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

Lecture 7: October 7

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

7.1 Recap of Week 6

In week 6, we discussed backtracking search, where we first pick a variable and assign a value to it, and then check if the assigned value satisfies the constraints. The algorithm was given as:

Algorithm 1 Backtrack Search

```
1:  procedure BacktrackSearch(prob, assign)
2:      if all var in (prob, assign) are assigned then
3:          return assign
4:      end if
5:      var ← unassigned variable in (prob, assign)
6:      for val ∈ OrderDomainValue(var, prob, assign) do
7:          if val is consistent with assign then
8:              add {var = val} to assign
9:              result ← BacktrackSearch(prob, assign)
10:             if result != Failure then
11:                 return result
12:             end if
13:             remove {var = val} from assign                ▷ Backtracking step
14:         end if
15:     end for
16:     return Failure
17: end procedure
```

Then, we moved further to discuss the possibility of inference on future moves, so that we could potentially avoid making unnecessary moves (i.e. moves that lead to a dead end) that the naïve backtracking algorithm would have made. The algorithm for backtracking with inference is shown on the next page.

Algorithm 2 Backtracking with Inference

```
1: procedure BacktrackSearch(prob, assign)
2:     if all var in (prob, assign) are assigned then
3:         return assign
4:     end if
5:     var ← unassigned variable in (prob, assign)
6:     for val ∈ OrderDomainValue(var, prob, assign) do
7:         if val is consistent with assign then
8:             add {var = val} to assign
9:             inference ← Infer(var, prob, assign)                ▷ Inference step
10:            add inference to assign
11:            if inference != Failure then
12:                result ← BacktrackSearch(prob, assign)
13:                if result != Failure then
14:                    return result
15:                end if
16:            end if
17:            remove {var = val} and inference from assign        ▷ Backtracking step
18:        end if
19:    end for
20:    return Failure
21: end procedure
```

This week, we will be discussing how we could implement the inference component of our algorithm.

7.2 Inference

Before implementing the inference component, we will need to first decide on a data structure which we could use to store our inferences.

7.2.1 Representing inferences

In lecture, we discussed the possibility of representing our inferences as a set of values $\{X \notin S\}$, where S is a set of values which the variable X cannot possibly have due to the constraints of the problem.

Notation: in CS3243, we will sometimes use $X \neq 1$ to mean $X \notin \{1\}$, and vice versa.

7.2.2 Domain computation

We also introduced a helper function *ComputeDomain(X, assign, inference)*, which returns the set of values of X that X can take under the current assignment and inference.

For example, if $Dom(X) = \{1, 2, 3, 4\}$, where $Dom(X)$ represents the domain of X:

- If *assign* $= [\{X = 1\}]$ and *inference* $= [\{X \notin \{2, 3\}\}]$, then *ComputeDomain* will return $X \in \{1\}$.
- If *assign* $= [\{Y = 1\}]$ and *inference* $= [\{X \notin \{2, 3\}\}]$, then *ComputeDomain* will return $X \in \{1, 4\}$.

7.2.3 Formalizing inference

Now, we will formalize the ideas discussed in Week 6 and discuss the algorithm for inference. The algorithm for inference is shown below:

Algorithm 3 AC-3

```
1: procedure INFER(prob, var, assign)
2:     varQueue ← [var]
3:     while varQueue is not empty do
4:         Y ← varQueue.pop()                          ▷ We pick a variable from the queue
5:         for each constraint C where Y ∈ Vars(C) do   ▷ Vars(C) rep. the set of variables of C
6:             for all X ∈ Vars(C)\Y do                 ▷ All the variables of C other than Y
7:                 S ← COMPUTEDOMAIN(X, assign, inference)
8:                 for each v ∈ S do
9:                     if ∀var ∈ Vars(C)\X, C[X ↦ v] is not satisfied for all var = val assignments then
10:                        inference.add(X ∉ {v})
11:                    end if
12:                end for
13:                T ← COMPUTEDOMAIN(X, assign, inference)
14:                if T = {∅} then                      ▷ Inference concludes that there are no possible values for X
15:                    return Failure
16:                end if
17:                if S ≠ T then                        ▷ We were able to infer something about X
18:                    varQueue.add(X)
19:                end if
20:            end for
21:        end for
22:    end while
23:    return inference
24: end procedure
```

Remark: The order in which the variables X are explored (line 6 of the algorithm) may have an impact on the speed of inference, and they will be explored in future modules on CSPs.

We will now revisit the n-queens problem from previous lectures, and attempt to arrive at a solution using Algorithm 3.

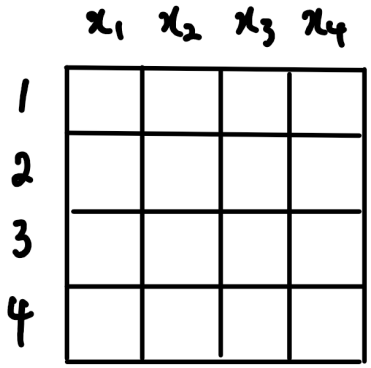

Figure 7.1: A starting state of the n-queens problem, where $n = 4$

1. Initially, we assign $x_1 = 1$.
   (a) If $x_2 = 1$, then C will not be satisfied, so we learn that $x_2 \neq 1$ (i.e. we add $x_2 \notin \{1\}$ to our inference). We will also learn that $x_2 \neq 2$.
   (b) Similarly, for x_3, we will infer that $x_3 \notin \{1, 3\}$.
   (c) For x_4, we will also infer that $x_4 \notin \{1, 4\}$.
2. Following our initial assignment, we have learnt that $Dom(x_2) = \{3, 4\}$. Regardless of whether we assign $x_2 = 3$ or $x_2 = 4$, we will be able to infer that $x_3 \neq 4$, and so we add $x_3 \notin \{4\}$ into our inference.
3. Eventually, we will hit a dead end and move on to assign $x_1 = 2$, and the process of inference will repeat.

7.2.4 Reducing the cost of inference

As discussed in the previous lecture, inference can be expensive, and sometimes it may not be worth checking all possible future states. Is there any way for us to eliminate unnecessary inference?

One way of reducing the cost of inference is by adopting an alternative implementation of inference. Some possible alternative implementations include:

1. *Forward checking*: we remove lines 17 to 19 from Algorithm 3. Thus, we are effectively only inferring one step ahead of us, and not too much into the future.
2. For line 17 of Algorithm 3, we replace "**if** $S \neq T$" with "**if** $|T| = 1$", i.e. we only infer further when there is only 1 value that variable X can take.
   - This implementation is more expensive that the former implementation, but it enables us to infer more about future states.

In practice, implementation 2 is usually used more than implementation 1. However, the choice of implementation mainly depends on the nature of the problem that we are trying to solve.

7.2.5 Speeding up inference

There are some ways which we could use to speed up inference. We have discussed two possible ways in lecture.

1. **Picking unassigned variables:**
   - One heuristic which we could adopt to pick the variable Y is the *Minimum Remaining Values (MRV)* heuristic. With this heuristic, we will always choose the variables with the smallest domain first.
     - For example, if *ComputeDomain(X_1, assign, inference)* $= \{1\}$ whereas *ComputeDomain(X_2, assign, inference)* $= \{2, 3\}$, then we will pick X_1 before X_2.
2. **Ordering domain values:**
   - We can also adopt a heuristic for picking domain values in line 8 of Algorithm 3: we can start from the values which are most likely to succeed, before going to the values which are less likely to give a successful assignment for subsequent variables.
     - One such heuristic is the *Least-Constraining-Value* heuristic. With this heuristic, we will always select the value that tries to leave the most flexibility for the assignment of subsequent variables.

7.2.6 Storing constraints

In the previous lecture, we have also briefly discussed on some ways of storing our constraints.

1. Truth table
   - For example, for the n-queens problem, we can store our constraints in the following truth table:

x_1	x_2	$NoAttack(x_1, x_2)$
1	1	*False*
1	2	*False*
1	3	*True*
$\vdots$	$\vdots$	$\vdots$

   - However, the limitation of using truth tables to store constraints is that our truth tables may be very large and expensive to store/read.
2. Mathematical methods
   - We can also express our constraints as functions/relations, or using arithmetic.

7.3 Back to CSPs

7.3.1 Local search on CSPs

After backtracking search, CSP researchers found that local search had a better performance than backtracking search, before they moved on to backjumping-enabled search.

To carry out local search on constraint satisfaction problems, we can simply replace the *BacktrackSearch* recursive call in Algorithm 2 with a recursive call on local search. The algorithm is shown below:

Algorithm 4 Local Search on CSPs

```
1: procedure BacktrackSearch(prob, assign)
2:     if all var in (prob, assign) are assigned then
3:         return assign
4:     end if
5:     var ← unassigned variable in (prob, assign)
6:     for val ∈ OrderDomainValue(var, prob, assign) do
7:         if val is consistent with assign then
8:             add {var = val} to assign
9:             inference ← Infer(var, prob, assign)                ▷ Inference step
10:            add inference to assign
11:            if inference != Failure then
12:                result ← LocalSearch(prob, assign)
13:                if result != Failure then
14:                    return result
15:                end if
16:            end if
17:            remove {var = val} and inference from assign        ▷ Backtracking step
18:        end if
19:    end for
20:    return Failure
21: end procedure
```

7.3.2 CSP Complexity

How hard are CSPs? CSPs are generally *NP*-complete, where *NP* refers to the class of algorithms that can be solved in non-deterministic polynomial time. What about other variants of CSPs?

1. *Binary CSPs*: where every constraint is defined over 2 variables.
   - Binary CSPs are also *NP*-complete.
2. *Boolean CSPs (SAT)*: where the domain of every variable is $\{0, 1\}$.
   - SAT is also *NP*-complete.
3. *2-SAT*: a binary and boolean CSP.
   - 2-SAT $\in P$. However, 3-SAT is *NP*-complete.

A fundamental problem in computer science is: is *P=NP*?

7.4 Midterm Discussion

7.4.1 Question 6

Taking the optimal path from s_0 to u to be

$$\pi^* : s_0, s_1, \ldots, s_k, s_{k+1}, \ldots, u$$

If we can prove that:

1. $f(s_0) \leq f(s_1) \leq \ldots \leq f(u)$ (required only along the optimal path), and
2. $\hat{f}_{\text{pop}}(s_i) = f(s_i)$.

then we could use these conditions to show that an A^* graph search with the heuristic will be optimal.

Hence, it suffices to show that condition 1 holds for the given heuristic *along the optimal path*. This can be shown by

$$\begin{aligned}
\forall s_i \in \pi^*, h(s_i) &= \frac{OPT(s_i)}{2} \\
&= \frac{c(s_i, s_{i+1}) + OPT(s_{i+1})}{2} \\
&< c(s_i, s_{i+1}) + \frac{OPT(s_{i+1})}{2} \\
&= c(s_i, s_{i+1}) + h(s_{i+1})
\end{aligned}$$

By showing the the heuristic is consistent along the optimal path, we would have shown that $\hat{f}_{\text{pop}}(s_g) = f(s_g)$, and thus the algorithm is optimal.

Remark: Consistency is a sufficient property for optimality, but it is not necessary. Consistency implies condition 1, but it suffices to show a property weaker than condition 1.