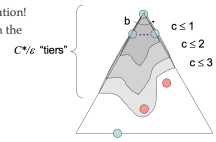
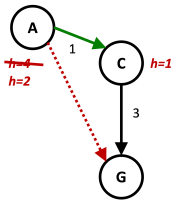
- Reflex agents:
  - Choose action based on current percept (and maybe memory)
  - May have memory or a model of the world's current state
  - Do not consider the future consequences of their actions
  - Consider how the world IS
- Planning agents:
  - Ask "what if"
  - Decisions based on (hypothesized) consequences of actions
  - Must have a model of how the world evolves in response to actions
  - Must formulate a goal (test)
  - Consider how the world WOULD BE

Uniform Cost Search (UCS) Properties

- What nodes does UCS expand?
  - Processes all nodes with cost less than cheapest solution!
  - If that solution costs C^* and arcs cost at least ε, then the "effective depth" is roughly C^*/ε
  - Takes time $O(b^{C^*/\varepsilon})$ (exponential in effective depth)
- How much space does the fringe take?
  - Has roughly the last tier, so $O(b^{C^*/\varepsilon})$
- Is it complete?
  - Assuming best solution has a finite cost and minimum arc cost is positive, yes!
- Is it optimal?
  - Yes! (Proof next lecture via A*)



Consistency of Heuristics



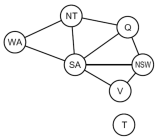
- Main idea: estimated heuristic costs ≤ actual costs
  - Admissibility: heuristic cost ≤ actual cost to goal
    - h(A) ≤ actual cost from A to G
  - Consistency: heuristic "arc" cost ≤ actual cost for each arc
    - h(A) – h(C) ≤ cost(A to C)
- Consequences of consistency:
  - The f value along a path never decreases
    - h(A) ≤ cost(A to C) + h(C)
  - A* graph search is optimal

- Fact 1: In tree search, A* expands nodes in increasing total f value (f-contours)
- Fact 2: For every state s, nodes that reach s optimally are expanded before nodes that reach s suboptimally
- Result: A* graph search is optimal
- Tree search:
  - A* is optimal if heuristic is admissible
  - UCS is a special case (h = 0)
- Graph search:
  - A* optimal if heuristic is consistent
  - UCS optimal (h = 0 is consistent)
- Consistency implies admissibility

```
function TREE-SEARCH(problem, fringe) return a solution, or failure
  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)
    end
  end
function GRAPH-SEARCH(problem, fringe) return a solution, or failure
  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)
      end
  end
```

Constraint Graphs

- Binary CSP: each constraint relates (at most) two variables
- Binary constraint graph: nodes are variables, arcs show constraints
- General-purpose CSP algorithms use the graph structure to speed up search. E.g., Tasmania is an independent subproblem!



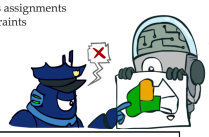
Example: N-Queens

- Formulation 1:
  - Variables: X_{ij}
  - Domains: $\{0, 1\}$
  - Constraints

$$\forall i,j,k\ \ (X_{ij}, X_{ik}) \in \{(0,0),(0,1),(1,0)\}$$
$$\forall i,j,k\ \ (X_{ij}, X_{kj}) \in \{(0,0),(0,1),(1,0)\}$$
$$\forall i,j,k\ \ (X_{ij}, X_{i+k,j+k}) \in \{(0,0),(0,1),(1,0)\}$$
$$\forall i,j,k\ \ (X_{ij}, X_{i+k,j-k}) \in \{(0,0),(0,1),(1,0)\}$$
$$\sum_{i,j} X_{ij} = N$$

Backtracking Search

- Backtracking search is the basic uninformed algorithm for solving CSPs
- Idea 1: One variable at a time
  - Variable assignments are commutative, so fix ordering -> better branching factor!
  - I.e., [WA = red then NT = green] same as [NT = green then WA = red]
  - Only need to consider assignments to a single variable at each step
- Idea 2: Check constraints as you go
  - I.e. consider only values which do not conflict previous assignments
  - Might have to do some computation to check the constraints
  - "Incremental goal test"
- Depth-first search with these two improvements is called *backtracking search* (not the best name)
- Can solve n-queens for n ≈ 25

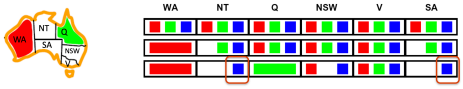


```
function BACKTRACKING-SEARCH(csp) returns solution/failure
  return RECURSIVE-BACKTRACKING({ }, csp)
function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
    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
```

- Backtracking = DFS + variable-ordering + fail-on-violation
- What are the choice points?

Filtering: Constraint Propagation

- Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:



- NT and SA cannot both be blue!
- Why didn't we detect this yet?
- *Constraint propagation:* reason from constraint to constraint

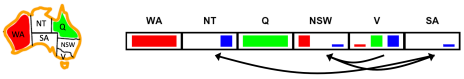
```
function AC-3(csp) returns the CSP, possibly with reduced domains
  inputs: csp, a binary CSP with variables {X1, X2, ..., Xn}
  local variables: queue, a queue of arcs, initially all the arcs in csp
  while queue is not empty do
    (Xi, Xj) ← REMOVE-FIRST(queue)
    if REMOVE-INCONSISTENT-VALUES(Xi, Xj) then
      for each Xk in NEIGHBORS[Xi] do
        add (Xk, Xi) to queue

function REMOVE-INCONSISTENT-VALUES(Xi, Xj) returns true iff succeeds
  removed ← false
  for each x in DOMAIN[Xi] do
    if no value y in DOMAIN[Xj] allows (x,y) to satisfy the constraint Xi ↔ Xj
      then delete x from DOMAIN[Xi];  removed ← true
  return removed
```

- Runtime: $O(n^2d^3)$, can be reduced to $O(n^2d^2)$
- … but detecting all possible future problems is NP-hard – why?

Arc Consistency of an Entire CSP

- A simple form of propagation makes sure all arcs are consistent:

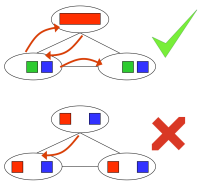


- Important: If X loses a value, neighbors of X need to be rechecked!
- Arc consistency detects failure earlier than forward checking
- Can be run as a preprocessor or after each assignment
- What's the downside of enforcing arc consistency?

Remember: Delete from the tail!

Limitations of Arc Consistency

- After enforcing arc consistency:
  - Can have one solution left
  - Can have multiple solutions left
  - Can have no solutions left (and not know it)



- Arc consistency still runs inside a backtracking search!

[Demo: coloring -- forward checking]
[Demo: coloring -- arc consistency]

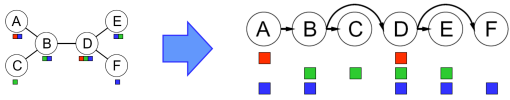
- 1-Consistency (Node Consistency): Each single node's domain has a value which meets that node's unary constraints
- 2-Consistency (Arc Consistency): For each pair of nodes, any consistent assignment to one can be extended to the other
- K-Consistency: For each k nodes, any consistent assignment to k-1 can be extended to the kth node.

Strong K-Consistency

- Strong k-consistency: also k-1, k-2, … 1 consistent
- Claim: strong n-consistency means we can solve without backtracking!
- Why?
  - Choose any assignment to any variable
  - Choose a new variable
  - By 2-consistency, there is a choice consistent with the first
  - Choose a new variable
  - By 3-consistency, there is a choice consistent with the first 2
  - …

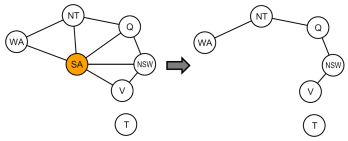
Ordering: Minimum Remaining Values
Ordering: Least Constraining Value

- Suppose a graph of n variables can be broken into subproblems of only c variables:
  - Worst-case solution cost is $O((n/c)(d^c))$, linear in n
  - E.g., n = 80, d = 2, c =20
  - 2^{80} = 4 billion years at 10 million nodes/sec
  - $(4)(2^{20})$ = 0.4 seconds at 10 million nodes/sec
- Algorithm for tree-structured CSPs:
  - Order: Choose a root variable, order variables so that parents precede children



  - Remove backward: For i = n : 2, apply RemoveInconsistent(Parent(X_i),X_i)
  - Assign forward: For i = 1 : n, assign X_i consistently with Parent(X_i)
- Runtime: $O(n\ d^2)$ (why?)

Nearly Tree-Structured CSPs



- Conditioning: instantiate a variable, prune its neighbors' domains
- Cutset conditioning: instantiate (in all ways) a set of variables such that the remaining constraint graph is a tree
- Cutset size c gives runtime $O((d^c)(n-c)d^2)$, very fast for small c

Tree Decomposition

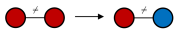
Idea: create a tree-structured graph of mega-variables

Each mega-variable encodes part of the original CSP

Subproblems overlap to ensure consistent solutions

Iterative Algorithms for CSPs

- Local search methods typically work with "complete" states, i.e., all variables assigned
- To apply to CSPs:
  - Take an assignment with unsatisfied constraints
  - Operators *reassign* variable values
  - No fringe! Live on the edge.



- Algorithm: While not solved,
  - Variable selection: randomly select any conflicted variable
  - Value selection: min-conflicts heuristic:
    - Choose a value that violates the fewest constraints
    - I.e., hill climb with h(x) = total number of violated constraints

Local search: improve a single option until you can't make it better (no fringe!)

Simulated Annealing

Theoretical guarantee:
- Stationary distribution (Boltzmann): $P(x) \propto e^{E(x)/T}$
- If T decreased slowly enough, will converge to optimal state!

Proof sketch
- Consider two adjacent states x, y with $E(y) > E(x)$ [high is good]
- Assume $x \to y$ and $y \to x$ and outdegrees $D(x) = D(y) = D$
- Let $P(x)$, $P(y)$ be the equilibrium occupancy probabilities at T
- Let $P(x \to y)$ be the probability that state x transitions to state y

Local beam search

Basic idea:
- K copies of a local search algorithm, initialized randomly
- For each iteration
  - Generate ALL successors from K current states
  - Choose best K of these to be the new current states

Or, K chosen randomly with a bias towards good ones

Why is this different from K local searches in parallel?
- The searches *communicate*! "Come over here, the grass is greener!"

How efficient is minimax?

- Just like (exhaustive) DFS
- Time: $O(b^m)$
- Space: O(bm)

Alpha-Beta Implementation

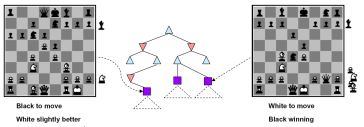
α: MAX's best option on path to root
β: MIN's best option on path to root

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

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

Evaluation Functions

- Evaluation functions score non-terminals in depth-limited search



- Typically weighted linear sum of features:
  - EVAL(s) = $w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$
  - E.g., $w_1 = 9$, $f_1(s)$ = (num white queens – num black queens), etc.
- Or a more complex nonlinear function (e.g., NN) trained by self-play RL

Expectimax Pseudocode

```
def value(state):
    if the state is a terminal state: return the state's utility
    if the next agent is MAX: return max-value(state)
    if the next agent is EXP: return exp-value(state)
```

```
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor))
    return v
```

```
def exp-value(state):
    initialize v = 0
    for each successor of state:
        p = probability(successor)
        v += p * value(successor)
    return v
```

Bayes' Rule

- Two ways to factor a joint distribution over two variables:

$$P(x,y) = P(x|y)P(y) = P(y|x)P(x)$$

That's my rule!

- Dividing, we get:

$$P(x|y) = \frac{P(y|x)}{P(y)}P(x)$$

- Why is this at all helpful?
  - Lets us build one conditional from its reverse
  - Often one conditional is tricky but the other one is simple
  - Foundation of many systems we'll see later (e.g. ASR, MT)
- In the running for most important AI equation!

Conditional Independence

- ***Conditional independence*** is our most basic and robust form of knowledge about uncertain environments.
- *X* is conditionally independent of *Y* given *Z* if and only if:

  ∀x,y,z *P(x | y, z) = P(x | z)*

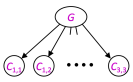
  or, equivalently, if and only if

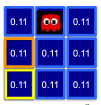
  ∀x,y,z *P(x, y | z) = P(x | z) P(y | z)*

Graphical Model Notation

- Nodes: variables (with domains)
  - Can be assigned (observed) or unassigned (unobserved)

- Arcs: interactions
  - Indicate "direct influence" between variables
  - Formally: absence of arc encodes conditional independence (more later)



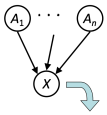


Bayes' Net Semantics

- A set of nodes, one per variable X
- A directed, acyclic graph
- A conditional distribution for each node
  - A collection of distributions over X, one for each combination of parents' values

    $$P(X|a_1 \ldots a_n)$$

  - CPT: conditional probability table
  - Description of a noisy "causal" process



$$P(X|A_1 \ldots A_n)$$

A Bayes net = Topology (graph) + Local Conditional Probabilities

- X and Y are independent if

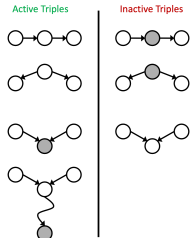
  $$\forall x,y \;\; P(x,y) = P(x)P(y) \;\; \text{----}\rightarrow \;\; X \perp\!\!\!\perp Y$$

- X and Y are conditionally independent given Z

  $$\forall x,y,z \;\; P(x,y|z) = P(x|z)P(y|z) \;\; \text{----}\rightarrow \;\; X \perp\!\!\!\perp Y | Z$$

Active / Inactive Paths

- Question: Are X and Y conditionally independent given evidence variables {Z}?
  - Yes, if X and Y "d-separated" by Z
  - Consider all (undirected) paths from X to Y
  - No active paths = independence!
- A path is active if each triple is active:
  - Causal chain A -> B -> C where B is unobserved (either direction)
  - Common cause A <- B -> C where B is unobserved
  - Common effect (aka v-structure)
    A -> B <- C where B *or one of its descendants* is observed
- All it takes to block a path is a single inactive segment



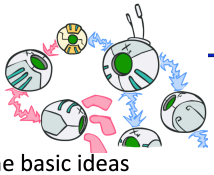
D-Separation

- Query: $X_i \perp\!\!\!\perp X_j | \{X_{k_1}, \ldots, X_{k_n}\}$?
- Check all (undirected!) paths between X_i and X_j
  - If one or more active, then independence not guaranteed

    $$X_i \not\perp\!\!\!\perp X_j | \{X_{k_1}, \ldots, X_{k_n}\}$$

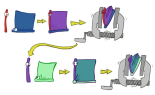
  - Otherwise (i.e. if all paths are inactive), then independence is guaranteed

    $$X_i \perp\!\!\!\perp X_j | \{X_{k_1}, \ldots, X_{k_n}\}$$

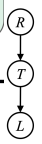


Variable elimination: The basic ideas

- Move summations inwards as far as possible
  - $P(B \mid j, m) = \alpha \sum_{e,a} P(B)\, P(e)\, P(a|B,e)\, P(j|a)\, P(m|a)$
  - $= \alpha\, P(B) \sum_e P(e) \sum_a P(a|B,e)\, P(j|a)\, P(m|a)$



- Do the calculation from the inside out
  - I.e., sum over *a* first, then sum over *e*
  - Problem: *P(a|B,e)* isn't a single number, it's a bunch of different numbers depending on the values of *B* and *e*
  - Solution: use arrays of numbers (of various dimensions) with appropriate operations on them; these are called ***factors***



$$P(L) = ?$$

- Inference by Enumeration

$$= \sum_t \sum_r P(L|t)P(r)P(t|r)$$

Join on r
Join on t
Eliminate r
Eliminate t

- Variable Elimination

$$= \sum_t P(L|t) \sum_r P(r)P(t|r)$$

Join on r
Eliminate r
Join on t
Eliminate t

Prior Sampling

For i = 1, 2, …, n in topological order

Sample x_i from P(X_i | Parents(X_i))

Return (x_1, x_2, …, x_n)

This process generates samples with probability:

$$S_{PS}(x_1 \ldots x_n) = \prod_{i=1}^{n} P(x_i|\text{Parents}(X_i)) = P(x_1 \ldots x_n)$$

…i.e. the BN's joint probability

Let the number of samples of an event be $N_{PS}(x_1 \ldots x_n)$

Then

$$\lim_{N\to\infty} \hat{P}(x_1,\ldots,x_n) = \lim_{N\to\infty} N_{PS}(x_1,\ldots,x_n)/N = S_{PS}(x_1,\ldots,x_n) = P(x_1 \ldots x_n)$$

I.e., the sampling procedure is consistent

Rejection Sampling

Input: evidence instantiation
For i = 1, 2, …, n in topological order

Sample x_i from P(X_i | Parents(X_i))
If x_i not consistent with evidence
Reject: return – no sample is generated in this cycle

Return (x_1, x_2, …, x_n)

Likelihood Weighting

Input: evidence instantiation
w = 1.0
for i = 1, 2, …, n in topological order
  if X_i is an evidence variable
    X_i = observation x_i for X_i
    Set w = w * P(x_i | Parents(X_i))
  else
    Sample x_i from P(X_i | Parents(X_i))
return (x_1, x_2, …, x_n), w

Likelihood weighting doesn't solve all our problems

Evidence influences the choice of downstream variables, but not upstream ones (C isn't more likely to get a value matching the evidence)

Gibbs Sampling

Procedure: keep track of a full instantiation $x_1, x_2, \ldots, x_n$. Start with an arbitrary instantiation consistent with the evidence. Sample one variable at a time, conditioned on all the rest, but keep evidence fixed. Keep repeating this for a long time.

Property: in the limit of repeating this infinitely many times the resulting samples come from the correct distribution (i.e. conditioned on evidence).

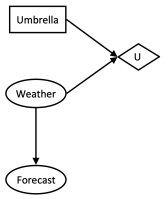
Rationale: both upstream and downstream variables condition on evidence.

In contrast: likelihood weighting only conditions on upstream evidence, and hence weights obtained in likelihood weighting can sometimes be very small. Sum of weights over all samples is indicative of how many "effective" samples were obtained, so we want high weight.

Decision Networks

MEU: choose the action which maximizes the expected utility given the evidence

- Can directly operationalize this with decision networks
  - Bayes nets with nodes for utility and actions
  - Lets us calculate the expected utility for each action
- New node types:
  - Chance nodes (just like BNs)
  - Actions (rectangles, cannot have parents, act as observed evidence)
  - Utility node (diamond, depends on action and chance nodes)



Maximum Expected Utility

Umbrella = leave

$$\text{EU}(\text{leave}|\text{bad}) = \sum_w P(w|\text{bad})U(\text{leave}, w) = 0.34 \cdot 100 + 0.66 \cdot 0 = 34$$

Umbrella = take

$$\text{EU}(\text{take}|\text{bad}) = \sum_w P(w|\text{bad})U(\text{take}, w) = 0.34 \cdot 20 + 0.66 \cdot 70 = 53$$

Optimal decision = take

$$\text{MEU}(F = \text{bad}) = \max_a \text{EU}(a|\text{bad}) = 53$$

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

W	P(W|F=bad)
sun	0.34
rain	0.66

Value of Perfect Information

MEU with no evidence

$$\text{MEU}(\emptyset) = \max_a \text{EU}(a) = 70$$

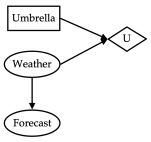
MEU if forecast is bad

$$\text{MEU}(F = \text{bad}) = \max_a \text{EU}(a|\text{bad}) = 53$$

MEU if forecast is good

$$\text{MEU}(F = \text{good}) = \max_a \text{EU}(a|\text{good}) = 95$$

Forecast distribution

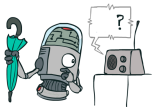


A	W	U
leave	sun	100
leave	rain	0
take	sun	20
take	rain	70

F	P(F)
good	0.59
bad	0.41

$$0.59 \cdot (95) + 0.41 \cdot (53) - 70$$
$$77.8 - 70 = 7.8$$

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



VPI Properties

- Nonnegative

  $$\forall E', e : \text{VPI}(E'|e) \geq 0$$

- Nonadditive
  (think of observing E_j twice)

  $$\text{VPI}(E_j, E_k|e) \neq \text{VPI}(E_j|e) + \text{VPI}(E_k|e)$$

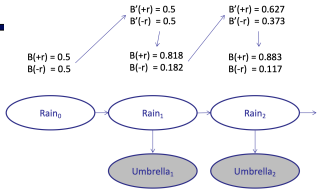
- Order-independent

  $$\text{VPI}(E_j, E_k|e) = \text{VPI}(E_j|e) + \text{VPI}(E_k|e, E_j)$$
  $$= \text{VPI}(E_k|e) + \text{VPI}(E_j|e, E_k)$$

Markov Assumption: Conditional Independence

Basic conditional independence:
Past and future independent given the present
Each time step only depends on the previous
This is called the (first order) Markov property

$$P_\infty(X) = P_{\infty+1}(X) = \sum_x P(X|x)P_\infty(x)$$



R_t	R_{t+1}	$P(R_{t+1}\|R_t)$
+r	+r	0.7
+r	-r	0.3
-r	+r	0.3
-r	-r	0.7

R_t	U_t	$P(U_t\|R_t)$
+r	+u	0.9
+r	-u	0.1
-r	+u	0.2
-r	-u	0.8

$$B(X_{t+1}) \propto_{X_{t+1}} P(e_{t+1}|X_{t+1})B'(X_{t+1})$$