CS3243: Introduction to Artificial Intelligence **Fall 2020**

Lecture 6: September 16

Lecturers: Prof. Kuldeep S. Meel *Scribe: Ang Zheng Yong*

6.1 Recap of Week 5

6.1.1 n-queens puzzle

We used local search, where we take a state and then move towards the neighbours of this state with a more optimal $Val(s)$. However, this algorithm might get stuck in some local optimum.

6.2 Simulated Annealing

This idea came from metallurgy, where in high temperatures, particles are able to move from a location C_1 to C_2 easily when $E(C_2) > E(C_1)$, i.e.

$$Pr\{C_1 \to C_2\} \propto e^{-\frac{E(C_2)-E(C_1)}{k_B T}}$$

6.2.1 Generic algorithm for simulated annealing

Algorithm 1 Simulated Annealing

1: **procedure** SimulatedAnnealing(s)
2:     **for** $t = 1$ to ∞ **do**
3:         $s' \leftarrow$ a random neighbour of s
4:         **if** $Val(s') = 0$ **then**
5:             **return** s'
6:         **end if**
7:         $s \leftarrow s'$ with $Pr \propto e^{-\frac{Val(s')-Val(s)}{k_B T(t)}}$ ▷ k_B is the Boltzmann constant, and $T(t)$ is a decreasing function on t
8:     **end for**
9: **end procedure**

The general idea is to allow the algorithm to make mistakes early on, and punish it later (lower probability of moving to its neighbours when t is large).

The algorithm can be modified in a number of ways to give rise to various variants:

1. The constant factor used for the proportionality in line 7 may vary (the sigmoid function $\frac{1}{1+e^{-x}}$ is most commonly used).
2. The mechanism of selecting s' in line 3 may vary in different implementations as well.
3. Line 7 may also be replaced with the following:

- If $Val(s') < Val(s)$, $s \leftarrow s'$.
- Else, $s \leftarrow s'$ with $Pr \propto e^{-\frac{Val(s')-Val(s)}{k_B T}}$.

6.2.2 Back to the n-queens problem

In the n-queens problem, $Val(s)$ represented the number of queens that are attacking each other.

By using Algorithm 1, we will explore neighbouring states even when $Val(s_{i+1}) > Val(s_i)$ with a probability as defined in the algorithm. Thus, we might be able to get out of the local optima (as shown in the figure below) and approach the global optima (i.e. the state where none of the queens are attacking each other).

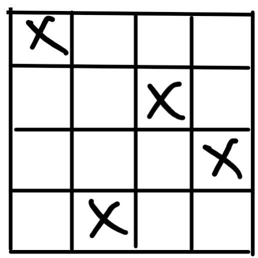

Figure 6.1: A state in the 4-queens puzzle, where "×" denotes the locations of the queens

However, there is still no guarantee of finding the global optima, since the simulated annealing algorithm is probabilistic. Is there any other approach which we could take?

6.3 Constraint Satisfaction Problems (CSPs)

We could define the 4-queens problem as a CSP:

1. Variables: $\{x_1, x_2, x_3, x_4\}$
2. Values/Domain for each of the variables: $\{D_1, D_2, D_3, D_4\}$, where each D_i represents the domain of variable x_i.
3. Constraints: NoAttack(x_1, x_2), NoAttack(x_1, x_3), ..., NoAttack(x_3, x_4).
   - We can represent the outcomes of NoAttack(x_i, x_j) in a truth table.

6.3.1 Solving a CSP

After modelling the problem as a CSP, we shall attempt to solve it. The key steps in solving a CSP include:

1. Select a value for the first variable.
2. Select a value for that next variable, that is consistent with the choices so far.
3. If we hit a dead end, backtrack. Else, repeat step 2.

For instance, let us consider the 4-queens problem:

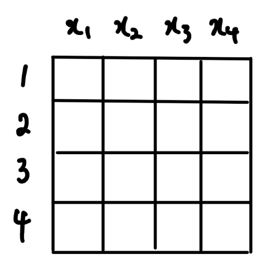

Figure 6.2: Start state of the 4-queens problem

1. We start by setting $x_1 = 1$.
    (a) Then, the domain of x_2 will be restricted to $\{3, 4\}$.
    (b) Assume we set $x_2 = 3$.
        i. Then, the domain of x_3 will be $\{\varnothing\}$. We have hit a dead end, so we backtrack.
    (c) Now, we set $x_2 = 4$.
        i. Then, the domain of x_3 will be restricted to $\{2\}$.
        ii. We set $x_3 = 2$.
            A. Then, the domain of x_4 will be $\{\varnothing\}$. We hit another dead end, so we backtrack.
2. Next, we set $x_1 = 2$.
    (a) Since the domain of x_2 is restricted to $\{4\}$, we set $x_2 = 4$.
        i. The domain of x_3 is restricted to $\{1\}$, so we set $x_3 = 1$.
            A. Finally, the domain of x_4 is $\{3\}$, and we set $x_4 = 3$ and obtain a valid solution.

The algorithm for solving the CSP is shown below.

Algorithm 2 Backtrack Search

```
procedure BACKTRACKSEARCH(prob, assign)
    if all var in (prob, assign) are assigned then
        return assign
    end if
    var ← unassigned variable in (prob, assign)
    for val ∈ ORDERDOMAINVALUE(var, prob, assign) do
        if val is consistent with assign then
            add {var = val} to assign
            result ← BACKTRACKSEARCH(prob, assign)
            if result ! = failure then
                return result
            end if
            remove {var = val} from assign        ▷ Backtracking step
        end if
    end for
    return failure
end procedure
```

6.3.2 Inference

Is it possible to avoid making the unnecessary moves that the algorithm shown in the previous page would have made?

Idea: Once we have assigned a value to a variable x_i, we look at all the constraints that x_i appears in.

- From there, we will infer the restrictions on the remaining variables.
- If one of the variables cannot have a value, then we should immediately backtrack.

We can improve our algorithm via inference.

Algorithm 3 Backtrack Search

```
procedure BacktrackSearch(prob, assign)
    if all var in (prob, assign) are assigned then
        return assign
    end if
    var ← unassigned variable in (prob, assign)
    for val ∈ OrderDomainValue(var, prob, assign) do
        if val is consistent with assign then
            add {var = val} to assign
            inference ← Infer(var, prob, assign)                ▷ Inference step
            add inference to assign
            if inference != failure then
                result ← BacktrackSearch(prob, assign)
                if result ! = failure then
                    return result
                end if
            end if
            remove {var = val} and inference from assign        ▷ Backtracking step
        end if
    end for
    return failure
end procedure
```

Remark: Whenever the algorithm is doing inference, it is inferring under the current *assign*. Hence, whenever it backtracks, it will need to remove *inference* from *assign*.