CS3243 **AY20/21 Sem1**

Lecture 6: September 16

Lecturer: MEEL, Kuldeep S. *Scribes: Song Qifeng*

Note: *LaTeX template courtesy of UC Berkeley EECS dept.*

6.1 Simulated annealing

6.1.1 Introduction to simulated annealing

Traditional local search algorithms suffer from the shortage of terminating at local minimums. To solve this problem, an algorithm may need to mix 'downhill' with 'random walks' such that the algorithm is able to get out of local minimum.

Inspired by what they had observed in Condensed Matter Physics, three physicists, Kirkpatrick, Gelatt, and Vecchi, proposed the Simulated Annealing algorithm in 1983.

Simulated annealing algorithm is interpreted as a slow decrease in the probability of accepting worse solutions (i.e. the temperature decreases) as the solution space is explored. Accepting worse solutions allows for a more extensive search for the global optimal solution.

6.1.2 Algorithm

Algorithm 1 Simulated annealing

```
1:  function SIMULATED-ANNEALING(problem, schedule) returns a solution state
2:      problem ← a problem
3:      schedule ← a mapping from time to temperature
4:      current ← MAKE-NODE(problem.INITIAL-STATE)
5:      for t = 1 to ∞ do do
6:          T ← schedule(t)
7:          if T = 0 then return current
8:          next ← a randomly selected successor of current
9:          if Val(next) < Val(current) then
10:             current ← next
11:         if for some probability then
12:             current ← next
```

6.2 Constraint satisfaction problem

6.2.1 Introduction

A **constraint satisfaction problem**, or CSP, is a problem that is represented using

- a set of variables
- each of this variables has a value
- some constraints to follow

A problem like this is solved when each variable is assigned a value and all constraints are satisfied.

6.2.2 Define a CSP

A CSP consists of 3 components,

- X $\leftarrow$ a set of variables $\{X_1, X_2, \ldots, X_n\}$
- D $\leftarrow$ a set of domains $\{D_1, D_2, \ldots, D_n\}$, one for each variable
- C $\leftarrow$ is a set of constraints $\{C_1, C_2, \ldots, C_n\}$ that specify allowable combinations of values.

Each constraint C_i is a pair ¡$scope, relation$¿, where $scope$ is a tuple of variables that participate in C_i and $relation$ is the relation that defines the values that these variables can take.

To solve a CSP, we need to define a state space and the notion of a solution. Each state is defined by an **assignment** of values to some or all of the variables.

An assignment that does not violate any constraints is called a **consistent assignment**. An assignment where all variables are assigned is called a **complete assignment**. A **partial assignment** is one that assigns values to only some of the variables.

6.2.3 Insights from n-queen problem

First, we define the CSP in n-queen problem.

- X $\leftarrow \{X_1, X_2, X_3, X_4\}$, stands for the position of chess in each column respectively.
- D $\leftarrow \{D_1, D_2, D_3, D_4\}$, each stands for row a chess ca be placed on a column.
- C $\leftarrow$ no queens attack each other.

$X_1 = 1 \ \rightarrow \ X_2 = 3 \ \rightarrow \ X_3 = N/A \ \rightarrow \ backtrack$

$X_1 = 1 \ \rightarrow \ X_2 = 4 \ \rightarrow \ X_3 = 2 \ \rightarrow \ X_4 = N/A \ \rightarrow \ backtrack$

...

Now we conclude a solution:

1. Pick a column
2. Pick a position
3. Pick a value that is consistent with the choices so far
4. Backtrack

6.2.4 Backtracking search for CSP

Before we come up with the solution of backtracking, we first define a concept called **inference** which will be elaborated in subsequent lectures.

Inference means looking one step further from the current move and check whether the current move will lead to other variables having no consistent value to choose.

Algorithm 2 Backtracking

```
1:  function BacktrackSearch(problem, assign)
2:      if AllVarsAssigned(problem, assign) then return assign
3:      var ← PickUnassignedVar(problem, assign)
4:      for val in OrderDomainValue(var, problem, assign) do
5:          if val is consistent with the assign then
6:              add var = val in assign
7:              inference ← Infer(problem, var, assign)
8:              add inference in assign
9:              if inference! = failure then
10:                 result ← BacktrackSearch(problem, assign)
11:                 if result! = failure then return result
12:             remove var = val and inference from assign
```