CS3243 Notes

Contents

1 Lecture 1: Introduction **4**
- 1.1 Intelligent Agents . . . 4
- 1.2 Rationality . . . 4
- 1.3 Task Environment: PEAS . . . 4
- 1.4 Properties of Task Environments . . . 4
- 1.5 Building an Agent . . . 5

2 Lecture 2: Uninformed Search **6**
- 2.1 ($\star$) Search Problem Formulation . . . 6
- 2.2 Searching for Solutions . . . 6
- 2.3 Search Strategies . . . 7
- 2.4 Breadth-First Search (BFS) . . . 7
- 2.5 Uniform-Cost Search (UCS) . . . 8
- 2.6 Depth-First Search (DFS) . . . 8
- 2.7 Depth-Limited Search (DLS) . . . 8
- 2.8 Iterative Deepening Search (IDS) . . . 8
- 2.9 Choosing a Search Strategy . . . 9
- 2.10 Search Tracing Problems . . . 9

3 Lecture 3: Informed Search **10**
- 3.1 Best-First Search . . . 10
- 3.2 Greedy Best-First Search (special case of Best-FS) . . . 10
- 3.3 A* Search (special case of Best-FS) . . . 10
- 3.4 Heuristic Design . . . 11
- 3.5 Local Search . . . 11

4 Lecture 4: Adversarial Search **13**
- 4.1 Adversarial Search Problems (Games) . . . 13
- 4.2 Strategies . . . 13
- 4.3 Optimal Decisions (Minimax) . . . 14
- 4.4 α-β Pruning . . . 14
- 4.5 Imperfect, Real-Time Solutions . . . 17

5 Lecture 5: Constraint Satisfaction Problems **18**
- 5.1 CSP Formulation . . . 18
- 5.2 CSP Variants . . . 19
- 5.3 Constraint Variants . . . 19
- 5.4 CSP Search . . . 21
- 5.5 Backtracking Search Algorithm . . . 21
- 5.6 Backtracking Heuristics: Variable and Value Ordering . . . 22
- 5.7 Inference: INFERENCE . . . 23

6 Lecture 6: Project Details **25**
- 6.1 Reinforcement Learning . . . 25
- 6.2 Supervised vs Unsupervised Learning . . . 25
- 6.3 Evaluating Classification Models . . . 25
- 6.4 Algorithm Selection . . . 25

7 Lecture 7: Constraint Satisfaction Problems II **27**
- 7.1 Inference in CSPs: Arc Consistency and AC-3 . . . 27

8 Logical Agents **29**

8.1 Knowledge-based Agents . . . 29
8.2 Example: Wumpus World . . . 30
8.3 Logic . . . 31
8.4 Validity and Satisfiability . . . 36
8.5 Applying Inference Rules . . . 36
8.6 Resolution (for CNF) . . . 36
8.7 KB and Horn Clauses . . . 37
8.8 Forward Chaining . . . 37
8.9 Backward Chaining . . . 39
8.10 Forward vs Backward Chaining . . . 39

9 Uncertainty **40**
9.1 Probability Basics . . . 40
9.2 Bayesian Inference . . . 40

10 Bayesian Networks **41**
10.1 Examples . . . 41
10.2 Inference in Bayesian Networks . . . 44
10.3 Algorithm for Constructing Bayesian Network . . . 44
10.4 Markov Blanket . . . 45
10.5 d-Separation . . . 45

1 Lecture 1: Introduction

1.1 Intelligent Agents

Agents interact with their environment

- Sensors take in percepts
- Actuators perform actions
- Agent function maps *percept histories* to *actions*: $f : P^* \to A$

1.2 Rationality

Rational if selected actions are:

- Based on evidence (prior knowledge/percept sequence)
- Maximise performance measure

Performance measure: defining and measuring 'performance' is difficult

- Task specificity: easier to define 'performance' for a narrower than more general task

Can be rational to explore (perform actions that gather information)

Agent is *autonomous* if behaviour is determined by its own experience

1.3 Task Environment: PEAS

PEAS: Performance measure, Environment, Actuators, Sensors

E.g. Automated Taxi

- Performance measure: safe, fast, comfort, revenue
- Environment: roads, traffic, pedestrians
- Actuators: steering wheel, accelerator, brake
- Sensors: sonar, speedometer, gps, engine sensors

1.4 Properties of Task Environments

- Observability: fully or partially observable? (e.g. fog of war)
- Deterministic vs. stochastic: are there random elements?
  - Still deterministic if random elements do not affect the transition function
  - Not deterministic if some elements are unobservable to player
- Episodic vs. sequential
  - Episodic: choice of current action does not depend on actions in past episodes
  - Sequential: need to consider previous actions too (e.g. chess); current action affects future actions
  - *Order* is important in sequential, not in episodic
- Static vs. dynamic: is environment changing as agent deliberates?
- Discrete vs. continuous: finite/infinite number of distinct states/percepts/actions
  - We prefer solving discrete problems
- Single vs. multi agent

1.5 Building an Agent

Lookup table agent

- For each possible percept, write its optimal action
- Problem: huge table with many many possible percepts
- Problem: no autonomy, hard to change on-the-fly if action is wrong. Unmaintainable and rigid

Types of agents (in increasing complexity):

1. Simple reflex agent: passive, only acts when it observes a percept
2. Model-based reflex agent: passive, has state/internal model of the world
3. Goal-based agent: not just passive and based on percept; has goals and acts to achieve them
4. Utility-based agent: has utility function, acts to maximise it

State is updated based on percept, current state, most recent action, model of the world

(*) Utility function is *internal*, performance measure is *external* and used to assess agent

Learning agent

- Critic + learner => adapt based on performance standard

Exploration vs. exploitation: a classic trade-off the agent must make

- Exploration: get more knowledge to improve future gains
- Exploitation: make use of knowledge to maximise current gains

2 Lecture 2: Uninformed Search

Problem-solving agent: one kind of goal-based agent

Environment: fully observable, deterministic, discrete

Uninformed search: no additional knowledge incorporated

2.1 ($\star$) Search Problem Formulation

- State: including initial state
  - Abstract ONLY the relevant information, and nothing else
  - Everything in the state should be a variable that can change, no constants
- Actions: ACTIONS(S) gives set of all valid actions that can be executed in state s
  - Define it for every possible state s
- Transition model: RESULT(S,A) gives new state s' upon doing action a in state s
  - Define it for every possible state s and its valid action a
- Goal test: test if a state s is the goal state
  - E.g. $IsCheckmate(s)$ or $IsSolved(s)$
- Path cost: path cost is additive sum of step costs
  - Step cost $c(s, a, s')$ — e.g. 1 per action taken

2.2 Searching for Solutions

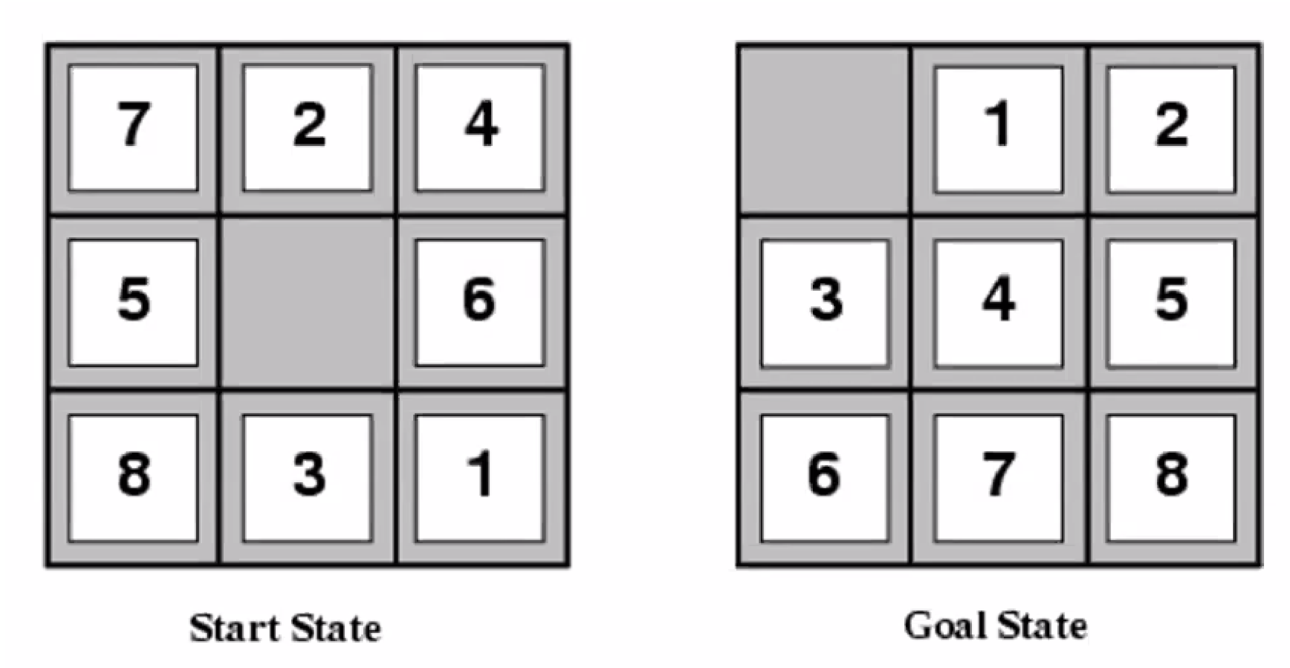
Solution: sequence of actions leading from initial to goal state

Example: route planning

- Reduce map down to nodes with edges between them of certain weights

Example: 8-puzzle

- State: an arrangement of numbers in 3x3 grid, represented as matrix/array
- Actions: moving one filled square to a blank adjacent square
- Transition model: [depends on representation] — function that takes in state + action => new state
- Goal test: whether each cell matches the goal state, one-for-one
- Cost function: uniform cost of 1 for each action



State vs node

- State: represents physical configuration
- Node: data structure constituting part of search tree: includes state, parent node, action, path cost $g(n)$
- Two different nodes can contain the same world state

2.3 Search Strategies

Which order should we expand the nodes in?

Evaluation criteria

- Completeness: always find a solution if it exists
- Optimality: finds a least-cost solution
- Time complexity: number nodes generated
- Space complexity: max number of nodes in memory

Problem parameters

- b: maximum # of successors for each node — branching factor
- d: depth of *shallowest* goal node
- m: maximum depth of search tree

2.4 Breadth-First Search (BFS)

Frontier: Queue

Properties of BFS

- Complete: yes, as long as b is finite
- Optimal: no, unless uniform step cost, or uniform across each level
- Time: $O(b^d) = O(b) + O(b^2) + \ldots + O(b^d)$
- Space: $O(b^d)$ (max size of frontier)

Applies goal test when pushing to frontier: reduces time and space complexity from $O(b^{d+1})$ to $O(b^d)$

2.5 Uniform-Cost Search (UCS)

Frontier: Priority queue, by path cost

- Idea: explore unexpanded node with *least-path-cost* (equivalent to BFS if all step costs are equal)

Properties of UCS

- Complete: yes, if all step costs are $\geq \epsilon$
  - If not, ever-decreasing step costs may get you stuck infinitely on a suboptimal path
  - Still yes even if b or d is infinite, or search space is infinite
- Optimal: yes (when it is complete)
- Time: $O(b^{1+\lfloor \frac{C^*}{\epsilon} \rfloor})$ where C^* is the optimal cost
  - Reach nodes at distance 0, ϵ, 2ϵ, ..., $\lfloor \frac{C^*}{\epsilon} \rfloor \epsilon$ of goal => total $\lfloor \frac{C^*}{\epsilon} \rfloor + 1$ steps
- Space: $O(b^{1+\lfloor \frac{C^*}{\epsilon} \rfloor})$

2.6 Depth-First Search (DFS)

Frontier: Stack

Properties of DFS

- Complete: yes, as long as depth is finite
- Optimal: no
- Time: $O(b^m)$
- Space: $O(bm)$ (can be $O(m)$ — at each level, just keep track of self and parent)

2.7 Depth-Limited Search (DLS)

Idea: run DFS with depth limit ℓ

- Only works if we know the goal is within ℓ steps
- Time: $O(b^\ell)$
- Space: $O(b\ell)$ (can be $O(\ell)$)

2.8 Iterative Deepening Search (IDS)

Idea: keep performing DLSs with increasing depth limit, until goal node is found

- Better if state space is large and depth of solution is unknown
- It can be wasteful with repeated effort
- But overhead is not that large (e.g. $b = 10, d = 5$ — 11%)

Properties of IDS

- Complete: yes, if b is finite
- Optimal: no, unless step cost is uniform
- Time: $O(b^d)$

- Space: $O(bd)$ (can be $O(d)$)

Property	BFS	UCS	DFS	DLS	IDS
Complete	Yes[1]	Yes[2]	No	No	Yes[1]
Optimal	No[3]	Yes	No	No	No[3]
Time	$O(b^d)$	$O\left(b^{1+\lfloor \frac{C^*}{\varepsilon} \rfloor}\right)$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$
Space	$O(b^d)$	$O\left(b^{1+\lfloor \frac{C^*}{\varepsilon} \rfloor}\right)$	$O(bm)$	$O(b\ell)$	$O(bd)$

1. Complete if b is finite
2. Complete b is finite and step cost $\geq \epsilon$
3. Optimal if step costs are identical

2.9 Choosing a Search Strategy

Depends on the problem

- Depth: finite/infinite?
- Solution depth: known/unkwown?
- Repeated states
- Step costs: identical/different?
- Completeness and optimality – are they needed?
- Resource constraints (time/space)?

2.10 Search Tracing Problems

Tree-Search

Frontier
S(0)
A(1) B(5) C(15)
S(2) B(5) G(11) C(15)
...

Graph-Search

Frontier	Explored
S(0)	
A(1) B(5) C(15)	S
B(5) G(11) C(15)	S, A
G(10) C(15)	S, A, B

3 Lecture 3: Informed Search

Informed search: exploits problem-specific knowledge, uses *heuristics* to guide search

(AIMA Chapter 3.5.1-2, 3.6.1-...)

3.1 Best-First Search

Idea: use an *evaluation function* $f(n)$ for each node n

- Measures *cost estimate*
- Expand node with the lowest estimated cost first

Implementation: priority queue, ordered by non-decreasing cost f

3.2 Greedy Best-First Search (special case of Best-FS)

Evaluation function: $f(n) = h(n)$

- $h(n)$: cost estimate from n to goal (heuristic)
- Idea: expand the node that appears the closest to goal

Properties

- Complete: yes, if b is finite
- Optimal: no
- Time: $O(b^m)$, but if heuristic is good can reduce complexity substantially
- Space: $O(b^m)$ (max size of frontier)

3.3 A* Search (special case of Best-FS)

Idea: avoid expanding paths that are already expensive

- Expand the path that appears the cheapest

NOTE: remember we use a *priority queue* on $f(n) = g(n) + h(n)$; pick the smallest one

Evaluation function: $f(n) = g(n) + h(n)$

- $g(n)$: cost of reaching n from start node, under the current path (not necessarily the smallest among all paths!)
- $h(n)$: cost estimate from n to goal (heuristic)
- $f(n)$: estimated cost of cheapest path *through* n to goal

Properties

- Complete: yes, if there is finite number of nodes and $f(n) \leq f(G)$
- Optimal: yes, if you have an admissible/consistent heuristic
- Time (no great detail): $O(b^{h^*(s_0)-h(s_0)})$ where $h^*(s_0)$ is actual cost from root to goal
- Space: $O(b^m)$ (max size of frontier)

3.4 Heuristic Design

3.4.1 Admissibility

Admissible heuristics

- $h(n)$ is *admissible* if it never overestimates the cost to reach goal
- Definition: $\forall n, h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost from n to goal state

Theorem: if $h(n)$ is admissible, then A* using TREE-SEARCH is optimal

- (Proof: see lecture 3 slide 22)

3.4.2 Consistency

Consistent heuristic:

- $h(n)$ is *consistent* if it means that $f(n)$ is non-decreasing along any path (triangle inequality)
- Definition: $h(n) \leq d(n, n') + h(n')$, where n' is a successor of n
- Lemma: if h is consistent, then $f(n') \geq f(n)$
- (???)

Theorem: if $h(n)$ is consistent, then A* using GRAPH-SEARCH is optimal

- Claim: when A* selects a node n for expansion, the shortest path to n has been found
- (Proof: see lecture 3 slide 26)

3.4.3 Admissibility & Consistency

All consistent heuristics are admissible, but not the other way round.

Example: 8-puzzle

- Heuristic 1: number of misplaced tiles
- Heuristic 2: total Manhattan distance

3.4.4 Dominance

h_2 *dominates* h_1 if $h_2(n) \geq h_1(n)$ for all n, where both heuristics are admissible

- Dominating heuristics are better: incur lower search costs under A*

3.4.5 Deriving Admissible Heuristics

Common exam question: given a problem, derive an admissible heuristic

Solution: *relax* the problem — then it'll only be 'easier' to reach the goal. Heuristic that uses this relaxed problem can NEVER over-estimate goal

3.5 Local Search

Path to the goal is irrelevant; we only want to reach the goal state

Local search algorithms: maintain single "current best" state, and try to improve it

Advantages

- Very little/constant memory
- Find reasonable solutions in large state space

3.5.1 Hill-Climbing Algorithm

Hill-Climbing

- current $\leftarrow$ initial state
- while True:
  - neighbour $\leftarrow$ best successor of current
  - if neighbour's value $\leq$ current's value: return current
  - current $\leftarrow$ neighbour

Problem: depending on initial state, can get stuck in local maxima (or minima)

Solution: try random restarts or sideway moves

4 Lecture 4: Adversarial Search

4.1 Adversarial Search Problems (Games)

Game: agent vs. agent(s)

- Unlike a *search problem*, which is agent vs. environment
- There are other utility-maximising agents
- Solution is a strategy that specifies a move for every possible opponent response

Zero-sum game: agent utilities sum to zero

- Completely adversarial game

Two-player zero-sum game

- *MAX* player: wants to maximise value
- *MIN* player: wants to minimise value

Problem formulation

- Initial state s_0
- States s
- ($\star$ NEW) *Player* PLAYER(s): defines which player has the move in state s
- Actions ACTIONS(s): returns set of legal moves in state s
- Transition model RESULT(s, a): returns state that results from move a in state s
- Terminal test TERMINAL(s): check whether the game has ended
- ($\star$ NEW) Utility function UTILITY(s, p): final numeric value for game with terminal state s for player p

For now, we assume 2-player, deterministic, turn-taking

4.2 Strategies

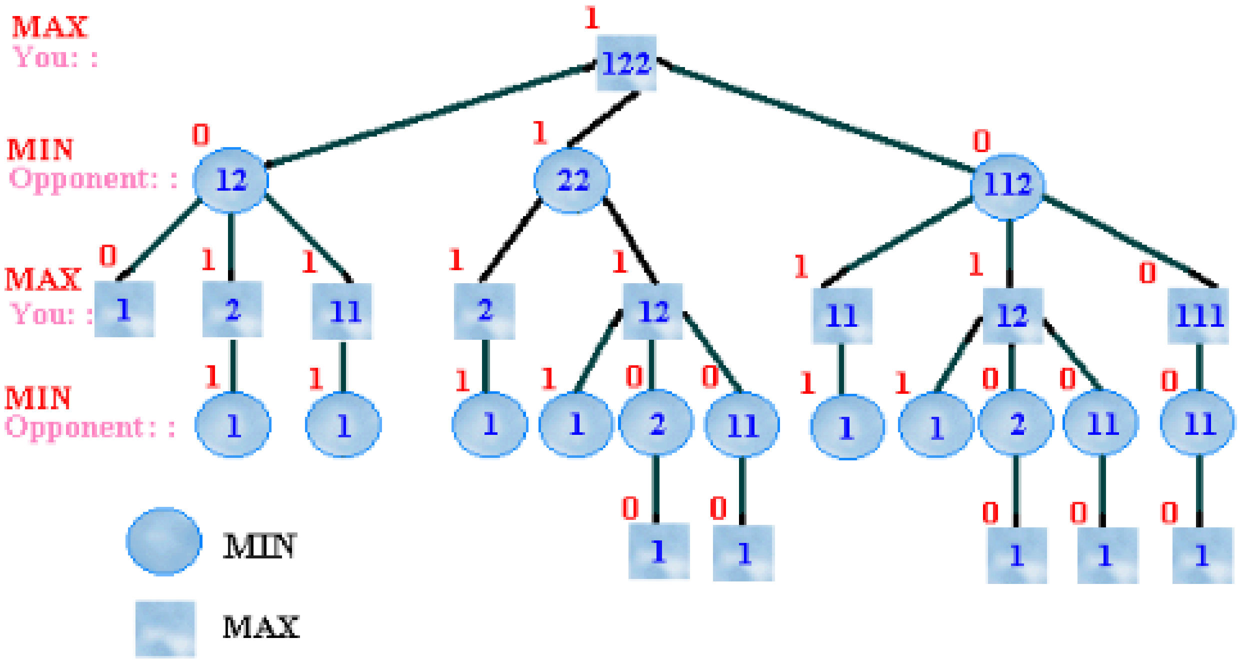
Strategy

- Strategy s for player i: for every node of the tree that the player can make a move in, specify what player will do
- Need to define strategy in states that will never be reached (I think this means instead that it needs to be defined for all possible states)

Winning strategy

- Winning: s_1^* for player 1 is *winning* — if for any strategy s_2 by player 2, game ends with player 1 as the winner
- Non-losing: t_1^* for player 1 is *non-losing* — if for any strategy s_2 by player 2, game ends with EITHER player 1 as the winner or tie

4.3 Optimal Decisions (Minimax)



Minimax(s)

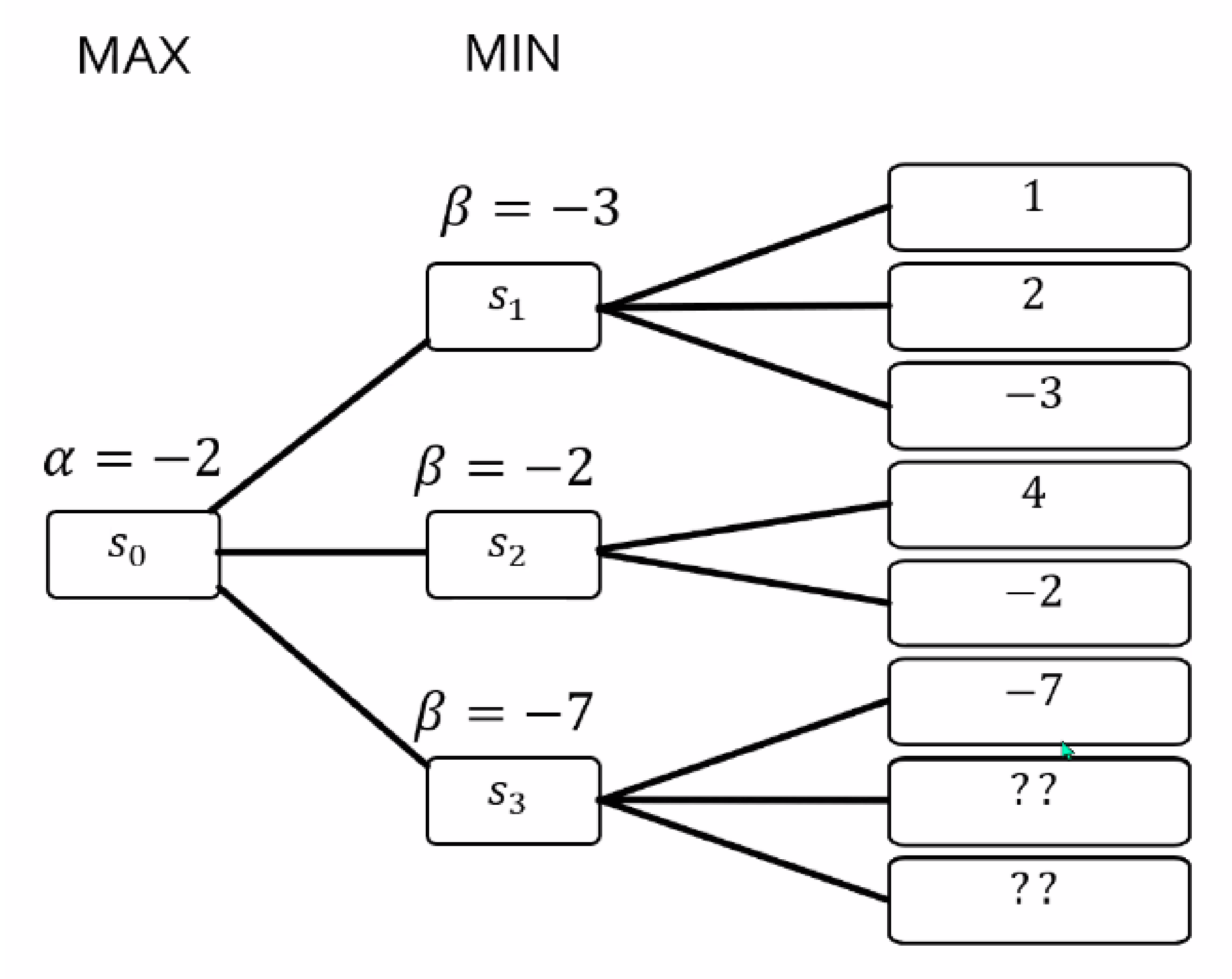
- Utility(s) if TerminalTest(s)
- $\max_{a \in \text{Actions}(s)}$ Minimax(Result(s, a)) if Player(s) $= MAX$
- $\min_{a \in \text{Actions}(s)}$ Minimax(Result(s, a)) if Player(s) $= MIN$

Properties

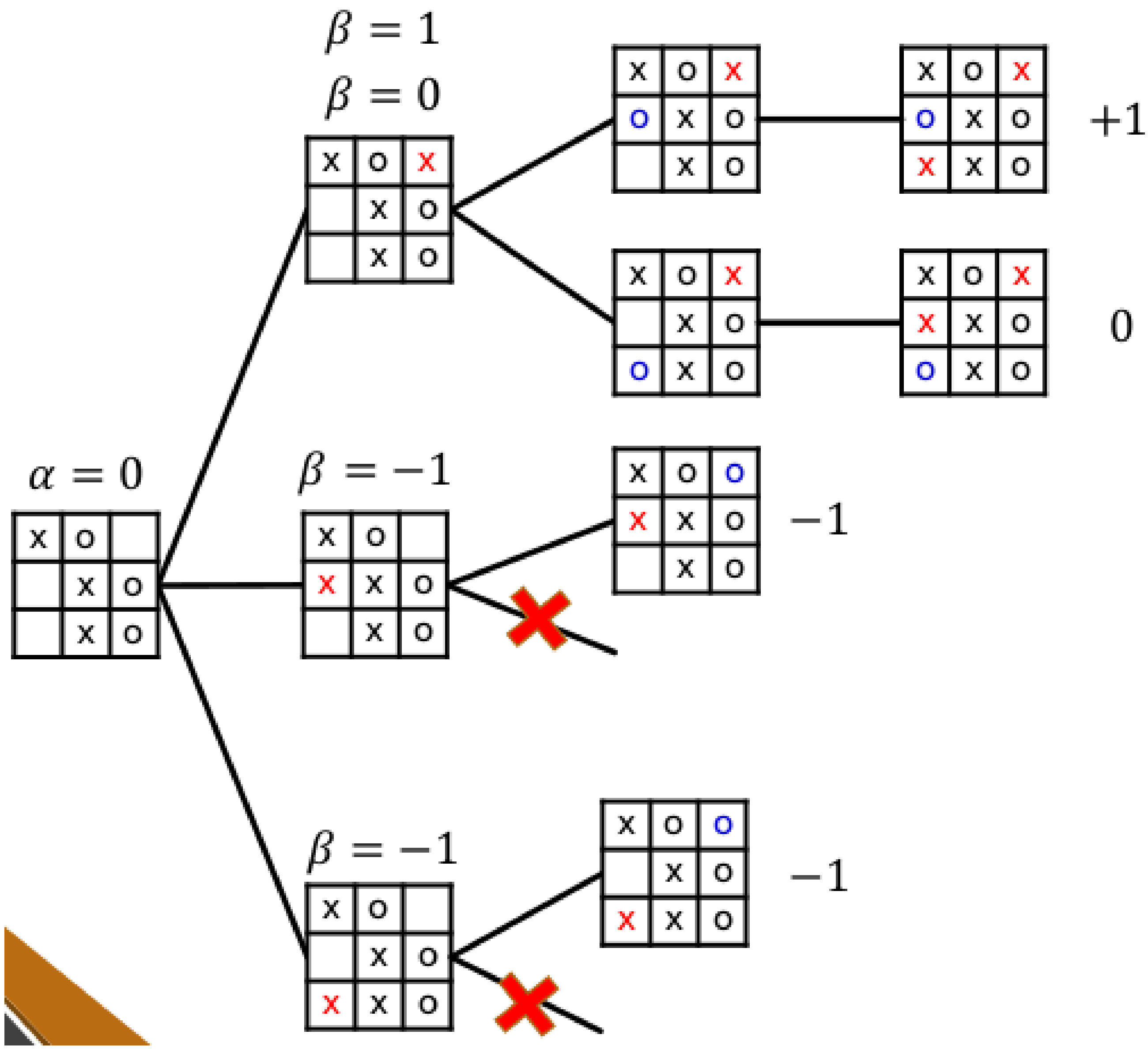
- Complete: yes, if game tree is finite
- Optimal: yes
- Time: $O(b^m)$ (similar to DFS)
- Space: $O(bm)$ (similar to DFS)

4.4 α-β Pruning

- α: largest value so far for MAX
- β: smallest value so far for MIN



Example above: in the bottom branch, β=-7, but α=-2 > β. So no need to explore the remaining branches



α-β pruning

- MAX node n: $\alpha(n)$ = highest observed value found on path from n. Initially $\alpha(n) = -\infty$
- MIN node n: $\beta(n)$ = lowest observed value found on path from n. Initially $\alpha(n) = -\infty$
- ($\star$) Given MIN node n, stop searching below n if there is some MAX ancestor i of n with $\alpha(i) \geq \beta(n)$
- ($\star$) Given MAX node n, stop searching below n if there is some MIN ancestor i of n with $\beta(i) \leq \alpha(n)$

Analysis of α-β pruning

- "Perfect" ordering: time complexity = $O(b^{\frac{m}{2}})$ — can search twice as deep!
- Random ordering: time complexity = $O(b^{\frac{3}{4}m})$ for $b < 1000$

Summary

- Initially, $\alpha(n) = -\infty$, $\beta(n) = +\infty$
- $\alpha(n)$ is MAX along search path containing n
- $\beta(n)$ is MIN along search path containing n
- If a MIN node has value $v \leq \alpha(n)$, no need to explore further

- If a MAX node has value $v \geq \beta(n)$, no need to explore further

4.5 Imperfect, Real-Time Solutions

Time limit

- How to deal with super large search trees? $\Rightarrow$ Limit maximum depth of tree
- Evaluation function: estimated expected utility of state (similar to heuristic)
- Cutoff test: e.g. depth limit

Cutting-Off Search: similar to Depth-Limited Search (DLS)

- Previously: $\textsc{Minimax}(s) = \textsc{Utility}(s)$ if $\textsc{Terminal-Test}(s)$
- Now: $\text{H-}\textsc{Minimax}(s) = \textsc{Eval}(s)$ if $\textsc{Cutoff-Test}(s)$
- i.e. run minimax until depth d, then use evaluation function to choose nodes
- Can also consider iterative deepening approach

Stochastic Games

- How to deal with games with *randomisation*?
- Game tree now has added *chance layers* — even more complex
- Calculating the expected value of a state — MUCH harder than deterministic games

5 Lecture 5: Constraint Satisfaction Problems

AIMA Chapter 6.1-6.4

5.1 CSP Formulation

<u>CSP representation</u>

- Variables $\vec{X} = X_1, \ldots, X_n$
- Domain D for variables — X_i has domain D_i — list of values a variable can take
- Constraints $\vec{C}$ — restrictions on values a variable can take
  - Defined by constraint language: algebra/logic (don't give abstract english description)

<u>CSP objective</u>

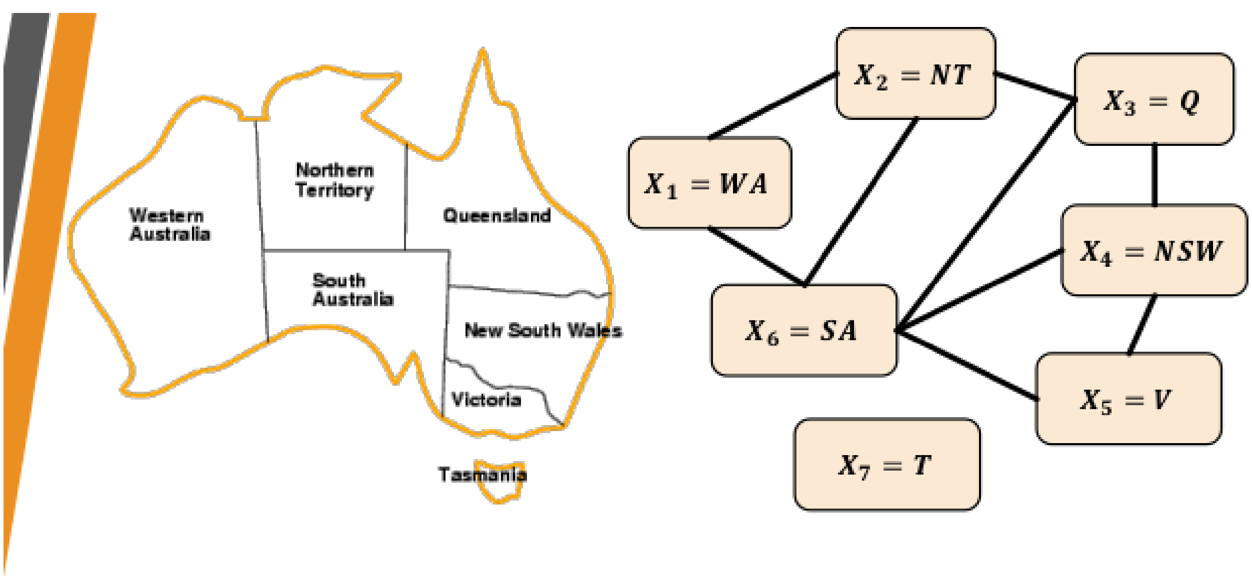
- Find a legal assignment $(y_1, \ldots, y_n)$ — $y_i \in D_i$ for all $i \in n$
- *Complete*: all variables assigned values
- *Consistent*: all constraints satisfied

5.1.1 Example: Graph Colouring

<u>Constraint graph</u>: node are variables, edges are constraints

- Variables: $\vec{X} = \langle WA, NT, Q, NSW, V, SA, T \rangle$
- Domains: $D_i = \{R, G, B\}$
- Constraints: if $(X_i, X_j) \in E$ then $color(X_i) \neq color(X_j)$

<u>Binary constraint</u>: involves 2 variables



Variables:	$\vec{X} = \langle WA, NT, Q, NSW, V, SA, T \rangle$
Domains:	$D_i = \{R, G, B\}$
Constraints:	If $(X_i, X_j) \in E$ then $color(X_i) \neq color(X_j)$

5.1.2 Example: Job-Shop Scheduling

- Car assembly consists of 15 tasks
- Variables: Axle_{F}, Axle_{B}, Wheel_{LF}, Wheel_{RF}, Wheel_{LB}, Wheel_{RB}, Nuts_{LF}, Nuts_{RF}, Nuts_{LB}, Nuts_{RB}, Cap_{LF}, Cap_{RF}, Cap_{LB}, Cap_{RB}, Inspect
- Domain: $D_i = \{1, 2, \dots, 27\}$
- Precendence constraints: e.g. $Axle_F + 10 \leq Wheel_{RF}$
- Disjunctive constraints: e.g. $(Axle_F + 10 \leq Axle_B) or (Axle_B + 10 \leq Axle_F)$

5.2 CSP Variants

Discrete variables

- Finite domains: e.g. sudoku
- Infinite domains: integers, strings etc. e.g. job-shop scheduling

Continuous variables

- E.g. start/end times for Hubble Space Telescope observations
- Linear programming problems can be solved in polynomial time

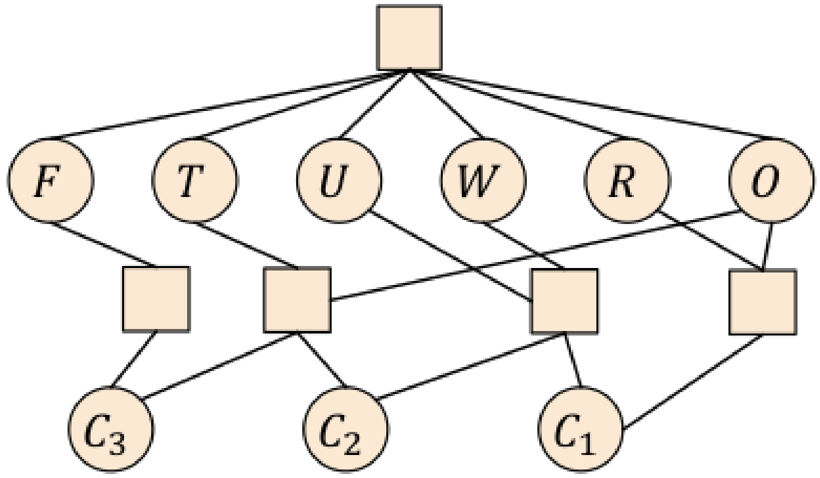
5.3 Constraint Variants

- Unary constraints: 1 variable e.g. $SA \neq Green$
- Binary constraints: 2 variables e.g. $SA \neq WA$
- Global/higher-order constraints: 3 or more variables e.g. $X_1 + X_2 - 4X_7 \leq 15$

5.3.1 Example: Cryptarithmetic Puzzle

- Each letter represents one digit (base 10)
- Different letters should correspond to different digits
- T and F cannot be 0

$$
\begin{array}{r}
TWO \\
+\,TWO \\
\hline
FOUR
\end{array}
$$



Variables:	$\vec{X} = \langle \mathrm{F, T, U, W, R, O, C_1, C_2, C_3} \rangle$
Domains:	$D_i = \{0, \dots, 9\}$
Constraints:	$AllDiff(F, T, U, W, R, O)$ $O + O = R + 10C_1$ $C_1 + W + W = U + 10C_2$ $C_2 + T + T = O + 10C_3$ $C_3 = F$ $T, F \neq 0$

(Also, C_1, C_2, C_3 should be either 0 or 1)

Drawing constraints

- Global constraints: draw using square
  - E.g. $AllDiff(F, T, U, W, R, O)$ — one square linking them all
- Binary constraints: can draw using square, if not just draw an edge directly
- Unary constraints: don't need to draw

5.3.2 Example: Sudoku

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

(a)

	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

(b)

Variables:	$A_1, \ldots, A_9, \ldots, I_1, \ldots, I_9$ (81 variables)
Domains:	$D_i = \{1, \ldots, 9\}$
Constraints:	$AllDiff(\ldots) \times 27$ (9 columns, 9 rows, 9 boxes) e.g. $AllDiff(A_1, \ldots, A_3; B_1, \ldots, B_3; C_1, \ldots, C_3)$ is the constraint for the top-left box.

15

5.4 CSP Search

5.4.1 Search Formulation

- State and initial state: initially empty assignment []
- Transition function: assign a valid value to an unassigned variable, fail if no valid assignments
- Goal test: all variables assigned
- Every solution appears at exactly depth n
- Search path is irrelevant

5.4.2 Search Tree

Each level: pick any remaining variable, give it any possible assignment.

Maximum size i.e. total number of leaves: $n! \times d^n$

- E.g. 4 Variables and 3 Values — size $= 4! \times 3^4 = 1944$

5.5 Backtracking Search Algorithm

Backtracking search

- More efficient than the search above
- Perform DFS with single-variable/level assignments: at every level, consider assignments to a *single* variable

- Order of variable assignment is irrelevant

BACKTRACKING-SEARCH(csp) returns a solution, or failure

- return BACKTRACK($\{\}, csp$)

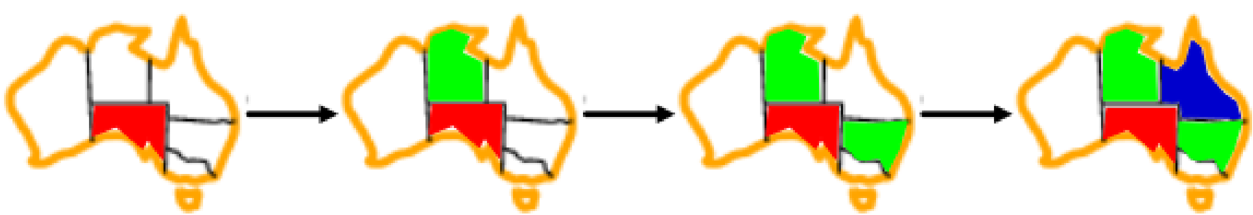
BACKTRACK($assignment, csp$) returns a solution, or failure

- if assignment is complete, return it
- var $\leftarrow$ SELECT-UNASSIGNED-VARIABLE(csp)
- for each value in ORDER-DOMAIN-VALUES($var, assignment, csp$):
  - if value is consistent with assignment:
    * add $\{var = value\}$ to assignment
    * inferences $\leftarrow$ INFERENCE($csp, var, value$)
    * if inferences $==$ failure: continue
    * add inferences to assignment
    * result $\leftarrow$ BACKTRACK($assignment, csp$)
    * if result $\neq$ failure: return result
  - remove $\{var = value\}$ and inferences from assignment
- return failure

5.6 Backtracking Heuristics: Variable and Value Ordering

5.6.1 Variable-Order Heuristics: SELECT-UNASSIGNED-VARIABLE

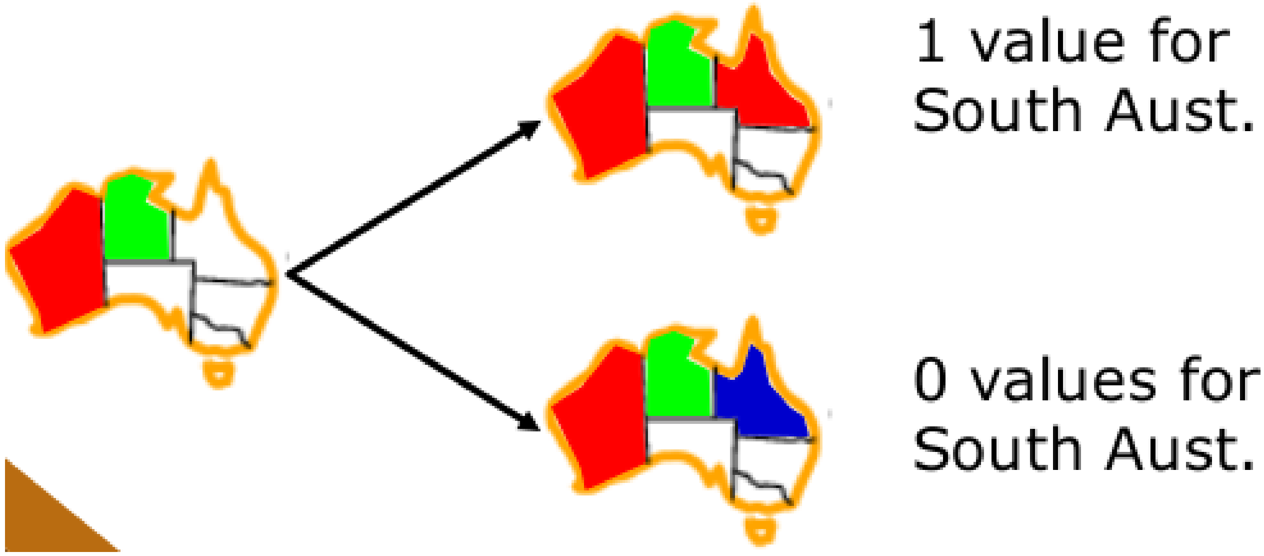
1. Most constraining variable a.k.a. degree heuristic: choose the variable that imposes the most constraints on the remaining unassigned variables
   - This is best: it reduces the branching factor => likely get to terminal state faster



1. Most constrained variable a.k.a. Minimum-Remaining-Values (MRV) heuristic: choose the variable with the fewest remaining legal values
   - Use as tiebreaker

5.6.2 Value-Order Heuristic: ORDER-DOMAIN-VALUES

1. Least constraining value: choose the value that rules out the fewest values for the neighbouring unassigned variables
   - Because we're "actually trying to solve the problem" in this stage, unlike the variable stage



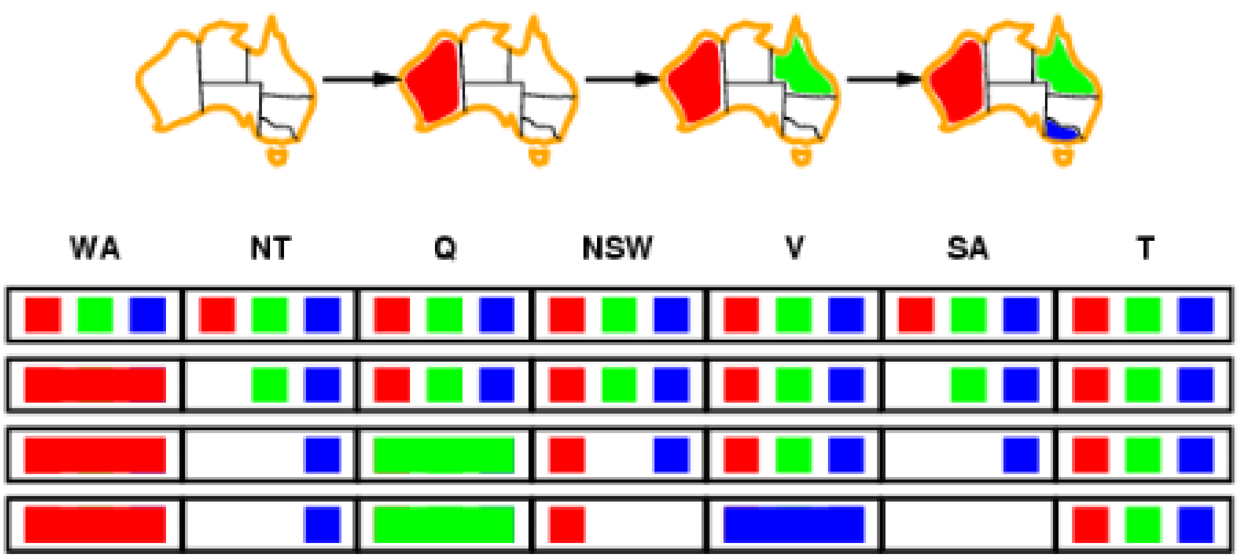
5.7 Inference: Inference

Idea: check for failures early.

5.7.1 Forward Checking

Forward checking

- Keep track of remaining legal values for unassigned variables
- (⋆) Terminate search when any variable has no legal values left

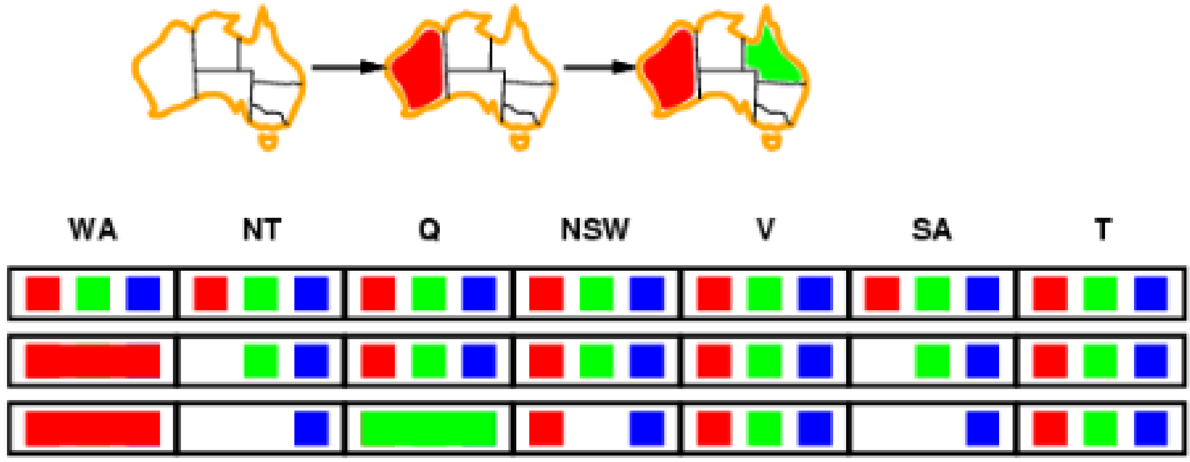


E.g. here SA has no remaining valid assignments => failure.

5.7.2 Constraint Propagation

Constraint propagation: 'move ahead' with the constraints

- Repeatedly locally enforce constraints
- Infer illegal values for assignments early on



E.g. here NT and SA both have to be blue, but by constraints, they can't be both blue

6 Lecture 6: Project Details

6.1 Reinforcement Learning

1. $\leftarrow$ Agent receives input information
2. $\rightarrow$ Agent performs valid action
3. $\leftarrow$ Agent obtains reward

State $s_t \rightarrow$ action $a_t \rightarrow$ reward r_t (also takes you to state s_{t+1})

6.2 Supervised vs Unsupervised Learning

Unsupervised: data are unlabelled => perform things like clustering

Supervised: data are labelled => perform things like predicting labels for new unlabelled data

- Classification problems: supervised learning problem with discrete-valued class

#	$\mathrm{x_1}$	...	$\mathrm{x_q}$	y
1	$\mathrm{x_{1,1}}$	...	$\mathrm{x_{1,q}}$	$\mathrm{y_1}$
$\vdots$	$\vdots$		$\vdots$	$\vdots$
p	$\mathrm{x_{p,1}}$	...	$\mathrm{x_{p,q}}$	$\mathrm{y_p}$

Goal: build a model F such that $F(X) = y$ with high accuracy, where X is a new unseen instance

6.3 Evaluating Classification Models

Generating models and evaluating models are different!

Idea behind evaluation: measure generalisation performance

- Assume instances are governed by overarching distribution D
- Want to determine, the probability of accurately classifying ANY instance drawn from D

Example methods

- Hold-out testing, i.e. training and testing sets
- k-fold cross-validation

6.4 Algorithm Selection

Given a classification dataset S, and a set of algorithms we'll use A, determine which algorithm $a^* \in A$ is optimal

Meta-learning: pose algorithm selection problem as another classification problem

Generate meta-dataset

- Each x_i corresponds to a *characteristic* of a dataset (e.g. number of instances, r^2, mutual information, etc.)
- Each y_i corresponds to the optimal algorithm a^* for that dataset

Problem: what is the *overarching distribution* governing the algorithm selection problem?

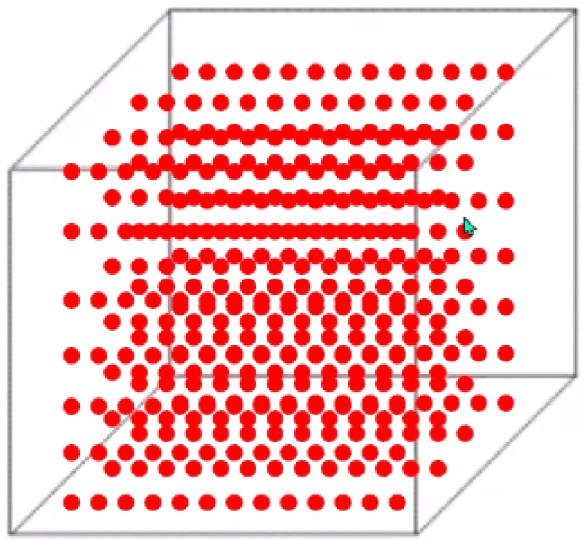
- Which datasets are properly representative for this problem?
- Where can we draw them from?
- Repositories exist, but are these representative of all possible problems?

Just ensure that the model built has *good coverage*; uniform distribution of datasets

- At least have many instances representing different patterns of when one algorithm will be better than another

Project 1 Search Problem

Search for some S^* such that when we plot each $s_j \in S^*$ within the expertise space, we get a distribution that is close to uniform



7 Lecture 7: Constraint Satisfaction Problems II

7.1 Inference in CSPs: Arc Consistency and AC-3

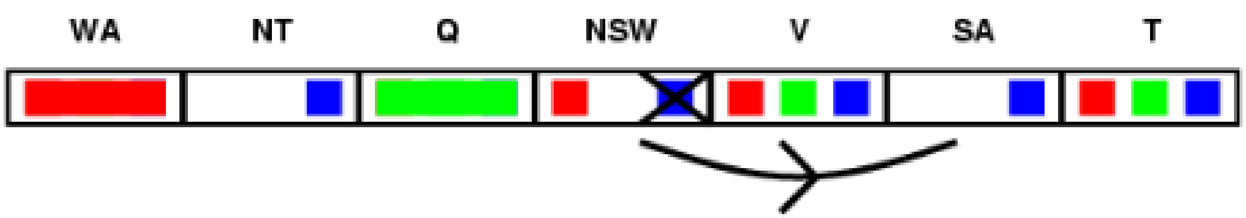
Constraint propagation: node consistency for unary constraints, arc conistency for binary constraints

7.1.1 Arc Consistency

Arc consistency: X is arc-consistent wrt X_j i.e. arc (X_i, X_j) is consistent, iff for every value $x \in D_i$ there exists some value $y \in D_j$ that satisfies binary constraint on arc (X_i, X_j)

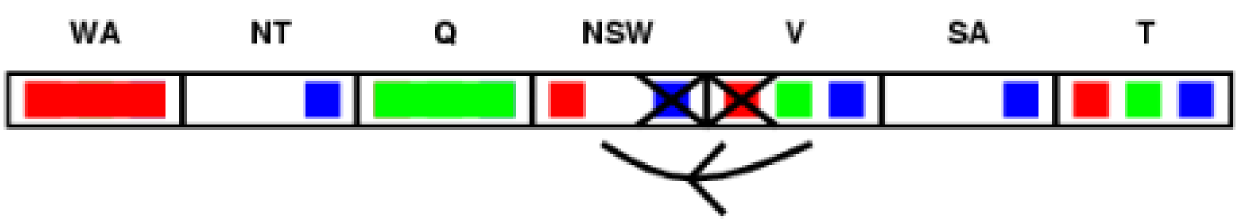
- Note that arcs are *directed*.
- To maintain AC: remove a value if it makes a constraint impossible to satisfy.

(SA, NSW): OK



(NSW, SA): Need to remove blue value from NSW

After an update on X_i where it loses a value, we MUST *re-check* the neighbours of X_i.



(V, NSW): Now that NSW cannot be blue, V cannot be red

7.1.2 AC-3 Algorithm

AC-3(csp) returns $false$ if inconsistency is found, otherwise $true$

- $queue \leftarrow$ all the arcs in csp
- while $queue$ is not empty:
  - $(X_i, X_j) \leftarrow$ REMOVE-FIRST($queue$)
  - if REVISE(csp, X_i, X_j):
    - if size of $D_i = 0$ then return $false$

        * for each X_k in NEIGHBOURS$(X_i) - \{X_j\}$:
            · add (X_k, X_i) to *queue*

REVISE(csp, X_i, X_j) returns *true* if we revise the domain of X_i

- $revised \leftarrow false$
- for each x in D_i:
    - if no value y from D_j allows (x, y) to satisfy constraint between X_i and X_j:
        * delete x from D_i
        * $revised \leftarrow true$
- return *revised*

<u>Time complexity</u>: $O(n^2d^3)$

- CSP has at most n^2 directed arcs
- Each arc (X_i, X_j) can be inserted at most d times into the queue, since X_i has at most d values
- REVISE: checking consistency of arc takes $O(d^2)$ time
- AC-3: $O(n^2 \times d \times d^2) = O(n^2d^3)$

7.1.3 Maintaining AC (MAC)

Search procedure

- Establish AC at root
- When AC-3 terminates, choose a new variable and value
- Re-establish AC given the new variable choice — maintain AC
- Repeat;
- Backtrack if AC gives *empty domain*

We could use AC-3 purely as preprocessing, or do it at every step

<u>AC-3 with preprocessing</u>

- Add all arcs

<u>AC-3 with backtracking</u>

- If domain of variable X' is updated, then only need to add all arcs leading to X'
- i.e. check each arc (X_i, X')

7.1.4 Generalised Arc Consistency (not covered in CS3243)

What if our arcs are global and not binary constraints?

- Can reduce to binary constraints if possible
- Otherwise, we can extend arc consistency (2-consistency) to k-consistency

8 Logical Agents

AIMA Chapter 7

8.1 Knowledge-based Agents

Previously: we use search; no real model of what the agent knows Now: we represent agent domain knowledge using logical formulas

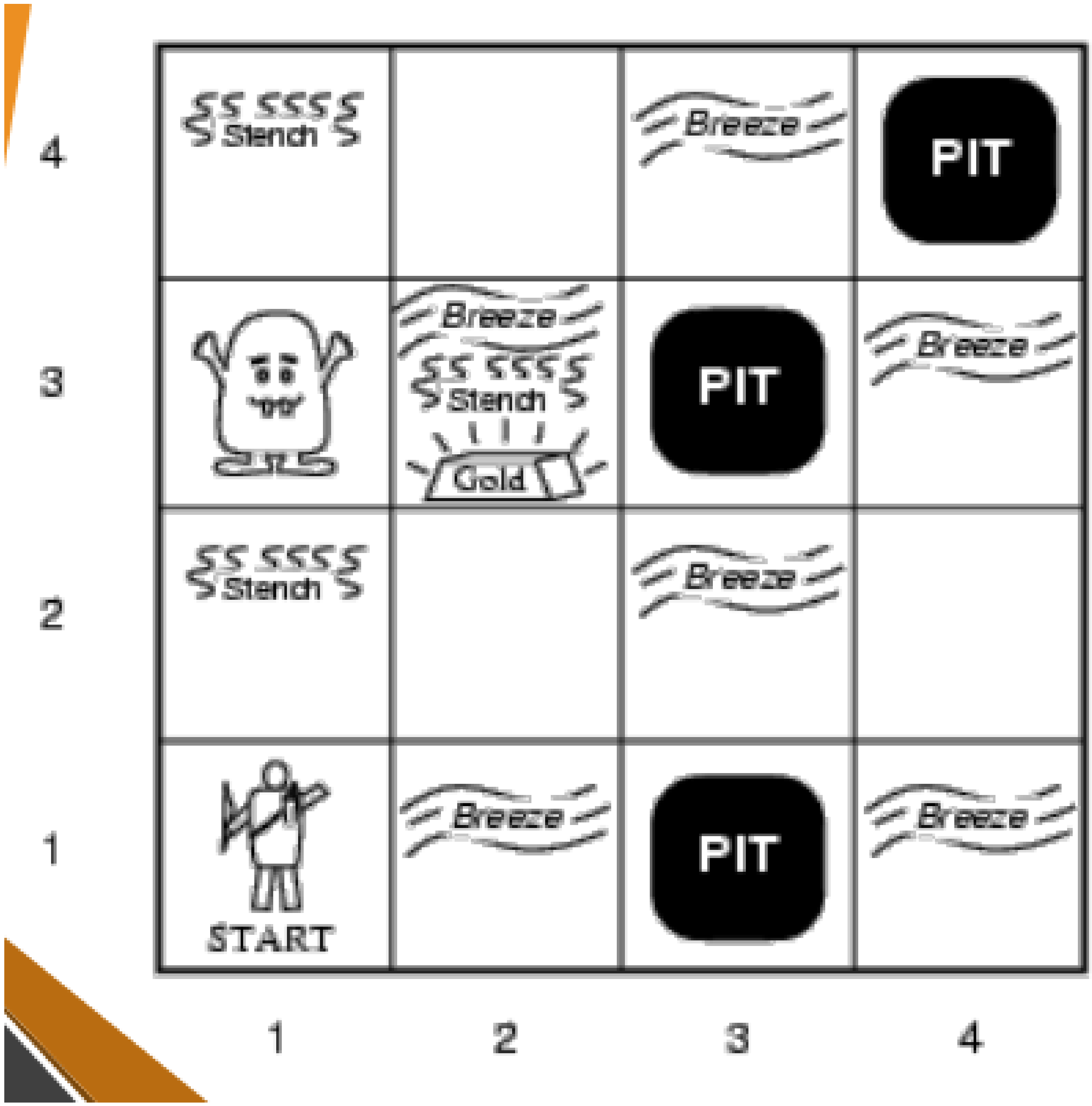
Logical agent: Inference Engine + Knowledge Base

- Inference Engine: domain-indepenedent algorithms
- Knowledge Base: domain-specific content — *set of sentences* in a formal language
  - Pre-populate with background/domain knowledge (e.g. game rules)

KB-AGENT(*percept*) returns an *action*

- persistent: KB, a knowledge base; t, a counter for time initally set to 0
- TELL(KB, MAKE-PERCEPT-SEQUENCE($percept, t$))
- $action \leftarrow$ ASK(KB, MAKE-ACTION-QUERY(t))
- TELL(KB, MAKE-ACTION-SEQUENCE($action, t$))
- $t \leftarrow t + 1$
- return *action*

8.2 Example: Wumpus World



Wumpus and pits will kill you

- Beside wumpus: stench
- Beside pit: breeze

Task environment (PEAS)

- Performance measure: +1000 for gold, -1000 for dying, -1 for each action, -10 for using arrow
- Environment: 4x4 grid of rooms
- Actuators: forward, turn left, turn right, grab gold, shoot arrow
- Sensors: perceive stench/breeze/glitter/scream

Environment

- Fully observable: no — only local perception

- Deterministic: yes
- Episodic: no — sequential actions
- Static: yes
- Discrete: yes
- Single-Agent: yes

Initial KB

- If there is a PIT, there is a BREEZE beside it
- If there is a WUMPUS, there is a STENCH beside it
- It's a 4x4 grid world
- ...

8.3 Logic

Logic: formal language for KR, consists of syntax + semantics

- Syntax: defines valid sentences in language: S_1, S_2, etc.
  - Provides logical connectives for constructing complex sentences from simpler ones, e.g. $S_1 \wedge S_2$ etc.
  - e.g $x + y = 4$ is a sentence
- Semantics: defines the meaning of a sentence; the "truth of each sentence with respect to each possible world"
  - i.e. defines truth (validity) of a sentence in a given world (for some given value assignments in an environment)
  - e.g. $x + y = 4$ is true in a world where $x = 2$ and $y = 2$, but false in a world where $x = 1$ and $y = 1$

8.3.1 Logical Reasoning: Entailment

Modelling: m models/satisfies sentence α if α is true under m

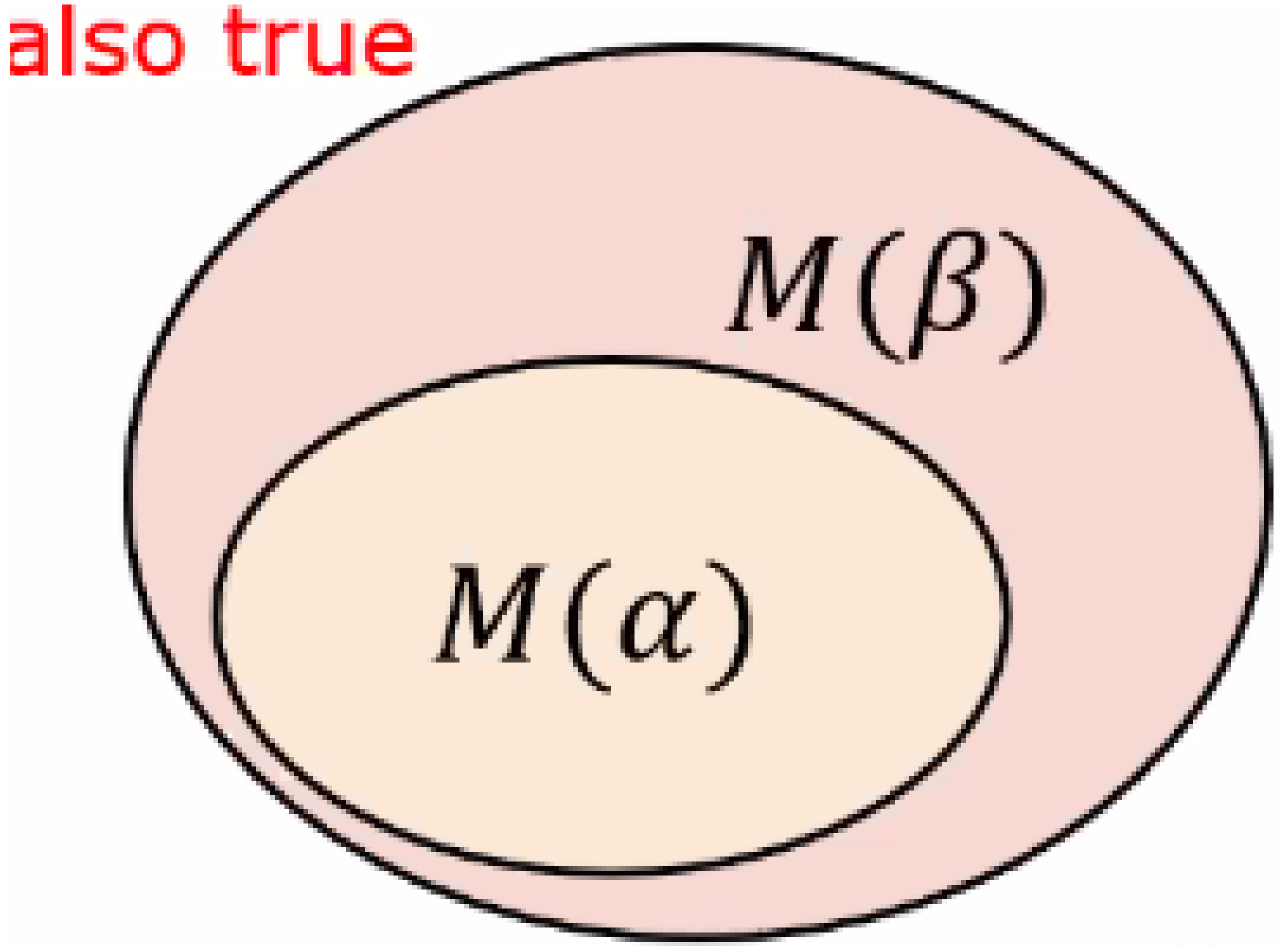
- A model represents the idea of a "possible world" — it assigns a truth value to all the variables
- Let $M(\alpha)$ be the set of all models satisfying α
- E.g. $\alpha = (q \in \mathbb{Z}_+) \wedge (\forall n, m \in \mathbb{Z}_+ : q = nm \Rightarrow n \vee m = 1)$

Entailment $\vDash$: means that one sentence follows logically from another sentence

- $\alpha \vDash \beta$ is equivalent to $M(\alpha) \subseteq M(\beta)$
- E.g. $\alpha = (q \text{ is prime})$ entails $\beta = (q \text{ is odd}) \vee (q = 2)$

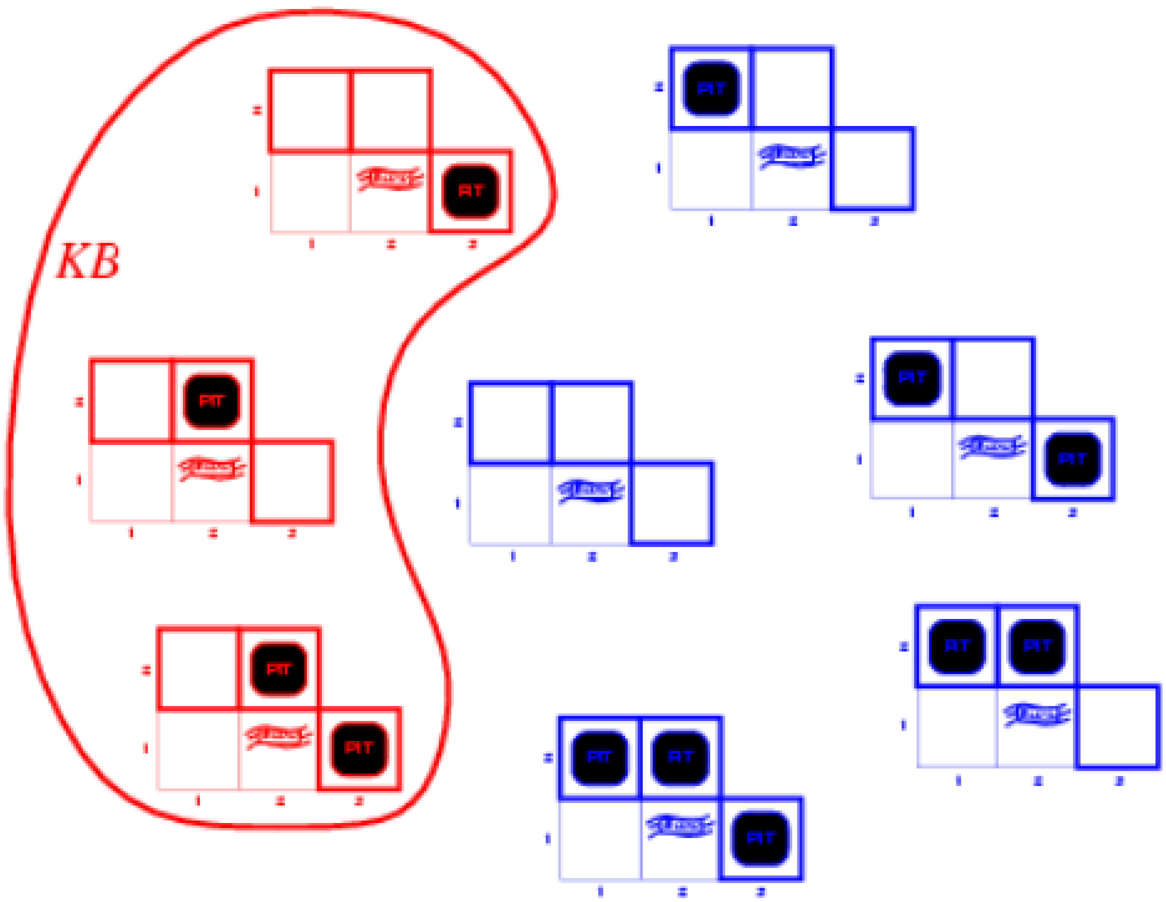
KB is true $\Leftrightarrow$ all its rules are true, i.e. $\bigwedge_{k=1}^{n} R_k$ is true

Key takeaway: if our model is a subset of a sentence α, then α is true

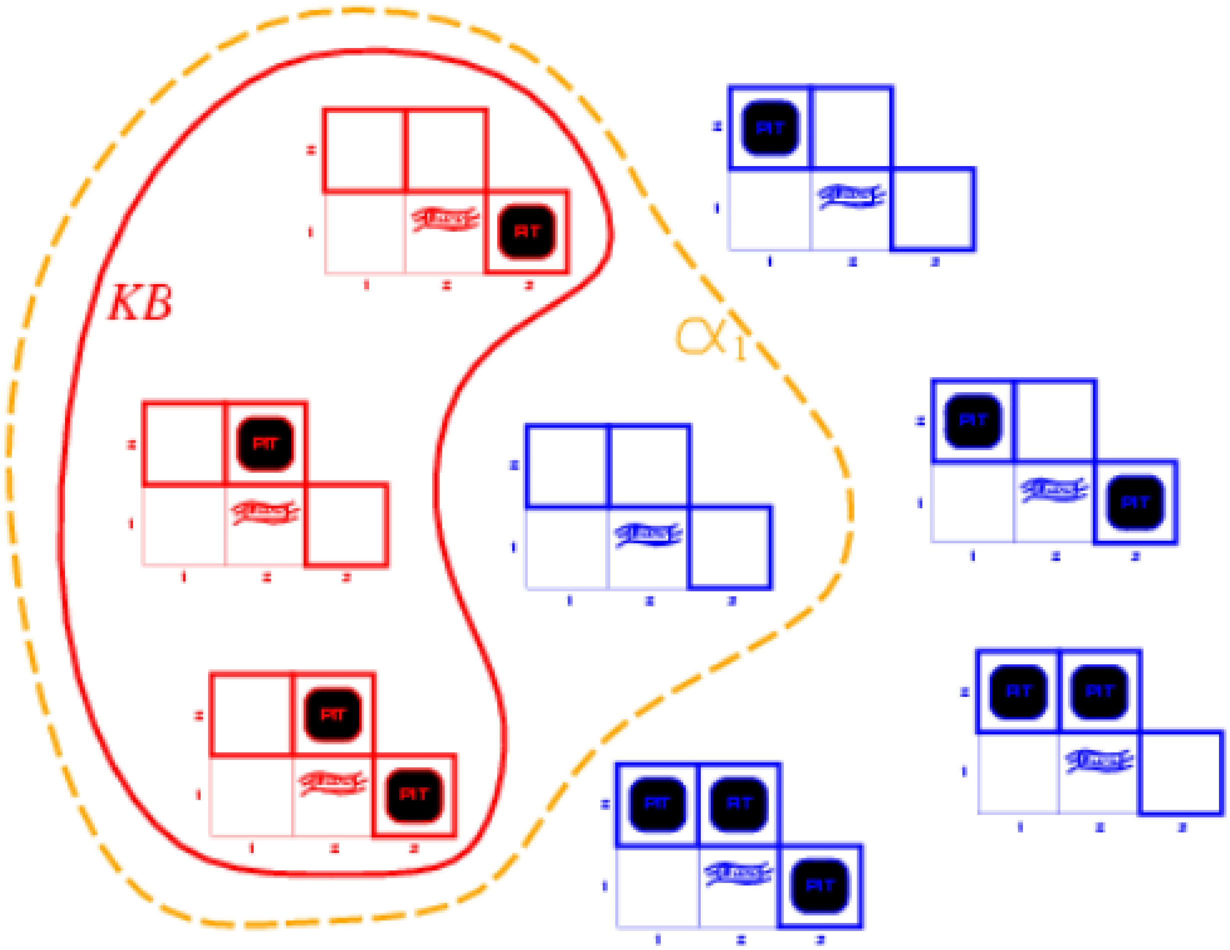


<u>Example: Wumpus World</u>

- Suppose we move right to (2,1) to detect a breeze
- Consider 8 possible models for KB with pits (3 boolean choices $\Rightarrow$ 8 possible models
- KB is only true



- Suppose we want to infer sentence α_1 = "(1,2) is safe".
- True: proved by model checking. Worlds satisfying KB $\subseteq$ worlds where (1,2) is safe



- Let P_{ij} be whether there's a pit in (i, j).
- Let B_{ij} be whether there's a breeze in (i, j).

Rules

- $R_1 : \neg P_{1,1}$
- $R_4 : \neg B_{1,1}$
- $R_5 : P_{2,1}$

"Pits cause breezes in adjacent squares"

- $R_2 : B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})$
- $R_3 : B_{2,1} \Rightarrow (P_{1,1} \vee P_{2,2} \vee P_{3,1})$

8.3.2 Inference Algorithm

Let $KB \vdash_A \alpha$ be "sentence α is derived/inferred from KB by inference algorithm A".

- A is *sound* if $KB \vdash_A \alpha$ implies $KB \vDash \alpha$
  - If KB derives α, then KB entails α
  - Whatever is derived is correct, i.e. "don't infer nonsense"
- A is *complete* if $KB \vDash \alpha$ implies $KB \vdash_A \alpha$
  - If KB entails α, then KB derives α
  - Whatever is correct is derived, i.e. if it's implied it will be inferred

We want an inference algorithm that is both *sound* and *complete*.

- Let X = all sentences derived from KB using A
- Let Y = all possible sentences entailed by KB
- $X = Y$: sound and complete
- $X \subset Y$: sound and not complete
- $Y \subset X$: not sound and complete
- Otherwise: not sound and not incomplete

8.3.3 Inference!

- Given a knowledge base, infer something about the world
- Inference: deriving new knowledge out of percepts
- Given KB and α, we want to know if $KB \vDash \alpha$, i.e. can we infer α from KB?

8.3.4 Truth Table for Inference

Truth Table for Inference

$B_{1,1}$	$B_{2,1}$	$P_{1,1}$	$P_{1,2}$	$P_{2,1}$	$P_{2,2}$	$P_{3,1}$	KB	α_1
$false$	$false$	$false$	$false$	$false$	$false$	$false$	$false$	$true$
$false$	$false$	$false$	$false$	$false$	$false$	$true$	$false$	$true$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$false$	$true$	$false$	$false$	$false$	$false$	$false$	$false$	$true$
$false$	$true$	$false$	$false$	$false$	$false$	$true$	$\underline{true}$	$\underline{true}$
$false$	$true$	$false$	$false$	$false$	$true$	$false$	$\underline{true}$	$\underline{true}$
$false$	$true$	$false$	$false$	$false$	$true$	$true$	$\underline{true}$	$\underline{true}$
$false$	$true$	$false$	$false$	$true$	$false$	$false$	$false$	$true$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$true$	$true$	$true$	$true$	$true$	$true$	$true$	$false$	$false$

KB true

- Build a truth table of all possible values
- Evaluate the models where the KB is true
- Does KB entail α_1: See if the remaining rows are true for α_1. If so, we can infer it

<u>Inference by Truth Table Enumeration</u>

- Sound: directly implements entailment, and calculates all possible inferences from KB by brute force
- Complete: only finitely many combinations of truth assignments, and goes through all
- (For above 2, see diagnostic quiz 8/Sam's slides W10)
- Time complexity: $O(2^n)$

- Space complexity: $O(n)$ — because the enumeration is depth-first

8.4 Validity and Satisfiability

Validity: a sentence is *valid* if it is true in *all* models

- E.g. statements that are true regardless of truth assignments (tautology), e.g. $True$, $A \vee \neg A$
- $KB \vDash \alpha$ iff $(KB \Rightarrow \alpha)$ is valid

Satisfiability: a sentence is *satisfiable* if it is true in *some* model

Unsatisfiability: a sentence is *unsatisfiable* if it is true in *no* models

- $KB \vDash \alpha$ iff $(KB \wedge \neg\alpha)$ is unsatisfiable

8.5 Applying Inference Rules

Form of search problem: search for more knowledge (search grows our KB)

- States: KBs
- Actions: inference rules
- Transition: add sentence to current KB
- Goal: KB contains sentence to prove

Examples of inference rules

- And-elimination: $a \wedge b \vDash a$
- Modus ponens: $a \wedge (a \Rightarrow b) \vDash b$
- Logical equivalences: $(a \vee b) \vDash \neg(\neg a \wedge \neg b)$

8.6 Resolution (for CNF)

CNF: conjunction of disjunctions i.e. 'and's of 'or's

- E.g. $(x_1 \vee \neg x_2) \wedge (x_2 \vee x_3 \vee \neg x_4)$
- Conversion to CNF: simple standard stuff

Resolution: if x appears in C_1 and $\neg x$ appears in C_2, it can be deleted (x must be a literal)

- $(P \vee x) \wedge (Q \vee \neg x)$ is the same as $(P \vee Q)$
- Resolution is *sound* and *complete* for propositional logic

($\star$) Resolution algorithm

- Proof by contradiction: to prove α, suppose otherwise add $\neg\alpha$ into the KB
- Step 1: add $\neg\alpha$ into KB
- Step 2: convert KB to CNF
- Step 3: pick 2 rules and reduce; repeat
- Use resolution to see if the eventual KB is $\emptyset$ i.e. contradiction

Resolution algorithm is sound and complete

- Soundness: why (???)
- Completeness: why (???)

8.6.1 Example

Assume we are at (1,1), and we want to infer if there is no pit at (1,2)

- $KB = \neg B_{1,1} \land B_{1,1} \Leftrightarrow (P_{1,2} \lor P_{2,1})$
- $\alpha = \neg P_{1,2}$

Resolution algorithm

- Step 1: add $\neg\alpha$ to KB
  - $KB = \neg B_{1,1} \land (B_{1,1} \Leftrightarrow (P_{1,2} \lor P_{2,1})) \land P_{1,2}$
- Step 2: convert KB to CNF
  - $KB = \neg B_{1,1} \land P_{1,2} \land (\neg B_{1,1} \lor P_{1,2} \lor P_{2,1}) \land (\neg P_{1,2} \lor B_{1,1}) \land (\neg P_{2,1} \lor B_{1,1})$
- Step 3: pick two rules and reduce
  - Reduce rule 2 and rule 4: $P_{1,2}$ in rule 2 and $\neg P_{1,2}$ in rule 4
  - Reduced to rule 6: $B_{1,1}$
  - Reduce rule 1 and rule 6: $\neg B_{1,1}$ in rule 1 and $B_{1,1}$ in rule 6
  - Reduce to $\emptyset$

8.7 KB and Horn Clauses

Horn clauses: of form $B_1 \land B_2 \land \ldots \land B_k \Rightarrow A$

- Forward/backward chaining is *sound* and *complete* for KB comprised of horn clauses

Clauses with at most 1 positive literal

- Clause is a sentence comprising disjunctions: e.g. $A \lor \neg B$, $\neg A \lor \neg C \lor D$

Three forms of horn clauses

- Literals (facts): e.g. A
- Definite clause (rules): e.g. $B_1 \land B_2 \land \ldots \land B_k \Rightarrow A$ i.e. $\neg B_1 \lor \neg B_2 \lor \ldots \lor \neg B_k \lor A$

8.8 Forward Chaining

Idea: keep adding literals/facts

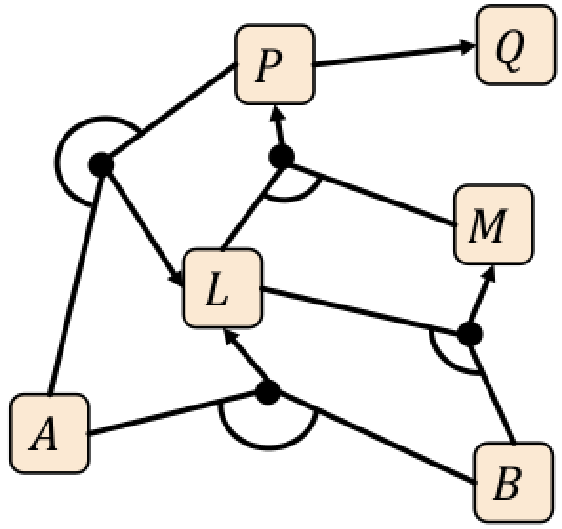
Idea: fire any rule whose premise is satisfied in the KB, add its conclusion to the KB, repeat until query Q is found

AND-OR graph

KB of horn clauses

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

AND-OR graph



<u>FC algorithm</u>

- For every rule c, let $count(c)$ be the number of literals in its premise
- For every literal s, let $inferred(s)$ be initially false
- Let $agenda$ be a queue of literals, initally containing all literals known to be true
- While $agenda \neq \emptyset$:
  - Pop literal p from $agenda$; if it is Q, we are done
  - Set $inferred(p)$ to be true
  - For each clause $c \in KB$ such that p is in the premise of c, decrement $count(c)$
  - If $count(c) = 0$, add conclusion of c to $agenda$

<u>Example</u>

- Iteration 1: agenda = [A, B]
- Iteration 2: agenda = [B]
- Iteration 3: agenda = [] $\Rightarrow$ [L]
- Iteration 4: agenda = [] $\Rightarrow$ [M]
- Iteration 5: agenda = [] $\Rightarrow$ [P]
- Iteration 6: agenda = [] $\Rightarrow$ [Q]

<u>Proof of completeness</u>

- FC derives every atomic sentence/literal entailed by a horn KB
- Suppose FC reaches a fixed point, where no new atomic sentences are derived
- Consider the final state as a model m that assigns true/false to symbols based on inferred table
- Every clause in the original KB is true in m
- Hence m is a model of KB

- If $KB \Vdash q$, then q is true in *every* model of KB, including m

8.9 Backward Chaining

Idea: work backwards from the query Q

To prove Q by backwards chaining,

- Check if Q is known already, or
- Prove by backwards chaining the premise of some rule concluding in Q
- We need to avoid loops: check if the new subgoal is already on the goal stack
- Backtracking DFS

8.10 Forward vs Backward Chaining

- FC: data-driven reasoning
  - When you don't know the goal, but want to try to build towards it
  - May do a lot of work that is irrelevant to the goal
- BC: goal-driven reasoning
  - When you know the goal, and want to work backwards to prove it
  - Complexity of BC can be sublinear in size of KB

9 Uncertainty

9.1 Probability Basics

Probability

- Random variable X: quantifies an outcome of a random occurrence
- Domain D_X: set of all outcomes of a random variable
- Event: subset of a domain
- Probability distribution: assigns a probability value $p(x) \in [0, 1]$ to every $x \in D_X$

Axioms of probability

- Total probability is 1: $\sum_{x \in D_X} p(x) = 1$
- $P(A \cup B) = P(A) + P(B) - P(A \cap B)$

Multiple random variables

- Joint probability: $p(x, y) = P(X = x, Y = y)$ (discrete)
- Marginal probability: $p(x) = \sum_{y \in D_Y} p(x, y)$
- Conditional probability: e.g. $P(A|B) = \frac{P(A \cap B)}{P(B)}$

Bayes' rule: $P(A|B) = \frac{P(B|A) \times P(A)}{P(B)}$ Chain rule: $P(X_1, X_2, \ldots, X_k) = \prod_{i=1}^{k} P(X_i|X_1, \ldots, X_{i-1})$

Independence

- $P(A \cap B) = P(A) \times P(B)$, i.e. $P(A|B) = P(A)$
- Conditional independence: $P(X \cap Y|Z) = P(X|Z) \times P(Y|Z)$

9.2 Bayesian Inference

$P(X|Y_1, \ldots, Y_k)$ — we want to find the probability of event X, given probabilities of other events Y_i

Inference by enumeration

- Find $P(X)$ by summing over all atomic events
- $P(X) = \sum_{x \in X} P(X = x)$

Bayes' rule and conditional independence

- $P(C|E_1, \ldots, E_n) = \frac{P(C) \times P(E_1, \ldots, E_n|C)}{P(E_1, \ldots, E_n)} \propto \prod_{i=1}^{n} P(E_i|C)$
- This is an example of the naive Bayes' model

Normalisation

- $P(X|Y_1, Y_2) = \frac{P(Y_1, Y_2|X) \times P(X)}{P(Y_1, Y_2)}$
- But we don't care about $P(Y_1, Y_2)$, so set it to α
- Then $P(X = \mathit{healthy}|A) = \alpha \times P(X = \mathit{healthy}) \times P(Y_1 = y_1|X = \mathit{healthy}) \times P(Y_2 = y_2|X = \mathit{healthy}) = \ldots$
- Then $P(X = \mathit{sick}|A) = \alpha \times P(X = \mathit{sick}) \times P(Y_1 = y_1|X = \mathit{sick}) \times P(Y_2 = y_2|X = \mathit{sick}) = \ldots$

10 Bayesian Networks

Represent joint distributions via a graph

- Nodes: random variables
- Edges: assume X causes/influences Y
- For each node X, we can get a conditional distribution for X given its parents, i.e. $P(X|Parents(X))$
- Conditional probability table (CPT): the conditional distribution of X for each combination of parent values

Then $P(X_1, \ldots, X_n) = \prod_{i=1}^{n} P(X_i|Parents(X_i))$

- The fewer parents overall, the better (the less complex the graph is)

Complexity

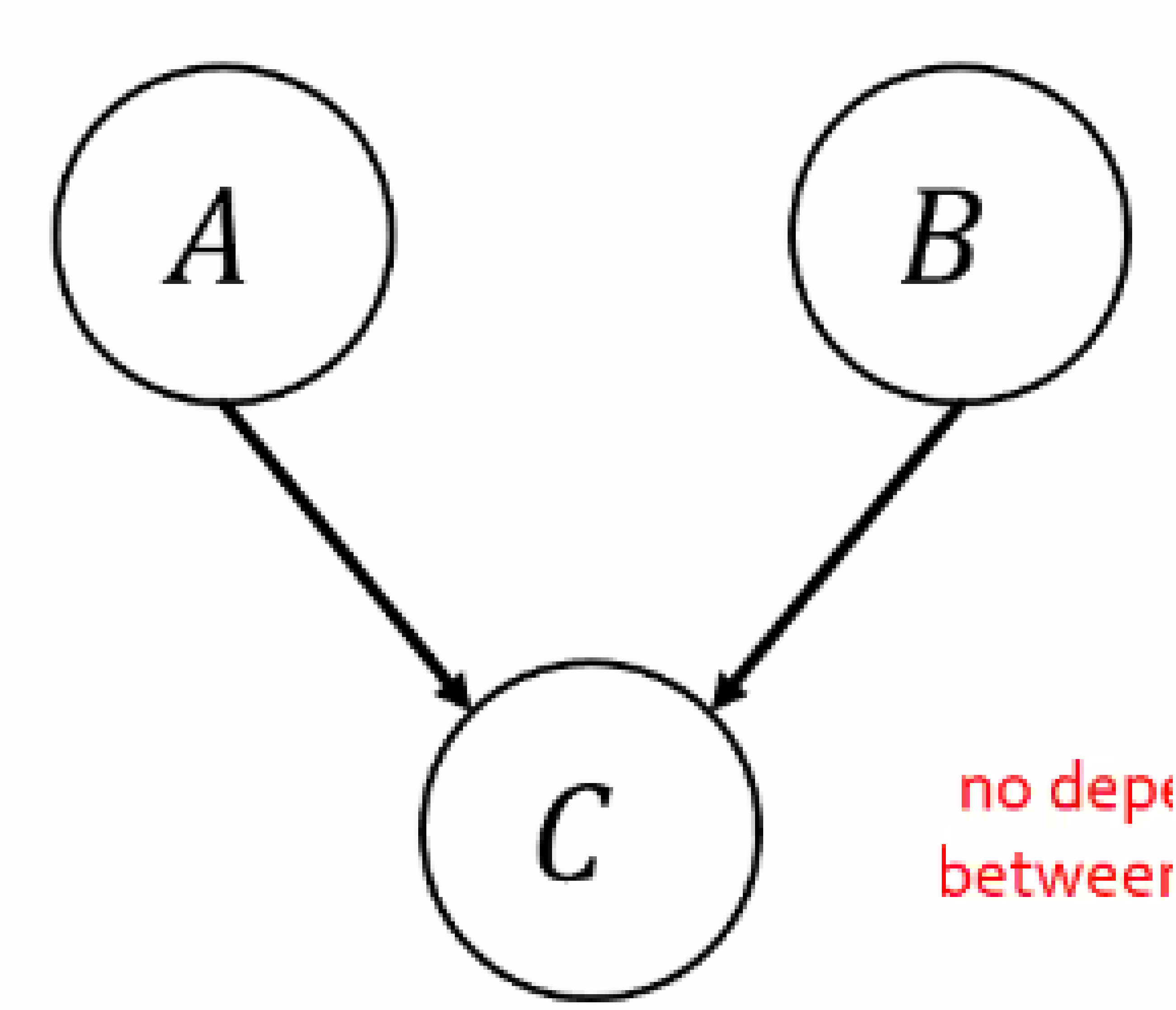
- If each variable has $\leq k$ parents, then network representation requires $O(n2^k)$ values, compared to $O(2^n)$ for full joint distribution

10.1 Examples

Example: independent causes/common effect

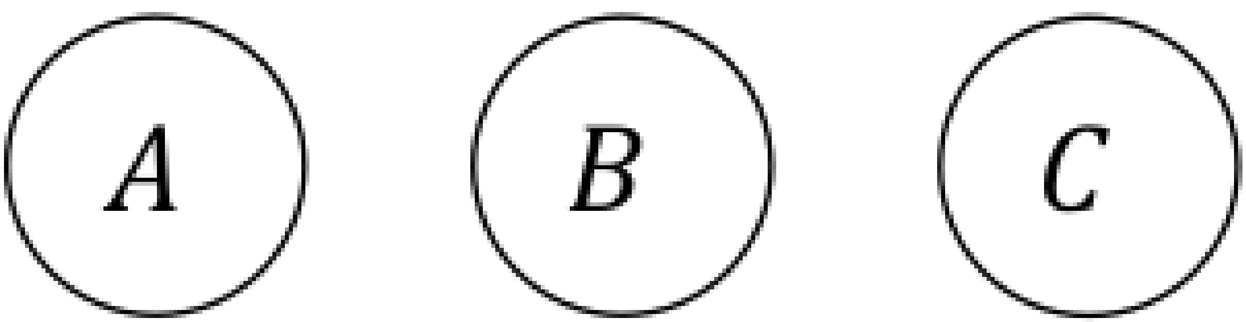
$P(A, B, C) = P(C|A, B) \cdot P(A) \cdot P(B)$

- A and B are pairwise independent, *unless* you condition on observing the effect C: then A and B are conditionally dependent



Example: independent events

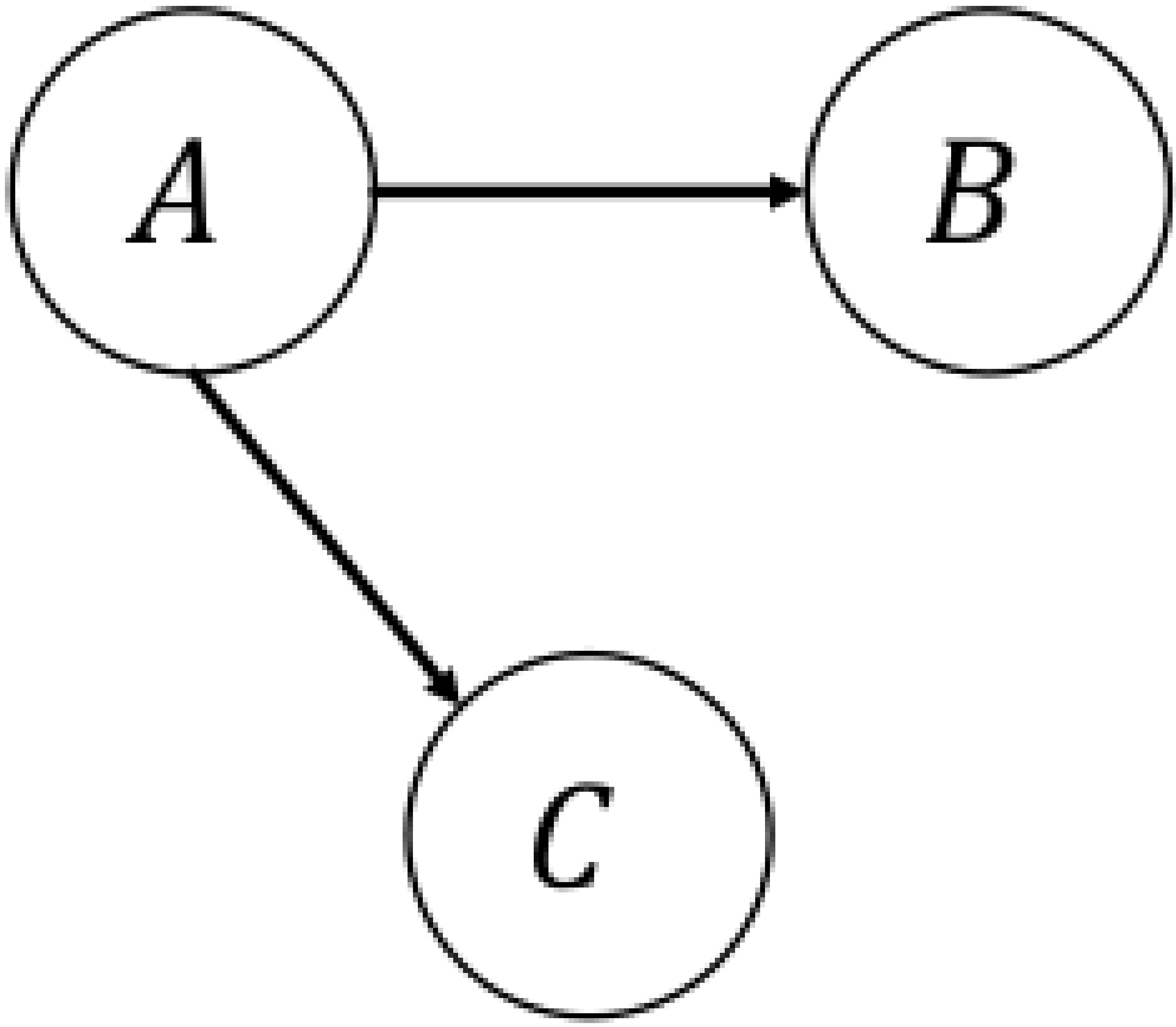
$P(A, B, C) = P(A) \cdot P(B) \cdot P(C)$



Example: conditionally independent effects/common cause

$P(A, B, C) = P(C|A) \cdot P(B|A) \cdot P(A)$

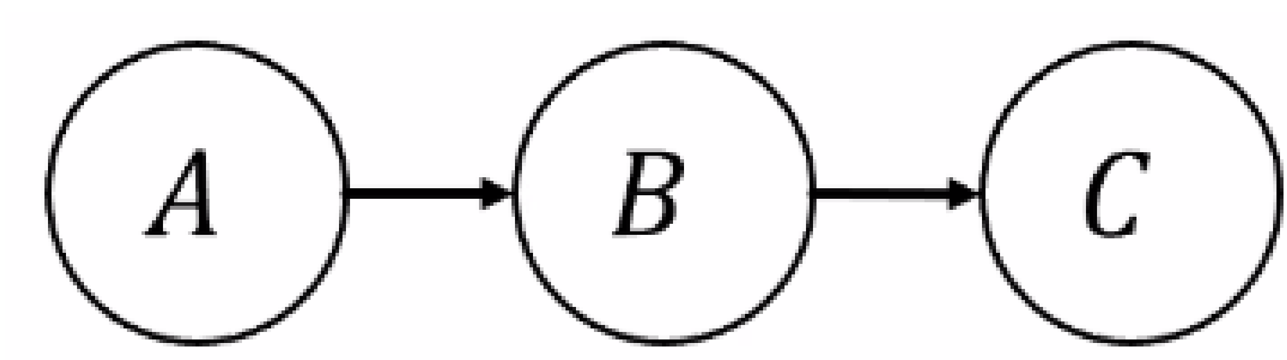
- B and C are conditionally independent given A



Example: causal chain

$P(A, B, C) = P(C|B) \cdot P(B|A) \cdot P(A)$

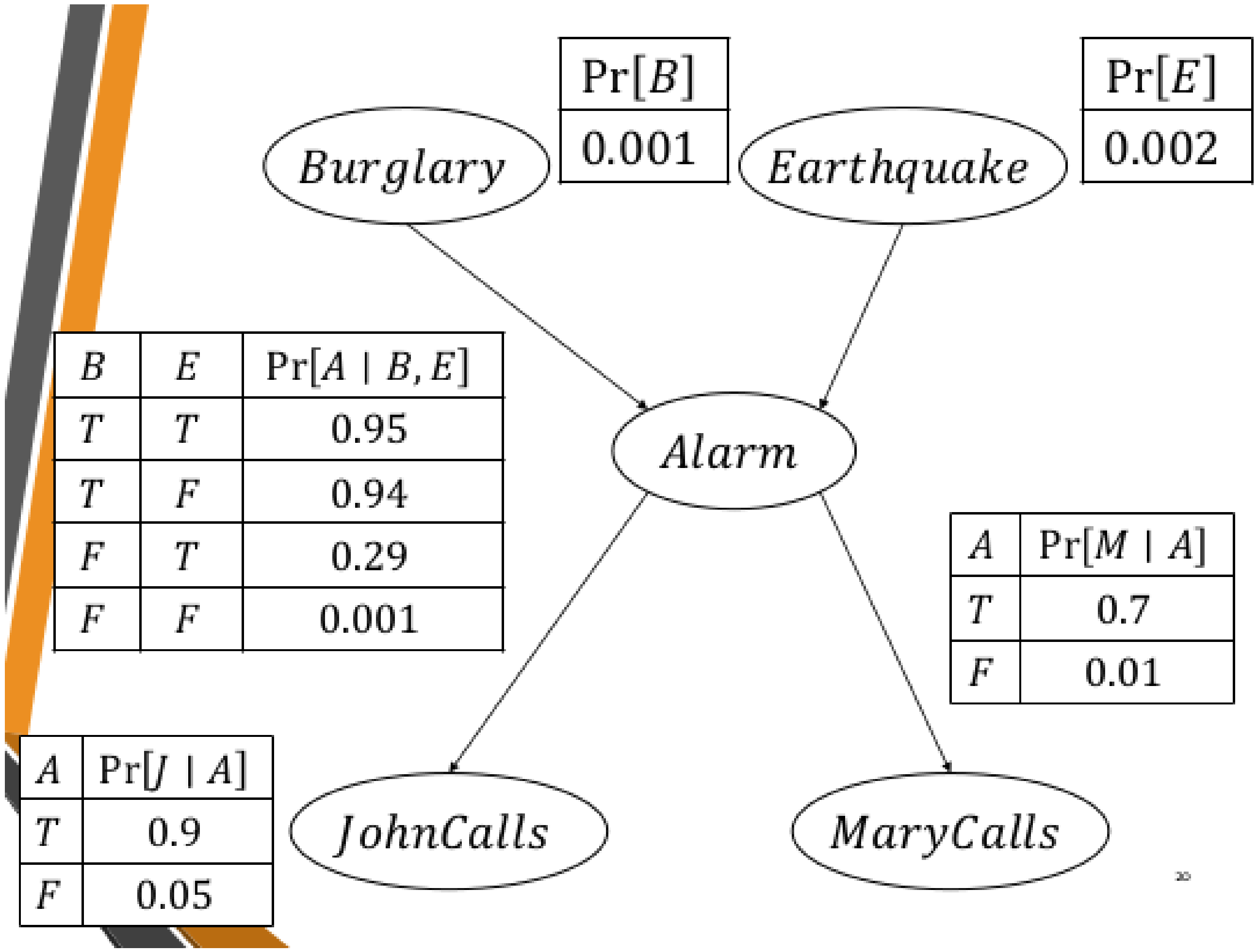
- C is conditionally independent of A given B – note that $P(C|B) = P(C|B, A)$



Example: burglary

- A: Alarm goes off
- E: Alarm sometimes set off by minor earthquake
- B: Alarm set off by burglar

- J: John calls to say my house alarm is ringing
- M: Mary calls to say my house alarm is ringing



B	E	Pr[A | B, E]
T	T	0.95
T	F	0.94
F	T	0.29
F	F	0.001

A	Pr[M | A]
T	0.7
F	0.01

A	Pr[J | A]
T	0.9
F	0.05

$P(B=1|J=1,M=0) = \frac{P(B=1,J=1,M=0)}{P(J=1,M=0)} = ?$

- To find $P(B=1,J=1,M=0)$: sum over 4 cases of A=0/1, E=0/1
- To find $P(J=1,M=0)$: sum over 8 cases of A=0/1, E=0/1, B=0/1
- whereby $P(J,M,A,B,E) = P(J|A) \cdot P(M|A) \cdot P(A|B,E) \cdot P(B) \cdot P(E)$

10.2 Inference in Bayesian Networks

Bayesian network represents the full joint distribution.

Infer any query by summing over all cases of the other variables.

10.3 Algorithm for Constructing Bayesian Network

Algorithm

- Choose an ordering for variables $X_1, \ldots, X_n$
- For i=1 to n:
  - Add node X_i to network
  - Select minimal set of parents from $X_1, \ldots, X_{i-1}$ such that $P(X_i|Parents(X_i)) = P(X_i|X_1, \ldots, X_{i-1})$
  - Add edges from every parent to X_i

– Write down CPT for $P(X_i|Parents(X_i))$

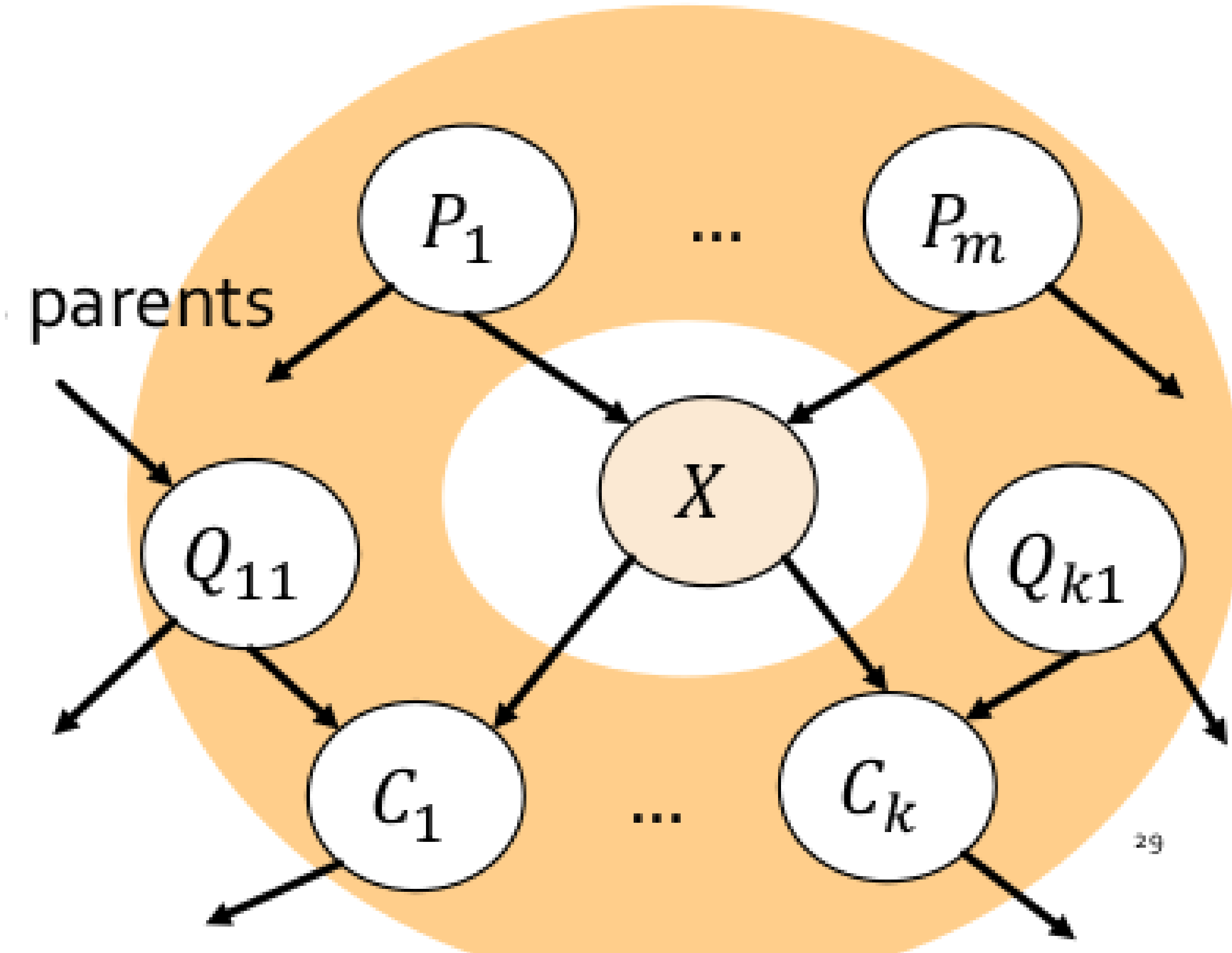
Variable order matters!

- Choosing a 'good' variable order can reduce the number of edges required

10.4 Markov Blanket

A node is conditionally independent of everything else given the values of its:

- Parents
- Children
- Children's parents



10.5 d-Separation

Given variables X and Y and known variables $\epsilon = \{E_1, \ldots, E_k\}$, are X and Y surely independent given ϵ?

Idea: any general graph can be broken down into three cases (causal chain/common cause/common effect) to determine conditional independence of X and Y given knowledge of ϵ

Check every undirected path between X and Y, ignoring direction of arcs

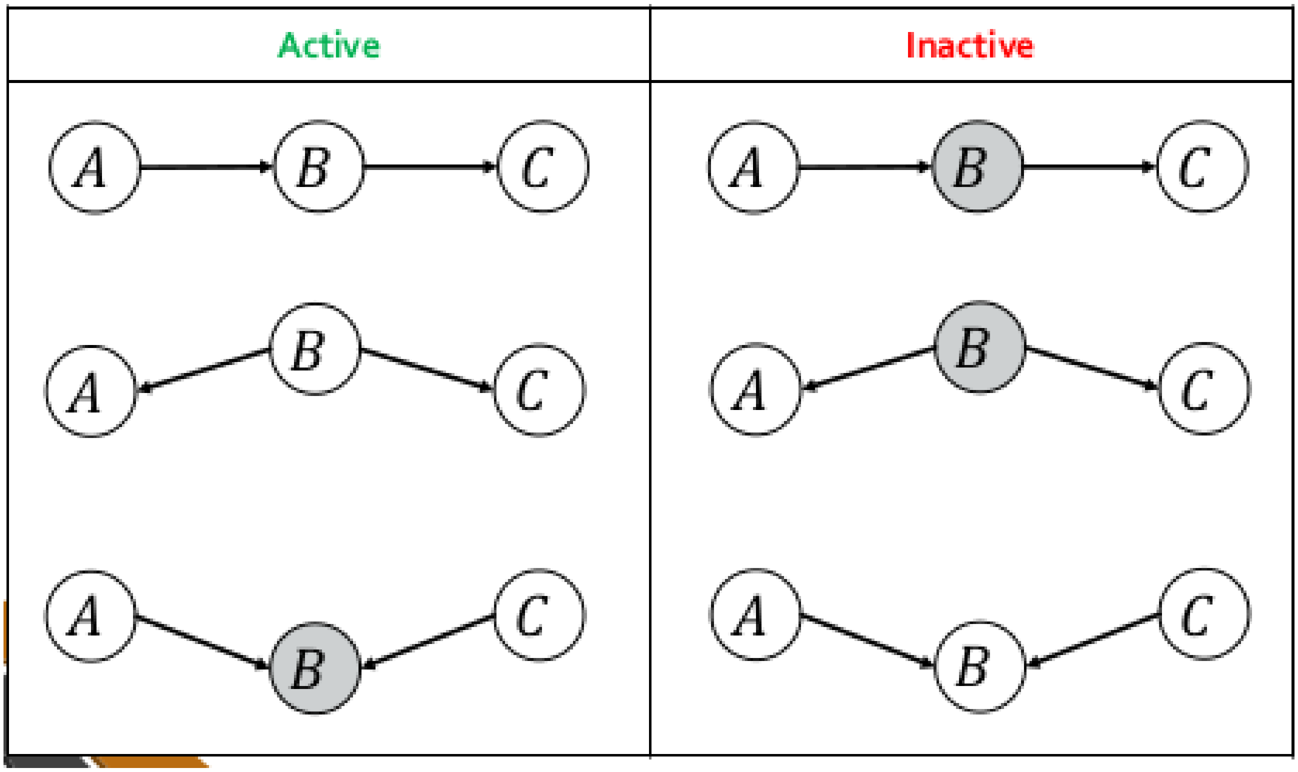
- ($\star$) If all paths are not active, then X and Y are independent given ϵ

Active path: Path is active iff every triple on the path is active

- I.e. if *any* triple on the path is inactive, the *entire path* is inactive

Active triple: see the chart

- Dark means we know B, light means we don't know B
- Note: only take into account knowledge of B, not A or C in these triples



Example

- Here, all 3 potential paths are inactive => 2 red-marked nodes are independent given ϵ