Week 3

Wednesday, June 24, 2020 6:05 PM

Optimization

Def :

Choosing the best option from a set of options

Local Search :

Search algorithms that maintain a single node and searches by moving to a neighboring node

Usage :

when we don't care about the path but what is the solution

- **Think about the problem as a state-space landscape :**
  In which we try to find :
  - Global MAX using objective function
  - Global MIN using cost function
  - Examples :
    - **Hill climbing**

      Function Hill_Climb(problem):
      - Current = initial state of the problem
      - Repeat :
        - Neighbor = (lowest-highest) value of a neighbor of current
        - If neighbor not better than my current :
          - Return current
        - Current = neighbor

    - **In hill climbing we can get stuck at a local max or local min or a flat local max or shoulder**
      That why we have variants

Hill Climbing Variants :

Steepest-ascent	choose the highest value neighbor
stochastic	choose randomly from higher value neighbors
First-choice	choose the first higher value neighbor
Random-restart	Conduct Hill Climbing multiple times
Local beam search	Choose k highest-valued neighbors

** all these Variants does not move ever to a worse state from its current state

If you want that use simulated Annealing

Simulated Annealing

- Early on, higher "temperature" :
  - More likely to accept neighbors that are worse than the current state
- Later on, lower "temperature" :
  - Less likely to accept neighbors that are worse than the current state

Sude Code :

- Function Simulated_annealing(problem, max):
  - Current = initial state of problem
  - From t = 1 to max :
    - T = temperature(t)
    - Neighbor = random neighbor of current
    - ΔE = how much better neighbor is than current
    - If ΔE > :
      - Current = neighbor
    - Else :
      - With probability of $e^{\Delta E/T}$ set current = neighbor
  - Return current

Linear Programming

Goal :

- Minimize a cost function $c_1x_1 + c_2x_2 + \dots$
  With constrains of form $a_1x_{1+\dots} \le b$
  Or of the form $a_1x_{1+\dots} = b$
  With bounds for each variable $l_i \le X_i \le u_i$

  **** we deal with < or = constrains so if we have > :**
  We just use (-)

Examples for linear Programming Algorithms:

- Simplex
- Interior-Point

Constraint Satisfaction

We have some number of variables and every variable have a
Value from a set of values
And that value satisfy all the constrains

- **Constraint Satisfaction Problem has :**
  - Set of variables {X_1, X_2, ….}
  - Set of values for each variable {D_1, D_2, ….}
  - Set of constraints C

- Types of Constraints :
  - **Hard constraint :**
    - Constraint that must be satisfied in a correct solution
  - **Soft Constraints :**
    - Constraint that express some notion of which solution is preferred over others
- ---
- Classes of Constrains :
  - **Unary Constraints :**
    - Constraint that only involve one variable
  - **Binary Constraints :**
    - Constraint that involve two variables
- ---
- Node Consistency :
  - All the values in a variable`s domain satisfy all the variable Unary Constraints
- ---
- Arc Consistency :
  - All the values in a variable`s domain satisfy all the variable binary Constraints
  - To make X arc-consistent with respect to Y :
    - Remove elements from X domain until :
      - EVERY choice for X has a possible choice in Y
  - ---
  - **Sudo Code :**
    - Function Revise(csp, X, Y):
      - Revised = false
      - For x in X.domain :
        - If no y in Y.domsin satisfy Constants for(X, Y) :
          - Delete x from X.domain
          - Revised = true
      - Return revised

AC-3

Force Arc consistency over all the nodes

Sudo Code :

- Function AC-3(cps):
  - Queue = all arcs in csp
  - While queue is not empty :
    - (X, Y) = dequeue(queue)
    - If revise(csp, X, Y):
      - If X.domain == 0:
        - Return false
      - For each Z in X.neighbors - {Y}:
        - Enqueue(queue, (Z, X))
  - Return true

Csp as a Search Problem

Consist of :

- Initial state :
  - empty assignment (no variable)
- Action :
  - add a (variable = value) to the assignment
- Transition Model :
  - show how adding an action change the assignment
- Goal test:
  - Check if all the variables are assigned and all the constraints are satisfied
- **Path cost function :**
  - All paths have some cost !! (doesn't really matter)

----
* The search Algorithm used in **CSP is :** BACKTRACIKNG

Backtracking Search :

- **SUDO CODE :**

- Function Backtrack(assignment, csp):
  - if assignment is complete:
    - Return assignment
  - Var = Select_Unassigned_Var(assignment, csp)
  - For value in Domain_values(var, assignment, csp):
    - If value is consistent with assignment:
      - Add {var = value} to the assignment
      - Result = Backtrack(assignment, csp)
      - If result is not failure:
        - Return result
    - Remove {var = value} from assignment
  - Return failure

Heuristics to use to improve the efficiency of the search process:

- **Select_unsigned_variable(assignment, csp):**
  - We can use:
    - **Minimum remaining values (MRV) heuristic:**
      - Choose the variable with the smallest domain
    - **degree heuristic:**
      - Choose the variable that has the highest degree
- ---
- Domain_values(var, assignment, csp):
  - **Least constraining value heuristic:**
    - **Return values order by :**
      - The number of choice that are removed from the neighbors domains by it
    - **Try least constraining value first**

We can improve this by :

- USING Maintaining arc-consistency WHICH IS :
  - **Algorithm for enforcing arc-consistency Every time we make a new assignment**

- USING Maintaining arc-consistency WHICH IS :
  - **Algorithm for enforcing arc-consistency Every time we make a new assignment**
  - ****When we make a new assignment to X :**
    - Call AC-3 with a queue of all arc (Y, X)
      - Y : is a neighbor of X
    - -------------------------------------
    - **SUDO CODE SAME AS BACKTRACK WITH :**
      - AFTER : add {var = value} to assignment
      - Inference = inference(assignment, csp)
      - If inference != failure :
        - ADD INFERENCE TO ASSIGNMENT
      - AND CONTINE BAACKTRACK CODE …
      - CHANGE : remove {var = value} to:
        - remove {var = value} and inference from assignment

- The number of choice that are removed from the neighbors domains by it

- **Try least constraining value first**
