

```

def chainageAutSimple (faitsInitiaux, regles):
    Q = copy.deepcopy (faitsInitiaux)
    while len(Q) > 0:
        q = Q.pop(0)
        if q not in faits:
            ajouteFait(q)
            print q
        for r in regles:
            if (satisfaitUneCondition(r, q) and
                satisfaitConditions(r) and
                consequenceRegle(r) not in faits):
                Q.append (consequenceRegle(r))

```

consequ. de r ajoutée si:
 1/ fait q est une cond. de r
 2/ autres cond. de r sont satisfaites par base de faits actuels
 3/ conséquence de r pas déjà dans la base de faits

```

def unifier (proposition1, proposition2):
    if (proposition1 == E et proposition2 == E): return E
    else:
        testeAtome (proposition1) or testeAtome (proposition2):
            not testeAtome (proposition1): echange les des prop. (swap)
            proposition1 <= proposition2: return {}
        testeVariable (proposition1):
            if proposition1 in proposition2: return echec
            else:
                return construitSubstitution (proposition1, prop2)
        testeVariable (proposition2):
            return construitSubstitution (prop2, proposition1)
    return ECHEC

```

Ex: $E_1 = E_2$
 $E_1 = \text{unifier}(f_1, f_2)$
 $f_1 = \text{substitutVariables}(t_1, f_2)$
 $f_2 = \text{unifier}(g_1, g_2)$
 $g_1 = \text{substitutVariables}(t_2, f_2)$
 $g_2 = \text{unifier}(g_1, g_2)$
 $g_2 = \text{ECHEC}$: return ECHEC
 return unionSubstitution(E_1, E_2)

```

def filter (datum, pattern):
    (pattern == E) and datum == E:
        return E
    else:
        datum == E or pattern == E:
            return ECHEC
        testeAtome (pattern):
            (pattern == datum):
                return {}
            else:
                return ECHEC
        testeVariable (pattern):
            return construitSubstitution (pattern, datum)
        else:
            return ECHEC
        testeAtome (datum):
            return ECHEC
        else:
            E1 = datum E2
            E1 = datum E1: J
            E2 = pattern E2: J
            E1 = pattern E1: J
            E2 = filter (E1, E2)
            if E2 == ECHEC: return ECHEC
            E1 = E2
            E1 = substitueVariables (E1, E2)
            E2 = filter (E1, E2)
            if E2 == ECHEC: return ECHEC
            return unionSubstitutions (E1, E2)

```

```

def UnionSubstitution (substitution1, substitution2):
    if (testeSubstitution (subst1) and testeSubstitution (subst2)):
        keys = substitution2.keys()
        for x in keys:
            substitution1[x] = substitution2[x]
        return substitution1
    else:
        return ECHEC

```

Fact. Inverse: $T(x, y) = [-1, 1]$, $T(x, x) = 1$, $T(x, y) = 0$ si $x \neq y$
 $C_{fact} : x+y = y+x$ \rightarrow $T(x, y) = 1$ \rightarrow $C_{fact} = 1$
 $C_{fact} : x+y = x+y$ \rightarrow $T(x, y) = 1$ \rightarrow $C_{fact} = 1$

```

def patternMatching (datum, pattern, env):
    if env is None: env = {}
    if env == echec: return echec
    else:
        pattern = substitueVariables (pattern, env)
        return unionSubstitutions (env, filter (datum, pattern))

```

```

def substitueVariables (pattern, substitutions):
    if testeVariable (pattern) and pattern in substitutions:
        return retourneValeur (trouveSubstitution (pattern, substitutions))
    else:
        for x in pattern.E:
            pattern.append (substitueVariables (x, substitutions))
        return pattern

```

Modus Ponens $A \Rightarrow B; A \vdash B$

```

def faitSatisfaitUneCondition (fait, conditions):
    environnements = E
    cond-restantes = E
    resultat = Environnements, cond-restantes
    for cond in conditions:
        garde une liste de conditions non-remplies et obtient un environ. dérivé par patternMatching avec le fait et la condition
        cond-pas-remplies = copy.deepcopy (conditions)
        environnement = partitionnement (copy.deepcopy (fait), copy.deepcopy (cond))
        // si un environnement a été trouvé, ajoute et supprime la condition satisfait
        if (environnement != ECHEC):
            environnements.append (environnement)
            cond-pas-remplies.remove (cond)
            cond-restantes.append (cond-pas-remplies)
    return resultat

```

```

def patternMatching (prop1, prop2, env):
    if env is None: env = {}
    if env == echec: return echec
    else:
        prop1 = substitueVariables (prop1, env)
        prop2 = substitueVariables (prop2, env)
        return unionSubstitutions (env, unifier (prop1, prop2))

```

Théo PlanifPSC

- mutex de Propositions = 2 propositions ne peuvent être vraies en même temps: $q(x), d(x)$
- mutex d'opérateurs = Opérateurs ne peuvent pas s'exécuter en même temps
- contraintes initiales et finales...
- absence de cadre \equiv var d'état pas touchée par une action, alors sa valeur reste inchangée...
- consistance des besoins: unifying contraintes unaires
- consistance des arcs: pour chaque valeur admissible pour un need x_i , il existe des valeurs admissibles pour les autres needs x_j telle que chaque contrainte entre x_i et x_j est satisfaite.

```

def chainageAvantArrièreVariables (regles, faits-initiaux):
    Q = copy.deepcopy (faits-initiaux)
    while len(Q) > 0:
        q = Q.pop(0)
        if q not in faits: ajouteFait(q); print q
        for r in regles:
            resultat = faitSatisfaitUneCondition (q, conditionRegle(r))
            env = resultat E
            and-restant = resultat E
            for c in range (len (env)):
                env = env E
                cond = cond-restant E
                env2 = satisfait (conditions (cond, env))
                if len (env2) > 0:
                    instances = instantieVariables (consequenceRegle(r), env2)
                    for inst in instances:
                        if inst not in faits:
                            Q.append (inst)

```

```

def backTrack (R, toutesLesSolutions=None, initialiser=None):
    algo = "bt"
    global ITERATIONS; global SOLUTIONS
    if toutesLesSolutions == None:
        toutesLesSolutions = False
    if initialiser == None or initialiser == True:
        ITERATIONS = 0
        SOLUTIONS = E
        R.BSC.UBCONTRAINTES = 0
        initialiser = False
    ITERATIONS += 1
    afficheUBITERATIONS (algo, R)
    if R >= len (VARIABLES):
        solution = {}
        for var in VARIABLES:
            solution[var.nom] = var.valeur
            afficheSolution (algo, solution)
            SOLUTIONS.append (solution)
        if not toutesLesSolutions:
            return SOLUTIONS
    else:
        var = VARIABLES[R]
        for d in var.domaine:
            var.metAJourValeur (d)
            if consistenceAvecVarsPrecedentes(R):
                reste = backTrack (R+1, toutesLesSolutions, initialiser)
                if reste == ECHEC:
                    return reste
            var.metAJourValeur (None)
        return ECHEC

```

df & DFS $P(n) = P(n) \cdot P(n)$

```

def forwardChecking (R, toutesLesSolutions=None, initialiser=None):
    algo = "fc"
    global ITERATIONS; global SOLUTIONS
    if toutesLesSolutions == None:
        toutesLesSolutions = False
    if initialiser == None or initialiser == True:
        ITERATIONS = 0
        SOLUTIONS = E
        R.BSC.UBCONTRAINTES = 0
        for var in VARIABLES:
            var.initLabel()
            initialiser = False
        ITERATIONS += 1
        afficheUBITERATIONS (algo, R)
        if R >= len (VARIABLES):
            solution = {}
            for var in VARIABLES:
                solution[var.nom] = var.valeur
                afficheSolution (algo, solution)
                SOLUTIONS.append (solution)
            if not toutesLesSolutions:
                return SOLUTIONS
        else:
            dvoch // heuristique dvo
            var = VARIABLES[R]
            vieuxLabels = retourneLabels (R) // backup des labels avant de les modifier
            for v in var.label E:
                var.metAJourValeur (v)
                var.label = E
                if propagationAvecVarsSuivantes (R): // voir consistance sur variable R et au moins une autre var. non encore initialisée
                    reste = forwardChecking (R+1, toutesLesSolutions, initialiser)
                    if reste == ECHEC:
                        return reste
                var.metAJourValeur (vieuxLabels) // la valeur ne convient pas, restore avant d'essayer la suite...
            var.metAJourValeur (None)
            return ECHEC

```

```

def propagationAvecVarsSuivantes (R):
    for c in CONTRAINTES:
        if VARIABLES[E] in c.vars:
            for i in range (len (c.vars)):
                if VARIABLES[i] == R:
                    if c.propage (VARIABLES):
                        break
            else:
                return False

```

```

def rechercheOptimisee (depart, but, methode):
    Q = [R.B Recherche. Noeud (depart)]
    C = lib Recherche. Noeuds ()
    nbIteration = 1
    while len(Q) > 0:
        n = Q.pop(0)
        if n.estUneSolution (but):
            print "Solut. à l'iter", nbIter, ":"
            return n
        else:
            if (not C.contient (n) and methode is not "A*") or
                (C.contient (n) and C.contientementInferieur (n) and methode is "A*"):
                nbIteration += 1
                print "Iteration", nbIteration, ":", n
                C.ajouteNoeud (n)
                S = n.successeurs (but)
                Q = ajouteSuccesseurs (Q, S, methode)
    return echec

```

```

def variableOrdering (C):
    VARIABLES.sort (key = lambda x: x.tailleDuDomaine())
    // trier par taille croissante de la taille du domaine

```

```

def dvo (R):
    indexMin = chercheIndexMin (avecUnTableauLabel(R))
    P(H, I) = P(H|I) P(I)
    if indexMin != R:
        VARIABLES[E], VARIABLES[indexMin] = VARIABLES[indexMin], VARIABLES[E]

```

La consistance des besoins revient à supprimer du domaine toutes les valeurs qui violent les contraintes unaires.

```

def consistenceDesBesoins (self):
    for c in CONTRAINTES:
        if c.dimension (I) == 1:
            for d in c.refVar.domaine E:
                if not c.estValide (c.refVar, d):
                    c.refVar.domaine.remove (d)

```

```

def revise (self):
    modifiee = False
    for paire in [c.refVar1, self.refVar2], [c.refVar2, self.refVar1]:
        for x in paire E:
            if not c.estValide (paire E): // s: contrainte pas possible...
                paire E, domaine.remove (x)
            modifiee = True // revenir à nouveau car p-e invalide
        paire E, metAJourValeur (None) // autres contraintes
    return modifiee

```

Probas: $P(A|B) = \frac{P(A, B)}{P(B)} = \frac{P(B|A)P(A)}{P(B)} = \alpha P(B|A)P(A)$

$P(X|Y_1, \dots, Y_n) = \alpha P(X) \prod_{i=1}^n P(Y_i|X)$

$P(B) = P(B|A)P(A) + P(B|\neg A)P(\neg A)$

$P(B) = P(B|A, C) \frac{P(B|C)}{P(A|C)} = P(A|B, C) \frac{P(B|C)}{P(A|C)} = P(A|B) P(B|C) \cdot \frac{1}{\pi}$

$P(B) \leftarrow \pi P(B) \lambda(B)$

$\pi P(B) = P(A|B) \lambda(B) = P(B|C)$

SUPPORT VECTOR MACHINES

FN conjonctive: $q(A_1, A_2, \dots, A_n)$

HORN: $\neg(A_1, A_2, \dots, A_n) \vee C = T_1, T_2, \dots, T_n$

STRIPS: $\text{precond } S_i, \text{postcond } S_i$

Suppressions S_i, T_i

Propositions non-modifiables \rightarrow axiomes

calculer l'entropie totale pour l'attribut : $H(C|A) = H(C|A, \text{attribut})$
 def entropie (attribut, exemples): // attribut = indice de l'attribut
 // entropie pour chaque attribut:
 liste = [P.Aj (attribut, x, exemples) * H.C.Aj (attribut, x, exemples)
 for x in retourneValeursPossiblesAttribut (attribut, exemples)]
 res = sum (liste)
 print ("DB : entropie -> res : %s" % (res))
 return res

retourne l'entropie conditionnelle de l'attribut pour une certaine valeur cdd : $H(C|A_j) = H(C|A, \text{attribut} = \text{valeur})$
 def H.C.Aj (attribut, valeur, exemples):
 //ajoute les probas...
 return -sum ([calculer xlog x (P.Ci - Aj (c, attribut, valeur, exemples))
 for c in retourneClassesPossibles (exemples)])

calculer la proba $P(Ci|Uj) = P(\text{classe}, \text{valeur} | \text{attribut} = \text{valeur})$
 def P.Ci.Aj (classe, attribut, valeur, exemples):
 return nombreOccurences (classe, attribut, valeur, exemples) /
 float (nombreOccurences (attribut, valeur, exemples))

P.Aj
 def P.Aj (attribut, valeur, exemples):
 return nombreOccurences (attribut, valeur, exemples) / float (len (exemples))

def propage (self, var) : // E CLASS Contrainte Arbre Cadré (1) // E CLASS Contrainte
 dernierVar = None
 for var2 in [self.varPre, self.varPost] + self.varsOp
 {
 if var2.valeur == None:
 //dernierVar = None
 //dernierVar = None
 else : //il reste plus d'une variable non-instantiée
 return true
 }
 for p in dernierVar.labelEJ:
 if not self.estValable (dernierVar, p):
 {dernierVar.enleveDuLabel (p)}
 return len (dernierVar.label) > 0

def chargeDonnee (methode):
 global NOEUDS, METHODE
 METHODE = methode
 NOEUDS = [NOEUD (len (DOUVEES), EJ, [donnee] for
 donnee in DOUVEES)]

def distanceClusters (cluster1, cluster2):
 if METHODE == "single-link":
 {return min ([distance (donnee1, donnee2) for donnee1 in cluster1
 donnee2 in cluster2]) }
 elif METHODE == "complete-link":
 {return max (...)}
 else : print "Methode", methode, "inconnue"
 return None

def reviseClusters (l):
 p (len (NOEUDS) > 1 : return //si moins de 2 noeuds
 //sinon, trouve les 2 noeuds dont les clusters sont les plus similaires
 pairesDeNoeuds = [(NOEUDS [i], NOEUDS [j]) for i in range (len (NOEUDS)-1)
 for j in range (i+1, len (NOEUDS)-1)]
 paireMin = min (pairesDeNoeuds, key = lambda paire :
 distanceClusters (paire [0].cluster, paire [1].cluster))
 NOEUDS.remove (paireMin [0])
 NOEUDS.remove (paireMin [1])
 NOEUDS.append (NOEUD (len (NOEUDS)+1, [paireMin [0]+paireMin [1]]))

def fini (l):
 return len (NOEUDS) <= 1

associe chaque donnee au + proche noyau
 def formeClusters (noyaux):
 clusters = [noyau] for noyau in noyaux
 for donnee in DOUVEES
 {
 if donnee in noyaux : continue
 cluster = min (clusters, key = lambda c : distance (donnee, c [0]))
 //selectionne le cluster dont le noyau est le plus proche de la donnee
 cluster.append (donnee)
 }
 return clusters

def recentreNoyau (cluster):
 noyau = min (cluster, key = lambda x : sum ([distance (donnee, x) ** 2 for donnee in cluster]))
 cluster.remove (noyau)
 cluster.insert (0, noyau)
 return cluster

pour chargeDonnee : NOYAUX = listeNoyaux. Si none, on prend les k premières DOUVEES [0:k]
 comme noyau. CLUSTERS = [noyaux] for noyau in noyaux et les donnees sont
 mises dans le 1er cluster. Le noyau est : cluster [0] for cluster in CLUSTERS

def fini (l):
 noyaux = retourneNOYAUX (l)
 noyaux.sort ()
 vieuxNOYAUX.sort ()
 return noyaux == vieuxNOYAUX

def reviseClusters (l):
 global VIEUXNOYAUX, CLUSTERS
 noyaux = retourneNoyaux (l)
 VIEUXNOYAUX = copy.deepcopy (noyaux)
 CLUSTERS = formeClusters (noyaux)
 CLUSTERS = [recentreNoyau (cluster) for cluster
 in CLUSTERS]

Formules:
 • entropie de la classification : moyenne pondérée de toutes les entropies des sous-arbres
 $H(C|A) = \sum_{j=1}^n p(a_j) \cdot H(C|a_j)$
 • entropie du sous-arbre où A vaut a_j:
 $H(C|a_j) = - \sum_{i=1}^n p(C_i|a_j) \cdot \log_2 p(C_i|a_j)$
 • entropie de la classification C:
 $H(C) = - \sum_{i=1}^n p(C_i) \cdot \log_2 p(C_i)$

Algo ID3: Fonction ID3 (E, A)
 if E == {} then return Nil
 else
 if V ∈ E, classe(c) = c then
 return c
 else
 P ← attribut ∈ A qui réduit au plus l'entropie de la classe
 L ← { e ∈ E et P(e) = succès }
 R ← { e ∈ E et P(e) = échec }
 N ← noeu vide : N.P ← P
 N.left ← ID3 (L, A), N.right ← ID3 (R, A)
 return N

def PARTITION (exemples, attrib):
 res = [[] for i in range (len (exemples))]
 if retourneValeursPossiblesAttribut (attrib, exemples) == v
 {
 for u in retourneValeursPossiblesAttribut (attrib, exemples)
 }
 return res

Op: CONTRAINTES binaires:
 NAND, ==, >, <, !=
 " " " VAIRES:
 <, <=, ==, !=

Labeland : propagation du FC
 => CONSISTENCE DES ARC à chaque noe !
 DVD (pp Label), min-within, max-within
 // min-within : connectés
 // max-within : un plus petit ab de voir min-within

def construireArbreDecisionAux (exemples, attributs):
 print ("DB : construit Arbre Decision aux -> attributs : %s" % (attributs))
 if exemples == [] : return EJ //si plus d'exemples, on a fini
 else :
 {
 print ("DB : construit Arbre Decision aux -> mereclasse") //fin classe
 return Noeud (EJ, exemples, EJ)
 }
 else : //plus d'une classe, on continue à descendre dans l'arbre
 {
 attributPartition = meilleurAttribut (attributs, exemples) //choix de l'attribut de partition
 attributsRestants = copy.deepcopy (attributs)
 attributsRestants.remove (attributPartition)
 sousListes = partition (exemples, attributPartition) //fin en sousListes d'après attribut
 fils = map (lambda p : construireArbreDecisionAux (p, attributsRestants), sousListes) //construit les noeuds fils
 return Noeud (attributPartition, EJ, fils)
 }

def meilleurAttribut (attributs, exemples):
 { entropieAttributs = [entropie (a, exemples) for a in attributs]
 return attributs [retourneIndiceMinimum (entropieAttributs)] }

def construireArbreDecision (l):
 global classification
 classification = construireArbreDecisionAux (l, retourneClassesPossibles (l))
 print (classification)
 // la classification de nouveaux objets sera globale

def construireVariablesPropositionnelles (self, props):
 if self.no-etat > 0 :
 self.varInitiales = ETATS [self.no-etat-1].varFinales
 for prop in props:
 var = libPSC.Variable (prop + "etat" + str (self.no-etat-1), [True, False])
 self.varFinales [prop] = var
 self.varInitiales [prop] = var
 if self.no-etat == 0 :
 var = libPSC.Variable (prop + "etat 0", [True, False])
 VARS.append (var)
 self.varInitiales [prop] = var

def ImplementerMaterPropositionnelles (self):
 for m in self.probleme2.retourneMaterPropositionnelles():
 for e in ETATS:
 CVTS.append (Contrainte (libPSC.Constraint (e.varInit (m [0]), "NAND", " " " " " 1)
 if e.no-etat == (self.no-etats-1):
 CVTS.append (Contrainte (...)
 return VarsFinales ...)

exemple : [val-classe, val-attribut1, ..., val-attributN]
 attribut : [non-attribut, valeur-1, ..., valeurN]

