program statement which asserts that the values are computed correctly.
- 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 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 relation that, given a value for the variables on both the left-hand side and the right-hand side, returns whether this setting is valid or not.
- The ramification of this is rather interesting: 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.

Overall Summary

- Constraint satisfaction problems as Factor graphs
- Definitions: variables factors, assignments, weights
- Examples: tracking, scheduling, program verification
- Next: Solving CSPs

- In summary, we started be defining constraint satisfaction problems as factor graphs
- Next, we covered some basic definitions, including variables factors, assignments and weights
- Then, we discussed example constructions of CSPs as factor graphs, including tracking, scheduling, and program verification
- Next Lecture, we will cover methods for solving CSPs