CheatSheet for Foundation of AI (midterm)

by Tanimal@NTU

Adversarial Search

Minimax Search (with alpha-beta purning)

α: MAX's best option on path to root
β: MIN's best option on path to root

```
def max-value(state, α, β):
  initialize v = -∞
  for successor of state:
    v = max(v, min-value(successor, α, β))
    if v > β: return v
    α = max(α, v)
  return v
```

```
def min-value(state, α, β):
  initialize v = +∞
  for successor of state:
    v = min(v, max-value(successor, α, β))
    if v < α: return v
    β = min(β, v)
  return v
```

Depth-limited Search
- search only to a limited path in the tree
- use evaluation function for non-terminal positions

Expectimax Search
- compute the average score under optimal play
- replace Min node with Chance node, where the outcome is uncertain
- perform better when against a random adversary

Monte Carlo Tree Search (MCTS)
- do N rollout (simple, fast policy) from each child of the root, and count win-rate
- pick the child with the best win-rate, only explore on that subtree

MCTS 2.0: UCB for Tree
- repeat until out of time:
  - given current search tree, recursively apply UCB to choose a path down to a leaf node n
  - add a new child c to n, and run a rollout from c
  - update the win counts from c back up to the root
- choose the action leading to the child with highest N (# rollouts)

Propositional Logic

- agent learn from a knowledge base (set of sentences in a formal language)
- agent can then inference new facts from the known facts (KB)
- can be used for localization (where am i), mapping (what's the environment like), planning

Logic
- Syntax: what sentences are allowed?
- World/Model: an assignment of literals
- Semantics: which sentences are true in which worlds?

P ∧ Q
{P=true, Q=false}
P ∧ Q means false

Propositional Logic Syntax
- **¬α** is true iff α is false
- **α ∧ β** is true iff both α and β are true
- **α ∨ β** is true iff at least one in α and β is true
- **α ⇒ β** is true iff α is false or β is true
- **α ⇔ β** is true iff α and β have the same value

Inference
- Entailment: α |= β iff in every world where α is true, β is also true
- Proofs: an process to demonstrate entailment
  - *model-checking*: loop through every possible world
  - *theorem-proving*: use inference rules to derive new sentences

Forward chaining

```
function FORWARD-CHAINING(KB, q)
  count ← a table, where count[c] is the number of symbols in c's premise
  inferred ← a table, where inferred[s] is initially false for all s
  agenda ← a queue of symbols, initially symbols known to be true in KB
  while agenda is not empty do
    p ← Pop(agenda)
    if p = q then return true
    if inferred[p] = false then
      inferred[p] ← true
      for clause c in KB where p is in c.premise do
        decrement count[c]
        if count[c] = 0 then add c.conclusion to agenda
  return false
```

CNF
- a conjunction of clauses, each clause is a disjunction of literals
- distributivity:
  - (A ∨ B) ∧ C = (A ∧ C) ∨ (B ∧ C)
  - (A ∧ B) ∨ C = (A ∨ C) ∧ (B ∨ C)
  - ¬(A ∧ B) = (¬A ∨ ¬B)
  - ¬(A ∨ B) = (¬A ∧ ¬B)

Reduction to CNF
1. replace α ⇔ β with (α ⇒ β) ∧ (β ⇒ α)
2. replace α ⇒ β with β ∨ ¬α
3. distribute ¬ to lower levels
4. make ∨ lower than ∧

Satisfiability
- a sentence is **valid** if it is true in every world
- a sentence is **satifiable** if it is true in at least one world

DPLL
- pure symbol: always appears with same "sign" in all clauses
- unit clause: contains exactly one unassign symbol

```
function DPLL(clauses, symbols, partmodel={})
  if every clause in clauses is true in partmodel then return true
  if some clause in clauses is false in partmodel then return false
  clauses ← every clause in clauses that is not true (or false)

  P, value ← FIND-PURE-SYMBOL(symbols, clauses, partmodel)
  if P is non-null then
    return DPLL(clauses, symbols – {P}, partmodel ∪ {P=value})

  P, value ← FIND-UNIT-CLAUSE(clauses, partmodel)
  if P is non-null then
    return DPLL(clauses, symbols – {P}, partmodel ∪ {P=value})

  P ← First(symbols); rest ← Rest(symbols)
  return (DPLL(clauses, rest, partmodel ∪ {P=true}) or
          DPLL(clauses, rest, partmodel ∪ {P=false}))
```

Uniformed Search

Search Tree
- each node is a plan
- fringe = nodes under consideration
- decide which fringe nodes to expand
- Complete: Guarante to find a solution if exists
- Optimal: Guarante to find the least cost path

Depth-First Search
- expand deepest node first
- fringe is a LIFO stack
- O(b^m) time; O(bm) space
- complete if no cycle; not optimal

Breadth-First Search
- expand shallowest node first
- Fringe is a FIFO queue
- O(b^s) time; O(b^s) space
- complete; optimal if costs are 1

Uniform-Cost Search
- expand cost less node first
- effective depth d = C*/e. where C* is solution cost, e is min arc cost
- O(b^d) time, O(b^d) space
- complete and optimal

Informed Search

Heuristics function h(x): estimate how close a state is to a goal
- Admissibility: h(x) <= cost(x to Goal)
- Consistency: h(x - y) <= cost(x to y)

Relaxed Problem
- a simplified version of original problem by remove some constraints or add new operations
- solution can be heuristic for original problem

A* Search
- f(n) = path cost + goal proximity = g(n) + h(n)
- expand node with min f(n)
- optimal if heuristic addmissible (tree), consistent (graph)

```
function TREE-SEARCH(problem, fringe)
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
    if fringe is empty then return failure
    node ← REMOVE-FRONT(fringe)
    if GOAL-TEST([problem], STATE[node]) then return node
    for child-node in EXPAND(STATE[node], problem) do
      fringe ← INSERT(child-node, fringe)
```

```
function GRAPH-SEARCH(problem, fringe)
  closed ← an empty set
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
    if fringe is empty then return failure
    node ← REMOVE-FRONT(fringe)
    if GOAL-TEST([problem], STATE[node]) then return node
    if STATE[node] is not in closed then
      add STATE[node] to closed
      for child-node in EXPAND(STATE[node], problem) do
        fringe ← INSERT(child-node, fringe)
```

Constraint Satisfaction Problem (CSP)

- *state* is defined by *variables* (each city)
- each variable has a *domain* ({red, green, blue})
- *constraints* specify allowable assignment for variables (WA != NT)
- *solutions* are assignments satisfying all constraints

Constraint Graph: node = variable; arc = constraint

Backtracking Search
DFS + ordering variables + check constraints as you go

```
function RECURSIVE-BACKTRACKING(assignment, csp)
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(Variables[csp], assignment, csp)
  for value in ORDER-DOMAIN-VALUES(var, assignment, csp)
    if value is consistent with assignment given Constraints[csp] then
      add {var = value} to assignment
      result ← RECURSIVE-BACKTRACKING(assignment, csp)
      if result != failure then return result
      remove {var = value} from assignment
  return failure
```

Ordering
- Minimum Remaining Values: choose the variable with the fewest legal left values in its domain
- Least Constraining Value: choose the value that rules out the fewest values in the remaining variables

Filtering
- Froward Checking: cross off values in neighbor when assign value to node
- Constraint Propagation (AC-3): If X loses a value, neighbors of X need to be recheck consistency
- Arc Consistency: arc X > Y is consistent iff for every x in the tail there is some y could be assigned
- K-Consistency: For each k nodes, any consistent assignment to k-1 nodes we can find allowable value for the last node

Tree-Structured CSP
- graph with no loop, can be solved in O(nd^2)
- algorithm:
  a. choose a root variable → order variables so that parents precede children
  b. remove backward: for N:2, removeInconsistent(Parent(X), X)
  c. assign forward: for 1:N, assign X with value consistent with Parent(X)
- cutset conditioning: init a set of variables such that the remaining constraint graph is a tree

Min-Conflicts Algorithm
- a local search method: it's not optimal
- start from all variables assigned, while not solved, randomly select any conflicted variables, then reassign value that violates fewest constraints

Local Search

- start from complete assignment (may not legal)
- reassign better value iteratively
- not optimal

Hill-Climbing: start whatever, keep moving to best neighbor state until no neighbor is better

Simulated Annealing: high temperature means more bad moves allowed, try to escape local minimum; temperature reduced by time

Local Beam Search: run K local search simultaneously, select best K successors from all successors generated to be the new current states

Bayes Net

- *Joint Distribution*: specify a real number for each assignment over a set of random variables
- *Marginal Distribution*: sub-tables of Joint with one random variable
- *Conditional Probabilities*: P(x|y) = P(x, y) / P(y)
- *Normalizing*: scale the probabilities making the sum be 1

Bayes' Rule
- P(x|y) = P(y|x)P(x) / P(y)
- two variables are independent if P(x, y) = P(x)P(y)

Bayes' Net
- a graph (DAG) to describe a set of random variables and their conditional independences
- Node: variables (with domains); can be observed or unobserved
- Arc: conditional independence
- CPT: describe node's conditional probability under each combination of it's parent nodes

Probabilistic Inference
- compute a probability of a query variable, given evidences
- *Exact inference*: sums of products of conditional probabilities from the BN

Enumeration
1. select entries consistent with the evidence
2. sum out hidden variables
3. normalize

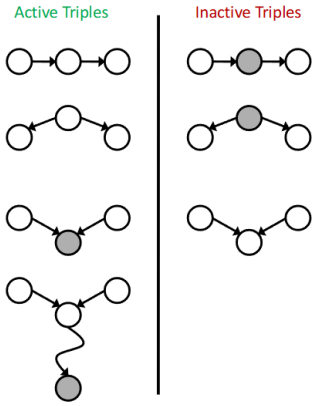
Variable Elimination
1. pointwise product:
   - P(B|A) x P(A) = P(A,B)
   - P(U,V) x P(V,W) x P(W,X) = P(U,V,W,X)
2. summing out a variable
   - \sum P(A,B) = P(A,b) + P(A, ¬b) = P(A)
3. interleave above steps

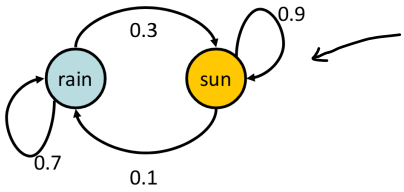
Chains

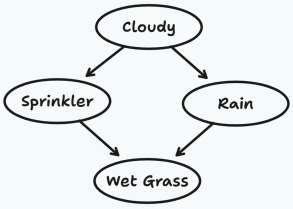
X indep from Z?		not given Y	given Y
Causal chains	**X > Y >Z**	No	Yes
Common causes	**X < Y > Z**	No	Yes
Common effect	**X > Y < Z**	Yes	No

D-Separation
- X and Y conditionally independent given evidence variables {Z}?
- **independence** if no active paths (all inactive paths)
- **active path** if no inactive triples (all active triples)

Markov Blanket
- A's parents, childrens, and childrens' other parents
- If Markov blanket of A is observed, then A is conditional independent from all other variables







Sampling

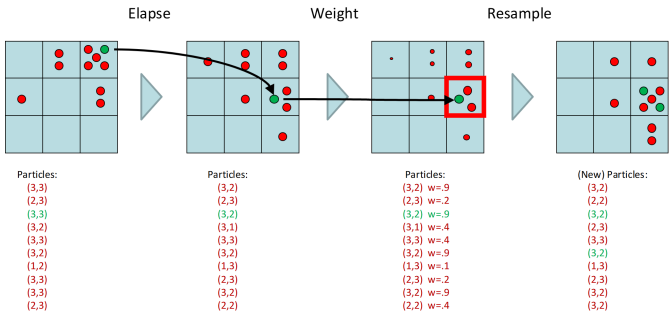
- a approximate inference method, used when 1) unknown distribution, 2) too expensive
- draw N samples from a sampling distribution S
- compute an approximate posterior probability, show this converges to the true probability P

Prior Sampling
- e.g. we want Joint Distribution of (C, S, R, W)
  - sample xi from P(Xi | Parent(Xi)) ⇒ C from P(C), S from P(S|C), W from P(W|S,R)
  - return (x1, x2, ..., xn) ⇒ (C=+c, S=+s, R=-R, w=+w)
  - repeat the process to get a collection of (x1, x2, ..., xn)

Rejection Sampling
- e.g. we want P(C|+s)
  - sample as in Prior Sampling
  - but ignore sample inconsistent with given evidences (i.e. S != +s)

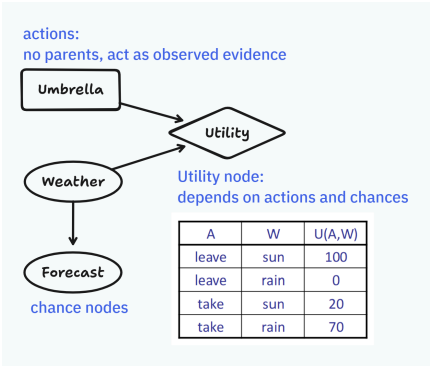
Likelihood Weighting
- for Rejection Sampling, we might reject too much sample if evidences are unlikely
- e.g. we want P(C|+s)
  - sample as in Prior Sampling
  - but fix the value of evidence variables (keep S=+s)
  - and set the weight as w = w * P(xi | Parent(Xi)) ⇒ w = w * P(+s|C)
  - return (x1, x2, ..., xn), w

Gibbs Sampling
- for Likelihood Weighting, evidence has no effect on upstream variables
- e.g. we want P(S|+r)
  - fix evidence R=+r
  - initialize other variables randomly
  - repeat:
    - choose one non-evidence variable X
    - resample X from P(X | all other variables) = P(X | Markov(X))



Decision Networks

- MEU: choose the action maximize the expected utility given the evidence
- bayes net with actions and utility



A	W	U(A,W)
leave	sun	100
leave	rain	0
take	sun	20
take	rain	70

- Utility is defined by Weather and Umbrella
- assume the P(W=sun) = 0.7
- we can calculate the expected utility of action
  - take: 20 * 0.7 + 70 * 0.3 = 35
  - leave: 100 * 0.7 + 0 = 70 ⇒ optimal decision
- MEU(ø) = 70

- Now consider if Forecast=bad
- P(W|F) = P(F|W)P(W) / \sum_w P(F|w)P(w)
- P(W|F=bad) = 0.34
- MEU(F=bad) = 53 ⇒ take

Value of Information
- value of information = expect grow in MEU from this information
- example
  - without Forecast: MEU(ø) = 70
  - with Forecast: MEU(F=bad) = 53, MEU(F=good) = 95
  - assume P(F=bad) = 0.59, P(F=good) = 0.41
  - VPI(F|ø) = 0.59 * 53 + 0.41 * 95 - 70 = 7.8

$$\text{VPI}(E'|e) = (\sum_{e'} P(e'|e)\text{MEU}(e, e')) - \text{MEU}(e)$$

  - E: evidences we already know
  - E': new information
- VPI is nonadditive, order-independent

Markov Models

- value of X at a given time is called state
- transition model P(X_t | X_t-1) sepcify how the state evolves over time
- transition probabilities are same all the time, future is independent of the past given present

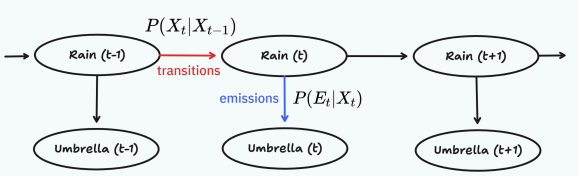
Markov Chain
- P(X1 = sun) = 1
- P(X2 = sun) = 0.9 * 1 + 0.3 * 0 = 0.9
- **Mini-Forward Algorithm**: what's P(X) on some day t?

$$P(x_t) = \sum_{x_{t-1}} P(x_t|x_{t-1})P(x_{t-1})$$

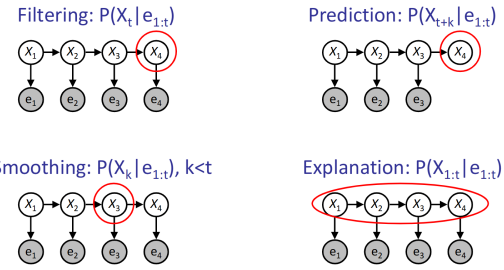
- influence of initial state decrease by time
- the distribution ultimately converge in **stationary distribution**, where $P_\infty(X) = P_{\infty+1}(X)$

Hidden Markov Models (HMM)

- hide Markov Chain over states X, we observe outputs at each time step
- future depends on past via the present
- current observation conditionally independent of all else given current state
- an HMM is defined by initial state + transitions + emissions



Inference Tasks



Filtering
- given all observations up to current,
  we want to "filter" out the distribution of current state $B_t(X) = P(X_t|e_{1:t})$

The Forward Algorithm
- passage of time: base on previous states to speculate current state
- observation: use new observation to adjust prediction on current state

$$\begin{aligned} P(x_t|e_{1:t}) &\propto_{X_t} P(x_t, e_{1:t}) \\ &= \sum_{x_{t-1}} P(x_{t-1}, x_t, e_{1:t}) \\ &= \sum_{x_{t-1}} P(x_{t-1}, e_{1:t-1})P(x_t|x_{t-1})P(e_t|x_t) \\ &= P(e_t|x_t) \sum_{x_{t-1}} P(x_t|x_{t-1})P(x_{t-1}, e_{1:t-1}) \end{aligned}$$

Particle Filtering
- representation: use N particle to represent possible current state
- elapse time: each particle move to new state by sample from transition model
- observe: if particle's new state more align to current observation, we give it higher weight
- resample: sample N new particle from weighted distribution