pseudo-code

Knowledge-based agents

```
function KB-Agent(percept) returns an action
  static: KB, a knowledge base
          t, a counter, initially 0, indicating time

  Tell(KB, Make-Percept-Sentence(percept, t))
  action ← Ask(KB, Make-Action-Query(t))
  Tell(KB, Make-Action-Sentence(action, t))
  t ← t + 1
  return action
```

Resolution algorithm

```
function PL-Resolution(KB, α) returns true or false
  inputs: KB, the knowledge base, a sentence in propositional logic
          α, the query, a sentence in propositional logic

  clauses ← the set of clauses in the CNF representation of KB ∧ ¬α
  new ← { }
  loop do
      for each C_i, C_j in clauses do
          resolvents ← PL-Resolve(C_i, C_j)
          if resolvents contains the empty clause then return true
          new ← new ∪ resolvents
      if new ⊆ clauses then return false
      clauses ← clauses ∪ new
```

DPLL

function DPLL-SATISFIABLE?(s) **returns** *true* or *false*
  inputs: s, a sentence in propositional logic

  clauses ← the set of clauses in the CNF representation of s
  symbols ← a list of the proposition symbols in s
  return DPLL(*clauses*, *symbols*, { })

---

function DPLL(*clauses*, *symbols*, *model*) **returns** *true* or *false*

  if every clause in *clauses* is true in *model* **then return** *true*
  if some clause in *clauses* is false in *model* **then return** *false*
  P, *value* ← FIND-PURE-SYMBOL(*symbols*, *clauses*, *model*)
  if P is non-null **then return** DPLL(*clauses*, *symbols* – P, *model* ∪ {P=*value*})
  P, *value* ← FIND-UNIT-CLAUSE(*clauses*, *model*)
  if P is non-null **then return** DPLL(*clauses*, *symbols* – P, *model* ∪ {P=*value*})
  P ← FIRST(*symbols*); *rest* ← REST(*symbols*)
  return DPLL(*clauses*, *rest*, *model* ∪ {P=*true*}) **or**
      DPLL(*clauses*, *rest*, *model* ∪ {P=*false*}))

Algorithm:

Given a learning algorithm A and a dataset D

Step 1: Randomly partition D into k equal-size subsets D_1, ..., D_k

Step 2:

For j = 1 to k

  Train A on all D_i, $i \in 1, ..., k$ and $i \neq j$, and get f_j.

  Apply f_j to D_j and compute E^{Dj}

Step 3: Average error over all folds: $\sum_{j=1}^{k} (E^{\mathcal{D}_j})$

Gradient Descent

Repeat {

Simultaneously update for all β's

$$\beta_j := \beta_j - \alpha \frac{\partial}{\partial \beta_j} R(\beta)$$

}

After some calculus:

Repeat {

Simultaneously update for all β's

$$\beta_j := \beta_j - \alpha \sum_{i=1}^{n} (f(x) - y) x_j$$

}

Note: Same as linear regression BUT with the new function *f*.

1. Initialize the example weights, $w_i = 1/n$, $i = 1, \ldots, n$.
2. For $m = 1$ to M (number of weak learners)
   (a) Fit a classier $G_m(x)$ to training data using the weights w_i.
   (b) Compute
   $$err_m := \frac{\sum_{i=1}^{n} w_i \ 1\{y_i \neq G_m(x_i)\}}{\sum_{i=1}^{n} w_i}$$
   (c) Compute
   $$\alpha_m = \frac{1}{2} log(\frac{1-err_m}{err_m})$$
   (d) Compute
   $$w_i \leftarrow w_i . exp[-\alpha_m y_i G_m(x_i)] \quad for \ i = 1, \cdots, n.$$
3. Output
   $$G(x) = sign[\sum_{m=1}^{M} \alpha_m G_m(x)]$$

Bagging

Training

For b = 1, ..., B

1. Draw a bootstrap sample B_b of size L from training data.
2. Train a classier f_b on B_b.

Classification: Classify by majority vote among the B trees:

$$f_{avg} := \frac{1}{B} \sum_{b=1}^{B} f_b(x)$$

Clustering: K-Means

Algorithm K-Means:

Initialize randomly μ_1, ..., μ_k.

Repeat

Assign each point xi to the cluster with the closest μ_j.

Calculate the new mean for each cluster as follows:

$$\mu_j = \frac{1}{|\mathcal{C}_j|} \sum_{x_i \in \mathcal{C}_j} x_i$$

Until convergence*.

*Convergence: Means no change in the clusters OR maximum number of iterations reached.

How to set *k* to optimally cluster the data?

G-means algorithm

1. Initialize *k* to be a small number
2. Run k-means with those cluster centers, and store the resulting centers as C
3. Assign each point to its nearest cluster
4. Determine if the points in each cluster fit a Gaussian distribution (Anderson-Darling test).
5. For each cluster, if the points seem to be normally distributed, keep the cluster center. Otherwise, replace it with two cluster centers.
6. Repeat this algorithm from step 2. until no more cluster centers are created.

34

Backtracking search

```
function BACKTRACKING_SEARCH(csp) returns a solution, or failure
    return BACKTRACK({}, csp)

function BACKTRACK(assignment, csp) returns a solution, or failure
    if assignment is complete then return assignment
    var = SELECT_UNASSIGNED-VARIABLE(csp)
    for each value in ORDER_DOMAIN_VALUES (var, assignment, csp)
        if value is consistent with assignment then
            add {var = value} to assignment
            result = BACKTRACK(assignment, csp)
            if result ≠ failure then return result
            remove {var = value} from assignment
    return failure
```

Arc consistency

Algorithm that makes a CSP arc-consistent!

```
function AC-3( csp)
returns False if an inconsistency is found, True otherwise
inputs: csp, a binary CSP with components (X, D, C)
local variables: queue, a queue of arcs, initially all the arcs in csp
while queue is not empty do
    (X_i, X_j) = REMOVE-FIRST(queue)
    if REVISE(csp, X_i, X_j)then
        if size of D_i = 0 then return False
        for each X_k in X_i.NEIGHBORS − {X_j} do
            add (X_k, X_i) to queue
return true


function REVISE( csp, X_i, X_j)
returns True iff we revise the domain of X_i
revised = False
for each x in D_i do
    if no value y in D_j allows (x, y) to satisfy the constraint between X_i and X_j then
        delete x from D_i
        revised = True
return revised
```

The minimax algorithm

```
/* Find the child state with the lowest utility value */

function MINIMIZE(state)
    returns TUPLE of ⟨STATE, UTILITY⟩ :

    if TERMINAL-TEST(state):
        return ⟨NULL, EVAL(state)⟩

    ⟨minChild, minUtility⟩ = ⟨NULL, ∞⟩

    for child in state.children():
        ⟨_, utility⟩ = MAXIMIZE(child)

        if utility < minUtility:
            ⟨minChild, minUtility⟩ = ⟨child, utility⟩

    return ⟨minChild, minUtility⟩
```

```
/* Find the child state with the highest utility value */

function MAXIMIZE(state)
    returns TUPLE of ⟨STATE, UTILITY⟩ :

    if TERMINAL-TEST(state):
        return ⟨NULL, EVAL(state)⟩

    ⟨maxChild, maxUtility⟩ = ⟨NULL, −∞⟩

    for child in state.children():
        ⟨_, utility⟩ = MINIMIZE(child)

        if utility > maxUtility:
            ⟨maxChild, maxUtility⟩ = ⟨child, utility⟩

    return ⟨maxChild, maxUtility⟩
```

```
/* Find the child state with the highest utility value */

function DECISION(state)
    returns STATE :

    ⟨child, _⟩ = MAXIMIZE(state)

    return child
```

$\alpha - \beta$ pruning

```
/* Find the child state with the lowest utility value */
function MINIMIZE(state, α, β)
    returns TUPLE of ⟨STATE, UTILITY⟩ :

    if TERMINAL-TEST(state):
        return ⟨NULL, EVAL(state)⟩

    ⟨minChild, minUtility⟩ = ⟨NULL, ∞⟩

    for child in state.children():
        ⟨_, utility⟩ = MAXIMIZE(child, α, β)

        if utility < minUtility:
            ⟨minChild, minUtility⟩ = ⟨child, utility⟩

        if minUtility ≤ α:
            break

        if minUtility < β:
            β = minUtility

    return ⟨minChild, minUtility⟩
```

```
/* Find the child state with the highest utility value */
function MAXIMIZE(state, α, β)
    returns TUPLE of ⟨STATE, UTILITY⟩ :

    if TERMINAL-TEST(state):
        return ⟨NULL, EVAL(state)⟩

    ⟨maxChild, maxUtility⟩ = ⟨NULL, −∞⟩

    for child in state.children():
        ⟨_, utility⟩ = MINIMIZE(child, α, β)

        if utility > maxUtility:
            ⟨maxChild, maxUtility⟩ = ⟨child, utility⟩

        if maxUtility ≥ β:
            break

        if maxUtility > α:
            α = maxUtility

    return ⟨maxChild, maxUtility⟩
```

```
/* Find the child state with the highest utility value */
function DECISION(state)
    returns STATE :

    ⟨child, _⟩ = MAXIMIZE(state, −∞, ∞)

    return child
```

Greedy search: Pseudo-code

```
function GREEDY-BEST-FIRST-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :  /* Cost f(n) = h(n) */

    frontier = Heap.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.deleteMin()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.insert(neighbor)
            else if neighbor in frontier:
                frontier.decreaseKey(neighbor)

    return FAILURE
```

Hill climbing: Pseudo-code

```
function HILL-CLIMBING(initialState)
    returns State that is a local maximum

    initialize current with initialState

    loop do
        neighbor = a highest-valued successor of current

        if neighbor.value ≤ current.value:
            return current.state

        current = neighbor
```

A* search: Pseudo-code

```
function A-STAR-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :  /* Cost f(n) = g(n) + h(n) */

    frontier = Heap.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.deleteMin()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.insert(neighbor)
            else if neighbor in frontier:
                frontier.decreaseKey(neighbor)

    return FAILURE
```

Genetic algorithms: Pseudo-code

```
function GENETIC-ALGORITHM(population, fitness-function)
    returns an individual

    repeat

        initialize new-population with ∅

        for i=1 to size(population) do

            x = random-select(population,fitness-function)
            x = random-select(population,fitness-function)
            child = cross-over(x,y)
            mutate (child) with a small random probability
            add child to new-population

        population = new-population

    until some individual is fit enough or enough time has elapsed

    return the best individual in population w.r.t. fitness-function
```

Simulated Annealing

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  current ← problem.INITIAL
  for t = 1 to ∞ do
      T ← schedule(t)
      if T = 0 then return current
      next ← a randomly selected successor of current
      ΔE ← VALUE(current) – VALUE(next)
      if ΔE < 0 then current ← next      //for maximization
      else current ← next only with probability e^(−ΔE/T)
```