CS3243: Introduction to Artificial Intelligence
Lecture Notes 6: Simulated Annealing and Constraint Satisfaction

1. Simulated Annealing

In the field of condensed-mater physics, the spin-glass model states that all atoms have a spin of ± 1. Let M_i denote the spin of an atom i. As such, the energy E of an entire system of N atoms can be given by the formula

$$E = e^{-\frac{\sum_{i,j \epsilon N} J\, M_i M_j}{K_B T}}$$

where K_B is to the Boltzmann constant, and T is the temperature.

Each atom has a probability of $\frac{1}{2}$ of having each spin value, where

$$Pr[M_i = +1] = \tfrac{1}{2}.$$

A *configuration* refers to a set of assignment of spin values to each atom in the system. The configuration will have the lowest possible energy level if all atoms in the system have the same state, i.e.

$$M_i = M_j = +1 \text{ or } M_i = M_j = -1 \ \forall i, j.$$

The probability of a configuration C_1 transitioning to another configuration C_2 is dependent on their energy differences, where

$$Pr[C_1\ to\ C_2] \propto e^{-\frac{E\{C_1\} - E\{C_2\}}{K_B T}}$$

In metallurgy, the process of getting to a lowest-energy configuration of a system involves lowering the temperature T according to a *cooling schedule* $T(t)$, which is a function that maps time t to the temperature T. $T(t)$ usually decreases with t.

At a high temperature, we can transition from C_1 to C_2 easily even when $E\{C_1\} > E\{C_2\}$. However, at lower temperatures, the probability of transition gets lower because the numerator of the fraction begins to dominate.

In 1983, the condensed-matter physicists Kirkpatrick, Gelalt and Veechi found an application of this phenomenon in the field of computer science, using *value* as a proxy for *Energy*. They came up with the following **Simulated Annealing** algorithm:

```
SimulatedAnnealing(s)
    for t = 1 to ∞:
        s' ← RANDOMLY-PICKED-NEIGHBOUR(s)
        if Val(s') == 0:
            return s'
        s ← s' with Probability proportional to $e^{-\frac{\mathrm{Val}(s') - \mathrm{Val}(s)}{K_B T}}$
```

Alternatively, we could also set the probability of assigning state s' to s as the *sigmoid function*, where

$$Pr[s\ to\ s'] = \frac{1}{1 + e^{-\frac{Val(s') - Val(s)}{K_B T}}}$$

Another possible variant of the algorithm is to go from $s\ to\ s'$ if $Val(s') < Val(s)$, and otherwise transition from $s\ to\ s'$ with probability proportional to $e^{-\frac{Val(s')-Val(s)}{K_B T}}$.

In some cases, we might not know what the value of our goal state is. An example is the n-Queens problem as covered in the previous lecture, but for $\frac{3n}{2}$ queens. Such problems are known as **Optimization Problems**. A strategy to solve such problems is to keep track of a state with minimum $Val(s)$. The basic idea of such strategies is to allow mistakes to be made early on, and punish later.

To apply the Simulated Annealing algorithm to solve the n-Queens problem, simply let $Val(s)$ be the number of queens that attack each other.

2. Backtracking Search

A strategy of solving a problem such as the n-Queens problem is to always pick a value that is consistent with the previous choices so far, and to backtrack when a mistake is made.

We can model such a problem with the following parameters.

- **Variables:** $\{x_1, x_2, \cdots, x_4\}$
- **Domains:** D_i is the domain of the variable x_i. The domain of a variable refers to the set of values which the variable can take.
- **Constraints:** $NoAttack(x_1, x_2)$, $NoAttack(x_1, x_3)$, $NoAttack(x_1, x_4)$, $NoAttack(x_2, x_3)$, $NoAttack(x_2, x_4)$, $NoAttack(x_3, x_4)$.

Constraints can be specified using truth tables. For example, $NoAttack(x_1, x_2)$ maps different combinations of values for x_1 and x_2 to a Boolean value which indicates if the queens at x_1 and x_2 attack each other.

The algorithm which involves backtracking is as follows:

```
BacktrackSearch(problem, assignment):
      if AllVariablesAssigned(problem, assignment):
            return assignment
      var ← PickUnassignedVariable(problem, assignment)
      for val in OrderDomainValue(var, problem, assignment):
            if (val is consistent with assignment):
                  add {var = val} into assignment
```

```
                    result ← BacktrackSearch(problem, assignment)
                    if (result != failure):
                          return result
              remove {var=value} from assignment
        return failure
```

3. Constraint Satisfaction Problems

Once we assign a value to a variable X_i, we will have to look at all the constraints that X_i appears in, and there "infer" restriction on the rest of the variables.

- $NoAttack(x_1, x_2)$ means that $x_1 = 1 \Rightarrow x_2 \notin \{1,2\}$
- $NoAttack(x_1, x_3)$ means that $x_1 = 1 \Rightarrow x_3 \notin \{1,3\}$
- $NoAttack(x_1, x_4)$ means that $\mathrm{x}_1 = 1 \Rightarrow \mathrm{x}_4 \notin \{1,4\}$

Let us take a look at how inference mechanism in action.

Suppose that we know that $x_1 = 1$. Based on the constraint $NoAttack(x_1, x_2)$, we know that $x_2 \in \{3,4\}$. From the constraint $NoAttack(x_2, x_3)$, we can infer that $x_2 \in \{3,4\} \Rightarrow x_3 \notin \{3,4\}$, since $NoAttack(x_2 = 4, x_3 = 2)$ is True. Hence, $x_3 \notin \{1,3,4\}$, which means that definitely $x_3 = 2$. From this, apply $NoAttack(x_3, x_4)$ and we find that $x_4 \notin \{1,2,3,4\}$. Hence, $x_1 = 1 \Rightarrow x_4 \notin \{1,2,3,4\}$, which means that we have to backtrack.

With the ability to now infer restrictions, we now have an improved version of the backtracking search algorithm.

```
BacktrackSearch(problem, assignment):
        if AllVariablesAssigned(problem, assignment):
              return assignment
        var ← PickUnassignedVariable(problem, assignment)
        for val in OrderDomainValue(var, problem, assignment):
              if (val is consistent with assignment):
                    add {var = val} into assignment
                    inference ← Infer(problem, var, assignment)
                    add inference into assignment
                    if (inference != failure):
                          result ← BacktrackSearch(problem, assignment)
                    if (result != failure):
                          return result
              remove {var=value} and inference from assignment
        return failure
```