Chaînages avant et arrière

- Connaissances sous forme de clauses de Horn :
  - un symbole de proposition
  - $A_1 \wedge \cdots \wedge A_n \rightarrow B$, avec les A_i et B des symboles de proposition
- Base de connaissances : conjonction de clauses de Horn
- Règle de déduction : *Modus Ponen* (pour forme de Horn)

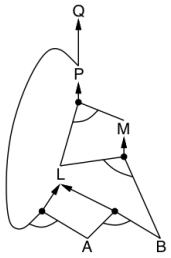
$$\frac{\alpha_1, \quad \ldots, \quad \alpha_n, \quad \alpha_1 \wedge \cdots \wedge \alpha_n \rightarrow \beta}{\beta}$$

- Même règle utilisée avec le chaînage avant (forward chaining) ou chaînage arrière (backward chaining)
- Ces algorithmes sont simples et s'exécutent en temps *linéaire* en taille de *KB*

Chaînage avant

- Chaque connaissance est une règle avec des prémisses et une conclusion
- Répéter jusqu'à saturation ou jusqu'à ce que l'expression à prouver soit trouvée
  - Activer chaque règle dont les prémisses sont satisfaites par des éléments de la base de connaissances *KB*
  - Ajouter la conclusion à la base *KB*
- Exemple : base *KB* représentation sous forme d'arbre ET-OU

$P \rightarrow Q$
$L \wedge M \rightarrow P$
$B \wedge L \rightarrow M$
$A \wedge P \rightarrow L$
$A \wedge B \rightarrow L$
A
B



Chaînage arrière

- Idée : pour prouver q on procède en arrière à partir de q
- Algorithme :
  - vérifier si q est déjà connu (q est dans *KB*)
  - sinon prendre une règle ayant q en conclusion et prouver par chainage arrière toutes les prémisses de la règle
- Éviter des boucles : vérifier si un nouveau but est déjà dans la pile de buts
- Éviter des tâches répétées : vérifier si un nouveau but a été déjà prouvé vrai/faux

Chaînage avant vs. arrière

- Le chaînage avant est piloté par les données (data-driven), un agent déduit des propriétés et des catégories à partir des séquence perceptives
- Il peut effectuer des tâches non pertinentes par rapport à l'objectif
- Le chaînage arrière est dirigé par les objectifs (goal-driven), approprié à la résolution de problème
- Le chaînage arrière est la base de la programmation logique, par exemple Prolog
- La complexité du chaînage arrière peut être *inférieur* au temps linéaire en taille de la base de connaissances

Construction de modèles

- Preuve de satisfaisabilité d'un ensemble de connaissances
- $KB \models \alpha$ si et seulement si $KB \wedge \neg\alpha$ n'a pas de modèle
- Algorithmes effectifs :
  - à base de backtracking : algorithme DPLL (Davis, Putnam, Logemann, Loveland)
  - à base de la recherche locale : algorithme Walksat

Algorithme DPLL

- Prendre comme entrée un ensemble de clauses
- Objectif : vérifier s'il existe une instanciation des symboles qui rend toutes les clauses vraies
- Algorithme : énumération récursive, à chaque étape
  - choisir un symbole et lui affecter une valeur de vérité
    - ⋆ symbole pur : symbole ayant le même signe en toute occurrence, ex A et B sont purs

$$(A \vee \neg B) \wedge (\neg B \vee \neg C) \wedge (C \vee A)$$

    - ⋆ clause unitaire : clauses contenant un seul symbole
    - ⋆ un symbole quelconque
  - évaluer les clauses avec l'affectation partielle
    - ⋆ une clause est vraie si au moins un littéral est vrai
    - ⋆ si toute clause est vraie alors l'affectation partielle est un modèle
    - ⋆ si une clause est fausse alors l'affectation partielle n'est pas un modèle

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 *true*
  P, *value* ← FIND-PURE-SYMBOL(*symbols*, *clauses*, *model*)
  if P is non-null then return DPLL(*clauses*, *symbols*–P, [$P =$ *value*|*model*])
  P, *value* ← FIND-UNIT-CLAUSE(*clauses*, *model*)
  if P is non-null then return DPLL(*clauses*, *symbols*–P, [$P =$ *value*|*model*])
  P ← FIRST(*symbols*) ; *rest* ← REST(*symbols*)
  return DPLL(*clauses*, *rest*, [$P =$ *true*|*model*]) or DPLL(*clauses*, *rest*, [$P =$ *false*|*model*])

La recherche locale pour trouver des modèles

- Les algorithmes de recherche locale peuvent s'appliquer avec une bonne fonction d'évaluation
- Les états sont des interprétations possibles, la recherche locale cherche une interprétation qui est un modèle
- Algorithme WalkSAT : à chaque étape
  - choisir une clause non satisfaite et un symbole
  - changer la valeur du symbole

Algorithme WalkSAT

function WALKSAT(*clauses*, *p*, *max-flips*) returns a satisfying model or *failure*
inputs: *clauses*, a set of clauses in propositional logic
p, the probability of choosing to do a "random walk" move, typically around 0.5
max-flips, number of flips allowed before giving up
model ← a random assignment of *truefalse* to the symbols in *clauses*
for $i = 1$ to *max-flips* do
if *model* satisfi es *clauses* then return *model*
clause ← a randomly selected clause from *clauses* that is false in *model*
with probability *p* flip the value in *model* of a randomly selected symbol from *clause*
else flip whichever symbol in *clause* maximizes the number of satisfi ed clauses
return *failure*

Propriétés des algorithmes de recherche locale

- Efficaces pour trouver un modèle s'il en existe
- Lors de terminaison avec échec, 2 cas possibles :
  - on n'a pas fait suffisamment d'itérations
  - il n'existe vraiment pas de modèle

  Ces algorithmes ne détectent pas l'insatisfaisabilité
- Un agent utilisant un algorithme de recherche locale opterai une stratégie prudente : comment ?