> AI is a shit, yet tasty. ——Anonymous

# Artificial Intelligence

*All rights reserved by [CWHer]*

## Content


- **Artificial Intelligence**
  - Content
  - Chapter 1: Introduction (*)
    - Intelligent Agents
  - Chapter 2: Uninformed Search
    - Search Problems
    - Representation
    - Tree Search
      - DFS
      - BFS
      - UCS
  - Chapter 3: Informed Search
    - Heuristics
      - Greedy Search
      - A* Search
        - Admissible Heuristics
    - Graph Search (*)
      - Consistency of Heuristics
    - Conclusion
  - Chapter 4: Constraint Satisfaction Problems (CSP)
    - CSP
    - Solving CSPs
      - Backtracking Search
      - Improvement: Filtering
        - Forward Checking
        - Constraint Propagation
      - Improvement: Ordering
      - Improvement: Structure
        - Independent Subproblems
        - Tree-Structured
        - Improving Structure (*)
    - Non-binary CSPs (*)
    - Local Search
      - Hill Climbing
      - Simulated Annealing (*)
      - Genetic Algorithms (*)
      - Summary
  - Chapter 5: Adversarial Search
    - Game Type
    - Minimax Search
      - Alpha-Beta Pruning
      - Bounded Lookahead
      - Generalized Minimax
    - Modeling Assumptions
    - Utilities

- Human Utilities (*)
- Chapter 6: Markov Decision Process (MDP)
  - Markov Decision Process
  - Solving MDPs
    - Expectimax
    - Value Iteration
    - Policy Iteration
    - Summary
- Chapter 7: Reinforcement Learning
  - Passive Reinforcement Learning
    - Model-based Passive RL
    - Model-free Passive RL
      - Value Learning
        - Direct Evaluation (MC Method)
        - TD-Learning
      - Q Learning
  - Active Reinforcement Learning
    - How to Explore
    - Regret (*)
  - Approximate Reinforcement Learning (*)
    - Approximate Q Learning
    - Policy Search
  - Summary
- Chapter 8: Regression
  - Linear Regression
    - Normal Equation
    - Gradient Methods
    - Probabilistic View of MSE
    - Regularization
    - Probabilistic View of L2 Regularization
  - Logistic Regression
    - Loss Function
    - Binary Classification
    - Multi-Class Classification (*)
- Chapter 9: Neural Network
  - Activation Functions
    - Sigmoid
    - Tanh
    - Rectified Linear Unit (ReLU)
  - Neural networks
  - Backpropagation
    - Using New Weights
    - Using Old Weights
    - Misc (*)
  - Overfitting (*)
- Chapter 10: Bayes Net Probabilistic Models
  - Probability
  - Probabilistic Inference
  - Build Joint Distribution Using Chain Rule
  - Conditional Independence
  - Bayes' Nets
    - Conditional Independence Semantics
      - Causal Chain
      - Common Causes
      - Common Effect
    - Bayes Nets: Independence
      - Bayes Ball / D-separation Algorithm
    - Structure Implications (*)

Summary
Chapter 11: Bayes Net Inference
Bayes Rule
Inference
Inference by Enumeration
Variable Elimination (VE)
Ordering
Sampling (*)
Prior Sampling
Rejection Sampling
Likelihood Reweighting
Gibbs Sampling
Reference
Demos
HW & Lab Recap
Appendix: Matrix Gradient

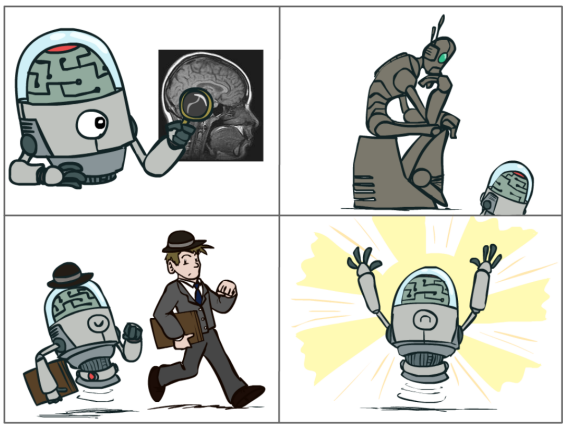
**(*)**: no covered in the final exam

# Chapter 1: Introduction (*)

## What is AI?

The science of making machines that:

Think like people



Think rationally

Act like people

Act rationally

Common definition: Machine Learning is the study of computer algorithms that improve automatically through experience.

AI: create intelligent machines that can simulate human thinking capability and behavior.

ML: allow machine to learn from data without being programmed explicitly.

ML $\subset$ AI, AI without ML: A*

## Intelligent Agents

An agent is an entity that perceives and acts

Rational: select actions to maximize (expected) utility

Environment example:

> Pacman Env: fully observable, stochastic (unknown ghost), static (ghost never changes), discrete

**Simple reflex agent**

Select actions on the basis of the current percept, ignoring the rest of the percept history.

Do not consider the future consequences of their actions

condition–action rule: if condition, then action

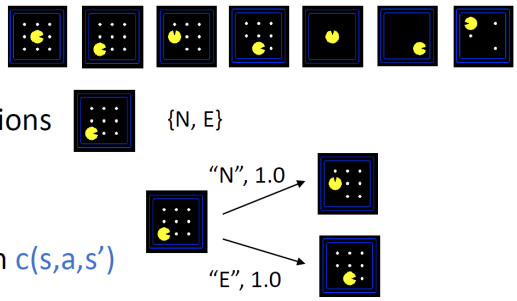
# Chapter 2: Uninformed Search

Search operates over models of the world. Planning is all “in simulation”.

## Search Problems

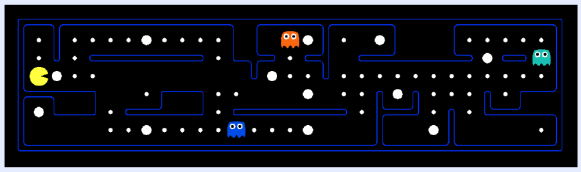
### Search Problems

- A search problem consists of:
  - A state space
  - For each state, a set Actions(s) of successors/actions
  - A transition model T(s,a)
  - A step cost(reward) function c(s,a,s’)
  - A start state and a goal test
- A solution is a sequence of actions (a plan) which transforms the start state to a goal state




### What’s in a State Space?

The world state includes every last detail of the environment



A search state keeps only the details needed for planning (abstraction)

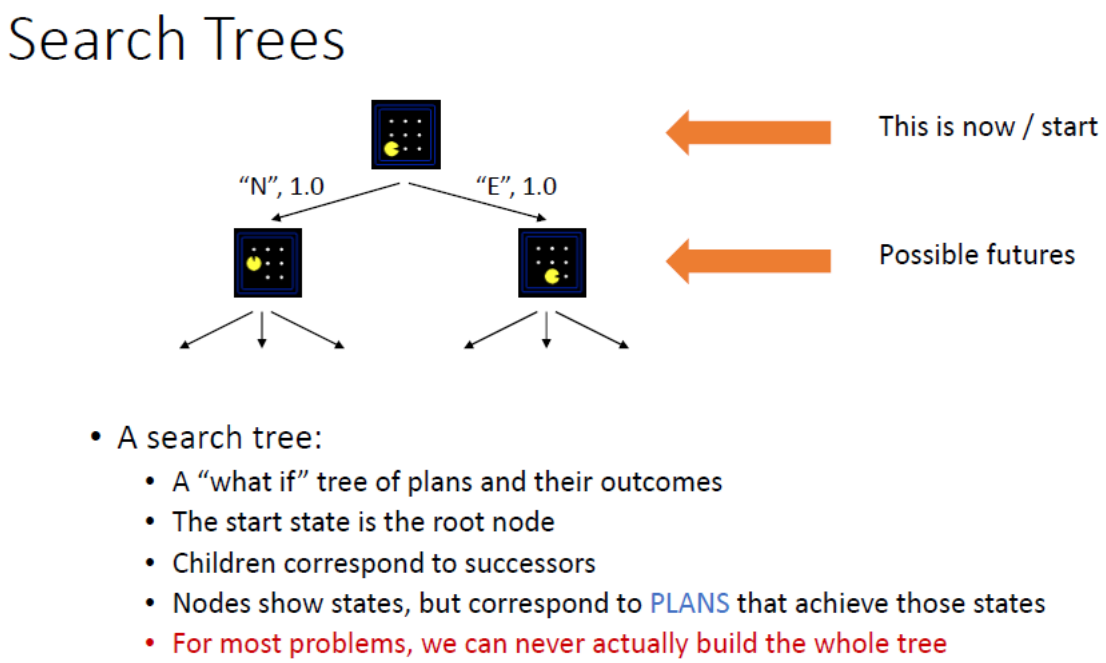
- Problem: Pathing
  - States: (x,y) location
  - Actions: NSEW
  - Successor: update location only
  - Goal test: is (x,y)=END
- Problem: Eat-All-Dots
  - States: {(x,y), dot booleans}
  - Actions: NSEW
  - Successor: update location and possibly a dot boolean
  - Goal test: dots all false

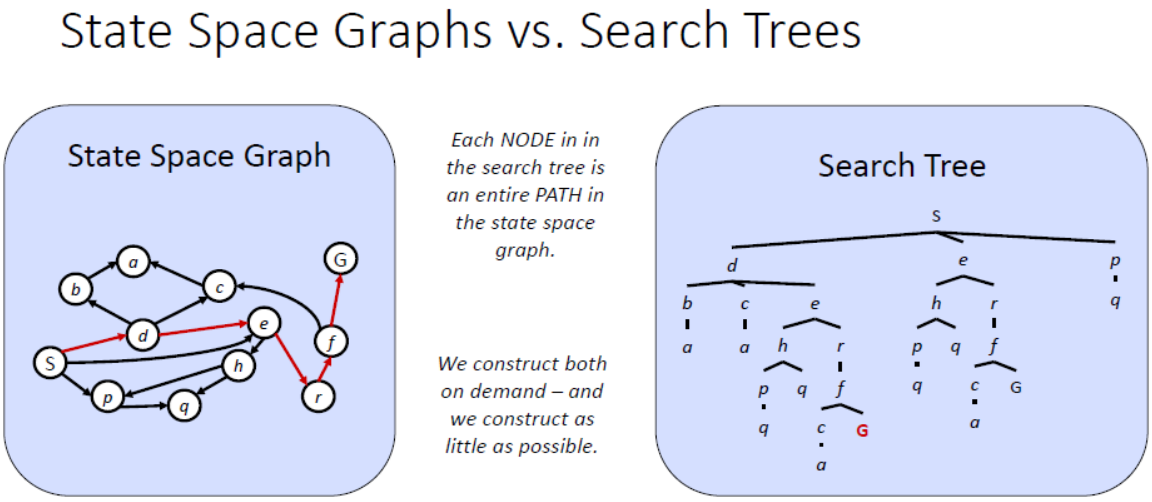
## Representation

State Space Graphs:

- Nodes are (abstracted) world configurations (states)
- Arcs represent successors (action results)

NOTE: In a state space graph, each state occurs only once!




NOTE: State can occur more than once in the search tree. (Lots of repeated structure)

## Tree Search

1. Expand out potential plans (tree nodes)
2. Maintain a <u>fringe</u> of partial plans under consideration
3. Try to expand as few tree nodes as possible

Main question: which fringe nodes to explore?

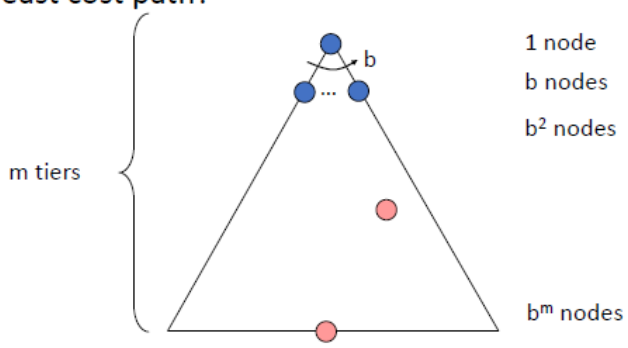
```
function TREE_SEARCH(problem) returns a solution, or failure

    initialize the frontier as a specific work list (stack, queue, priority queue)
    add initial state of problem to frontier
    loop do
        if the frontier is empty then
            return failure
        choose a node and remove it from the frontier
        if the node contains a goal state then
            return the corresponding solution
        for each resulting child from node
            add child to the frontier
```

## Search Algorithm Properties

- Complete: Guaranteed to find a solution if one exists?
- Optimal: Guaranteed to find the least cost path?
- Time complexity?
- Space complexity?

- Cartoon of search tree:
  - b is the branching factor
  - m is the maximum depth
  - solutions at various depths




- Number of nodes in entire tree?
  - $1 + b + b^2 + \ldots + b^m = O(b^m)$

### DFS

Implementation: Fringe is a LIFO stack.

What nodes DFS expand: Some left prefix of the tree.

Complete: m could be infinite, only if we prevent cycles

Optimal: No, the "leftmost" solution

Space: Only has siblings on path to root, so O(bm)

### BFS

Implementation: Fringe is a FIFO queue

What nodes does BFS expand: Processes all nodes above shallowest solution

Complete: s must be finite

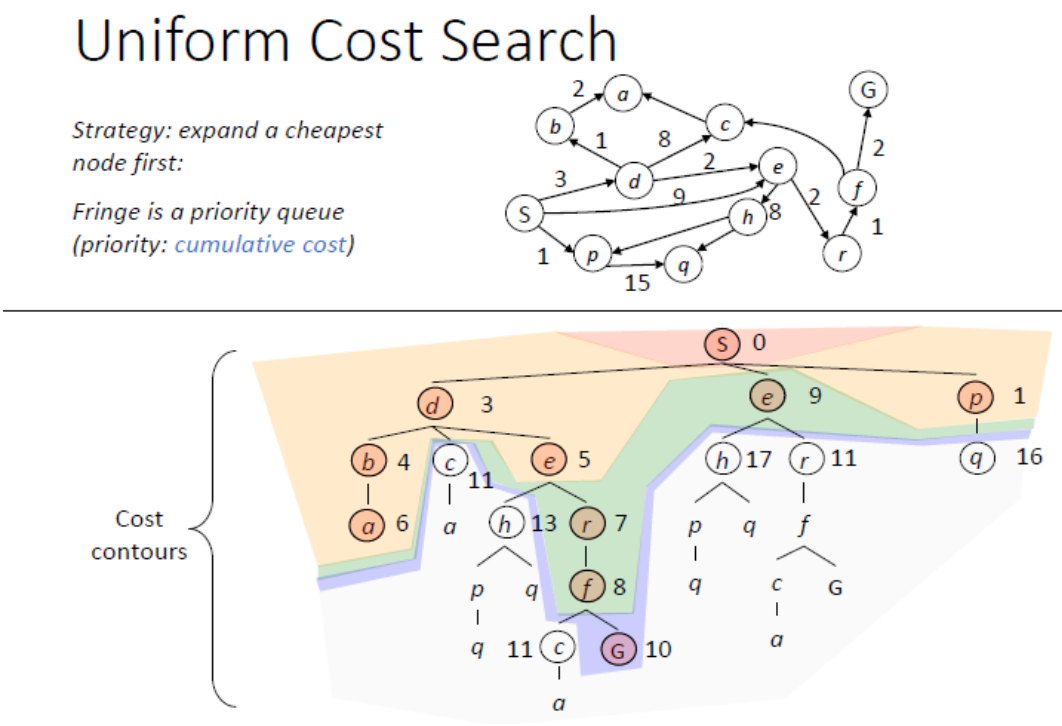
Optimal: Only if costs are all 1

Space: $O(b^S)$

**Iterative Deepening**: DFS's space advantage + BFS's time advantage

### UCS

BFS finds the shortest path in terms of number of actions, but not the least cost path




What nodes does UCS expand: Processes all nodes with cost less than cheapest solution

Space: if solution = C* and arcs >= e, then the effective depth is roughly C*/e. $O(b^{C^*/e})$

complete and optimal

Problems:

- Explores options in every “direction”
- No information about goal location

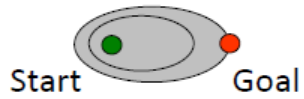
# Chapter 3: Informed Search

## Heuristics

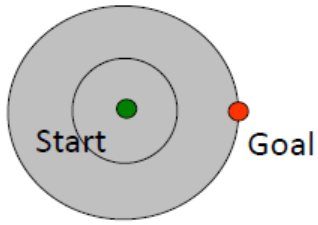
- A function that estimates how close a state is to a goal
- Designed for a particular search problem
- Examples: Manhattan distance, Euclidean distance for pathing

# Effect of heuristics

- Guide search ***towards the goal*** instead of ***all over the place***




Informed




Uninformed

### Greedy Search

Strategy: expand the node with least h(n) value

Heuristics might be wrong

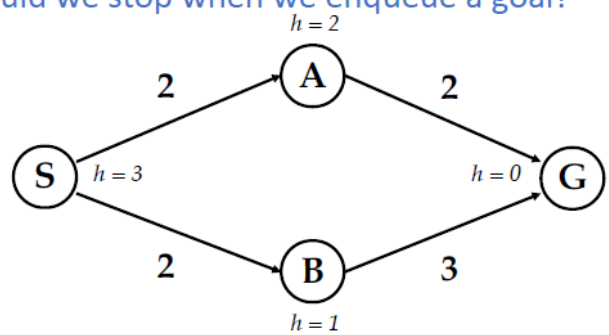
### A* Search

combining UCS and Greedy

A* Search orders by the sum: f(n) = g(n) + h(n)

# When should A* terminate?

- Should we stop when we enqueue a goal?

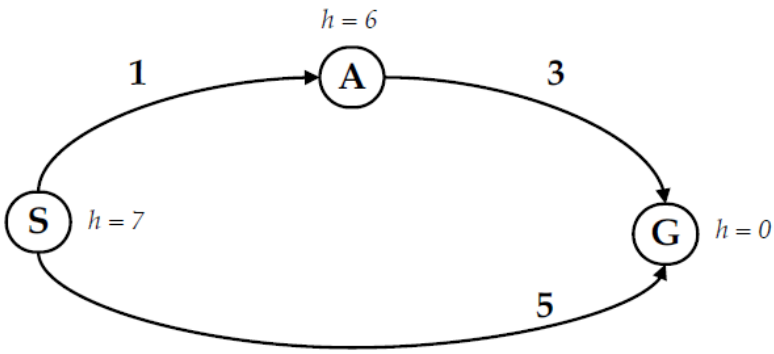



| | g | h | + |
|---|---|---|---|
| ~~S~~ | ~~0~~ | ~~3~~ | ~~3~~ |
| ~~S->A~~ | ~~2~~ | ~~2~~ | ~~4~~ |
| ~~S->B~~ | ~~2~~ | ~~1~~ | ~~3~~ |
| S->B->G | 5 | 0 | 5 |
| S->A->G | 4 | 0 | 4 |

- No: only stop when we dequeue a goal

**Admissible Heuristics**

# Is A* Optimal?




| | g | h | + |
|---|---|---|---|
| ~~S~~ | ~~0~~ | ~~7~~ | ~~7~~ |
| S->A | 1 | 6 | 7 |
| S->G | 5 | 0 | 5 |

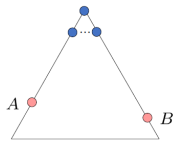
- What went wrong?
- Actual bad goal cost < estimated good goal cost
- We need estimates to be less than actual costs!

Admissible (optimistic) heuristics slow down bad plans but never outweigh true costs.

A heuristic h is admissible (optimistic) if 0<= h(n) <= h*(n), where h*(n) is the true cost to a nearest goal.

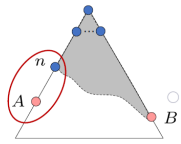
## Optimality of A* Tree Search

- Assume:
  - A is an optimal goal node
  - B is a suboptimal goal node
  - h is admissible
- Claim:
  - A will exit the fringe before B




## Optimality of A* Tree Search: Blocking

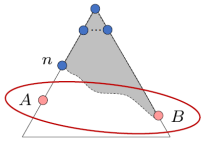
- Proof:
  - Imagine B is on the fringe
  - Some ancestor n of A is on the fringe, too (maybe A!)
  - Claim: n will be expanded before B
    1. f(n) is less or equal to f(A)




| | |
|---|---|
| $f(n) = g(n) + h(n)$ | Definition of f-cost |
| $f(n) \leq g(A)$ | Admissibility of h |
| $g(A) = f(A)$ | h = 0 at a goal |

## Optimality of A* Tree Search: Blocking 2

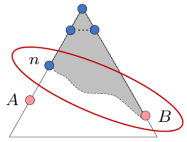
- Proof:
  - Imagine B is on the fringe
  - Some ancestor n of A is on the fringe, too (maybe A!)
  - Claim: n will be expanded before B
    1. f(n) is less or equal to f(A)
    2. f(A) is less than f(B)




| | |
|---|---|
| $g(A) < g(B)$ | B is suboptimal |
| $f(A) < f(B)$ | h = 0 at a goal |

## Optimality of A* Tree Search: Blocking 3

- Proof:
  - Imagine B is on the fringe
  - Some ancestor n of A is on the fringe, too (maybe A!)
  - Claim: n will be expanded before B
    1. f(n) is less or equal to f(A)
    2. f(A) is less than f(B)
    3. *n* expands before B
  - All ancestors of A expand before B
  - A expands before B
  - A* search is optimal

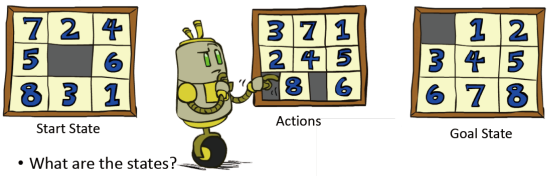


$f(n) \leq f(A) < f(B)$

Most of the work in solving hard search problems optimally is in coming up with admissible heuristics.

Often, admissible heuristics are solutions to relaxed problems, where new actions are available

## Example: 8 Puzzle

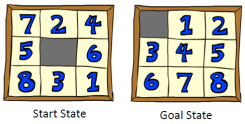


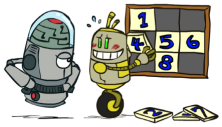

- What are the states?
- How many states?
- What are the actions?
- How many successors from the start state?
- What should the costs be?

Admissible heuristics?

## Example: 8 Puzzle - 2

- Heuristic: Number of tiles misplaced
- Why is it admissible?
- h(start) = 8
- This is a relaxed-problem heuristic



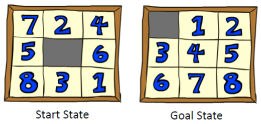



| | Average nodes expanded when the optimal path has... | | |
|---|---|---|---|
| | ...4 steps | ...8 steps | ...12 steps |
| UCS | 112 | 6,300 | $3.6 \times 10^6$ |
| TILES | 13 | 39 | 227 |

Statistics from Andrew Moore

## Example: 8 Puzzle - 3

- What if we had an easier 8-puzzle where any tile could slide any direction at any time, ignoring other tiles?

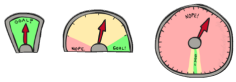



- Total Manhattan distance
- Why is it admissible?
- h(start) = 3 + 1 + 2 + ... = 18

| | Average nodes expanded when the optimal path has... | | |
|---|---|---|---|
| | ...4 steps | ...8 steps | ...12 steps |
| TILES | 13 | 39 | 227 |
| MANHATTAN | 12 | 25 | 73 |

## Example: 8 Puzzle - 4

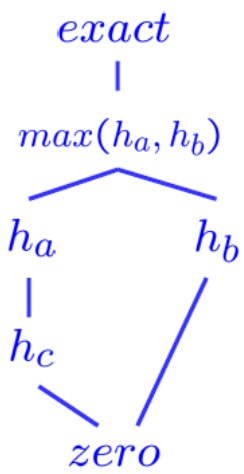
- How about using the actual cost as a heuristic?
  - Would it be admissible?
  - Would we save on nodes expanded?
  - What's wrong with it?



- With A*: a trade-off between quality of estimate and work per node
  - As heuristics get closer to the true cost, you will expand fewer nodes but usually do more work per node to compute the heuristic itself

# Combining Heuristics, Dominance

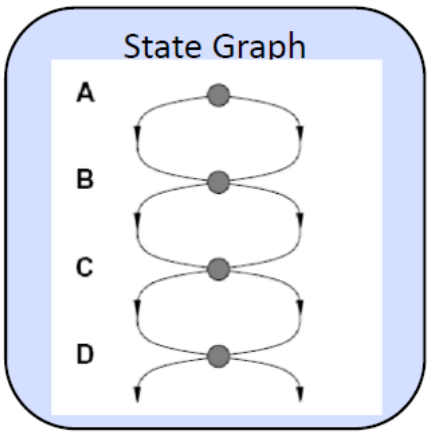
- Dominance: $h_a \geq h_c$ if
  $$\forall n : h_a(n) \geq h_c(n)$$
  - Roughly speaking, larger is better as long as both are admissible
- Heuristics form a semi-lattice:
  - Max of admissible heuristics is admissible
  $$h(n) = max(h_a(n), h_b(n))$$
- Trivial heuristics
  - Bottom of lattice is the zero heuristic (what does this give us?)
  - Top of lattice is the exact heuristic, but usually too expensive

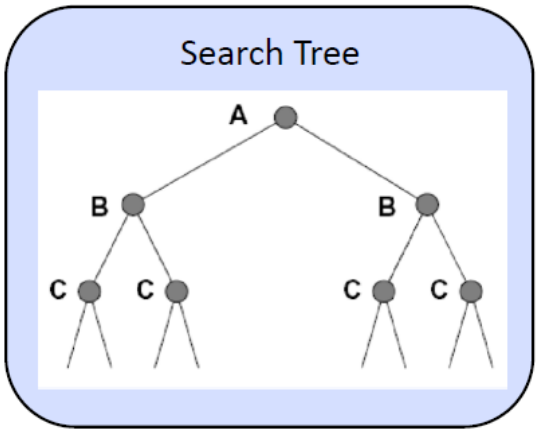



## Graph Search (*)

# Tree Search: Extra Work!

- Failure to detect repeated states can cause exponentially more work



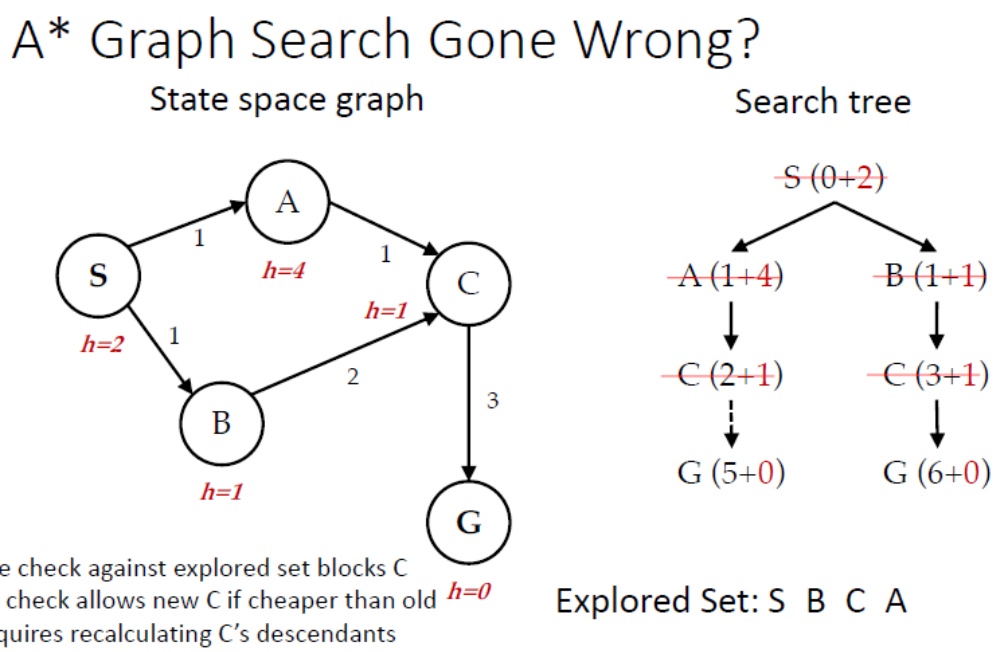
Idea: never expand a state twice

Important: store the closed/explored set as a set, not a list

- This graph search algorithm overlays a tree on a graph
- The frontier states separate the explored states from never seen states

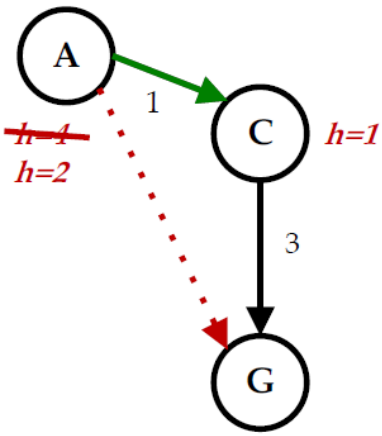
```
function A-STAR-GRAPH-SEARCH(problem) returns a solution, or failure
  initialize the explored set to be empty
  initialize the frontier as a priority queue using f(n) = g(n) + h(n) as the priority
  add initial state of problem to frontier with priority f(S) = 0 + h(S)
  loop do
      if the frontier is empty then
          return failure
      choose a node and remove it from the frontier
      if the node contains a goal state then
          return the corresponding solution
      add the node state to the explored set
      for each resulting child from node
          if the child state is not already in the frontier or explored set then
              add child to the frontier
          else if the child is already in the frontier with higher f(n) then
              replace that frontier node with child
```

## Consistency of Heuristics



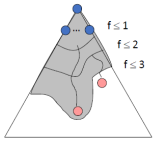
# Consistency of Heuristics

- 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)
    - triangle inequality: h(A) ≤ c(A-C) + h(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



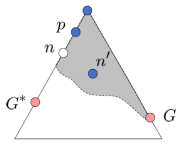

## Optimality of A* Graph Search

- Sketch: consider what A* does with a consistent heuristic:
  - 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




## Optimality of A* Graph Search: Proof

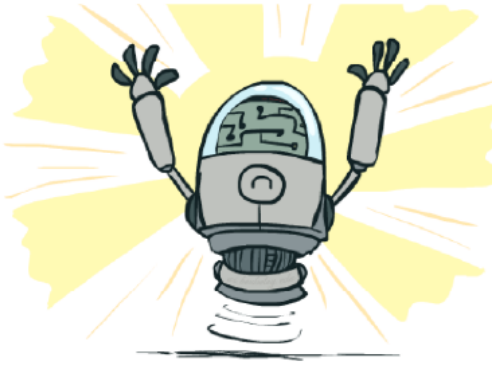
- For any $n$ on path to $G^*$, let $n'$ be a worse node for the same state
- Let $p$ be the ancestor of $n$ that was on the queue when $n'$ was added in the queue
- Claim: $p$ will be expanded before $n'$
  - $f(p) \leq f(n)$ because of consistency
  - $f(n) < f(n')$ because $n'$ is suboptimal
  - $p$ would have been expanded before $n'$
- Thus $n$ will be expanded before $n'$
- All ancestors of $G^*$ are not blocked




## Conclusion

# Optimality of A* Search

- 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
- In general, most natural admissible heuristics tend to be consistent, especially if from relaxed problems



# Chapter 4: Constraint Satisfaction Problems (CSP)

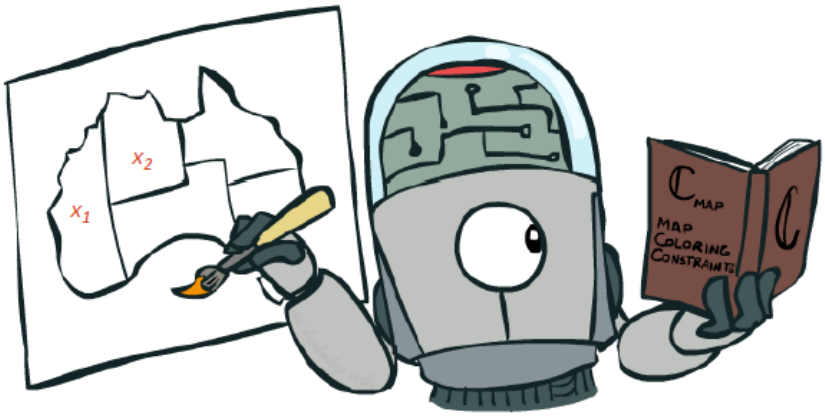
# CSP

## Constraint Satisfaction Problems

*N variables*

*domain D*

*constraints*




| *states* | *goal test* | *successor function* |
|---|---|---|
| *partial assignment* | *complete; satisfies constraints* | *assign an unassigned variable* |

**Planning**: sequences of actions

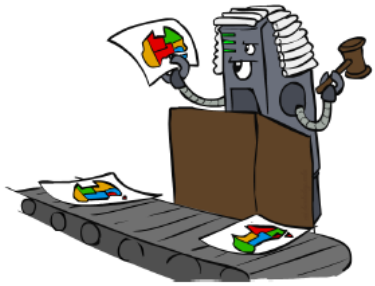
- The path to the goal is the important thing
- Paths have various costs, depths
- Heuristics give problem specific guidance

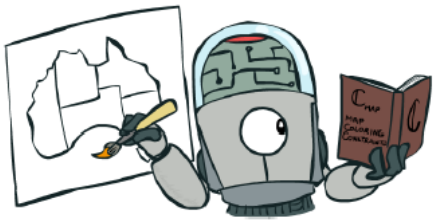
**Identification**: assignments to variables

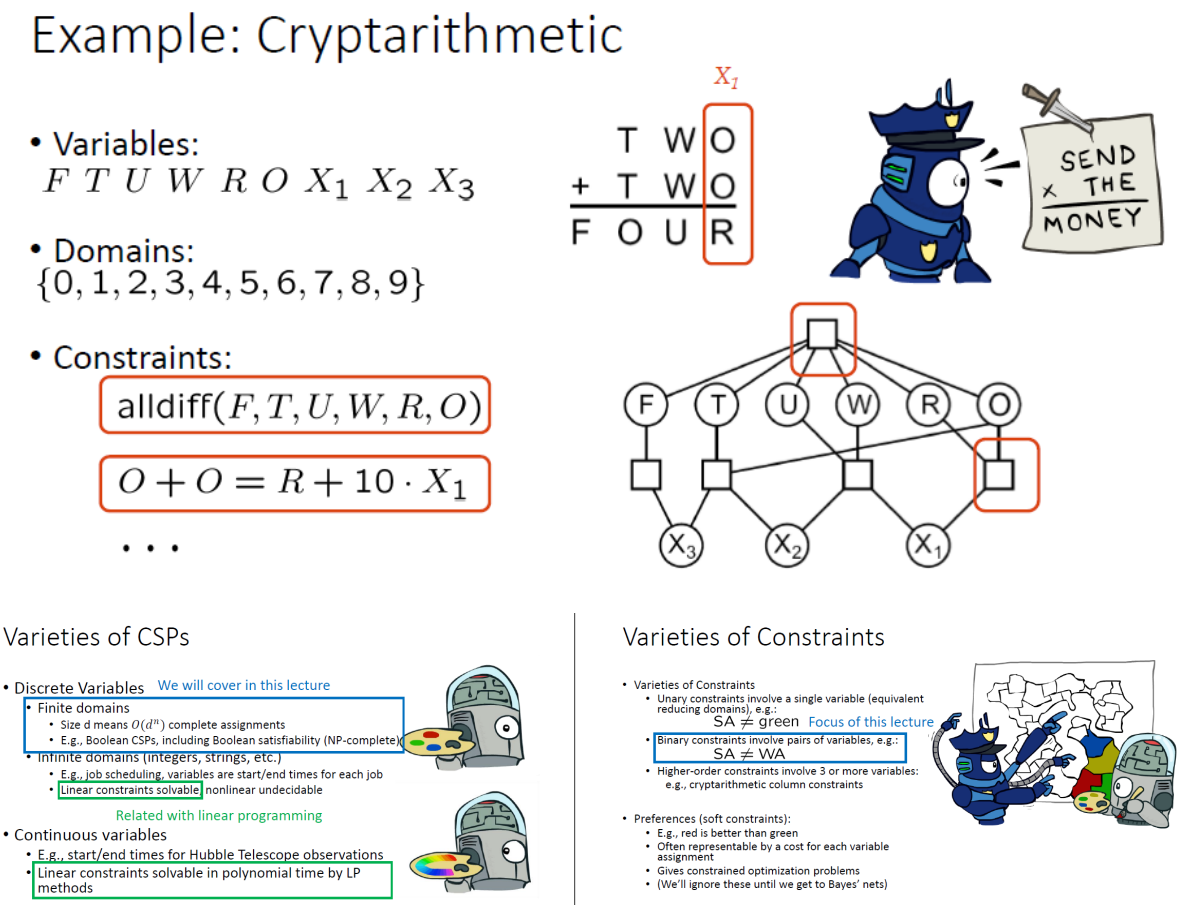
- The goal itself is important, not the path
- All paths at the same depth (for some formulations)
- CSPs are specialized for identification problems

## Constraint Satisfaction Problems

- Standard search problems:
  - State is a "black box": arbitrary data structure
  - Goal test can be any function over states
  - Successor function can also be anything
- Constraint satisfaction problems (CSPs):
  - A special subset of search problems
  - State is defined by variables $X_i$ with values from a domain D (sometimes D depends on i)
  - Goal test is a set of constraints specifying allowable combinations of values for subsets of variables
- Allows useful general-purpose algorithms with more power than standard search algorithms








Binary constraint graph: nodes are variables, arcs show constraints.

Use the graph structure to speed up search.

## Solving CSPs

States defined by the values assigned so far (partial assignments)

- Initial state: {}
- Successor function: assign a value to an unassigned variable
- Goal test: the current assignment is complete and satisfies all constraints

CSP is trivial when # constraints >> # variables or # constraints << # variables.

Naive Approach: Check feasibility only when the assignment is complete.

### Backtracking Search

Basic uninformed algorithm

Backtracking Search = DFS + 2 improvements

- One variable at a time (fix ordering)
- Check constraints as you go

  Consider only values which do not conflict previous assignments

```
function BACKTRACKING_SEARCH(csp) returns a solution, or failure
  return RECURSIVE_BACKTRACKING({}, csp)

function RECURSIVE_BACKTRACKING(assignment, csp) returns a solution, or 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
```

No need to check consistency for a complete assignment

What are choice points?

Checks consistency at each assignment

Backtracking = DFS + variable-ordering + fail-on-violation

## Improvement: Filtering

Q: Can we detect inevitable failure early?

A: Keep track of domains for unassigned variables and cross off bad options

**Forward Checking**

Cross off values that violate a constraint when added to the existing assignment.

Failure is detected if some variables have no values remaining

**Constraint Propagation**

Constraint propagation: reason from constraint to constraint

An arc `X->Y` is **consistent** iff <u>for every x in the tail</u> <u>there is some y in the head</u> which could be assigned without violating a constraint

Remember: Delete from the <u>tail</u>!

Forward checking: Enforcing consistency of arcs <u>pointing to</u> each new assignment. (special case)

Important: If X loses a value, neighbors of X need to be rechecked!

```
function AC-3(csp) returns the CSP, possibly with reduced domains
   initialize a queue of all the arcs in csp
   while queue is not empty do
      (X_i, X_j) ← REMOVE_FIRST(queue)
      if REMOVE_INCONSISTENT_VALUES(X_i, X_j) then
         for each X_k in NEIGHBORS[X_i] do
            add (X_k, X_i) to queue
```

- An arc is added after a removal of value at a node
- $n$ node in total, each has $\leq d$ values
- Total times of removal: $O(nd)$
- After a removal, $\leq n$ arcs added
- Total times of adding arcs: $O(n^2 d)$

```
function REMOVE_INCONSISTENT_VALUES(X_i, X_j) returns true iff succeeds
   removed ← false
   for each x in DOMAIN[X_i] do
      if no value y in DOMAIN[X_j] allows (x,y) to satisfy the constraint X_i ↔ X_j then
         delete x from DOMAIN[X_i]; removed ← true
   return removed
```

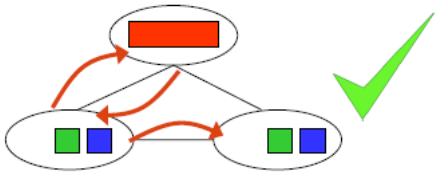
- Check arc consistency per arc: $O(d^2)$
- Complexity: $O(n^2 d^3)$
- Can be improved to $O(n^2 d^2)$

... but detecting all possible future problems is NP-hard – why?

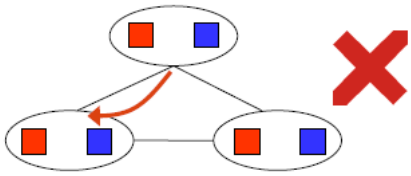
Arc consistency detects failure earlier than forward checking.

# 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)

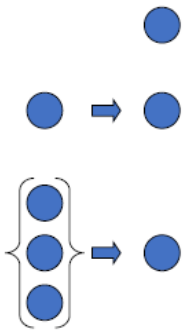


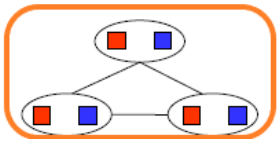
- Arc consistency still runs inside a backtracking search!
- And will be called many times



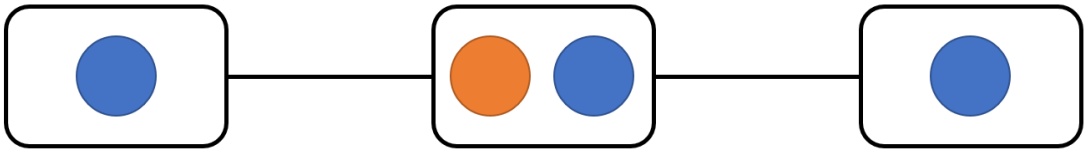
# K-Consistency

- Increasing degrees of 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 $k^{th}$ node.
- Higher k more expensive to compute
- (You need to know the k=2 case: arc consistency)





Note: $k$ consistency and $k-1$ consistency are irrelevant.



Strong K-Consistency

- 1~k consistency
- can solve without backtracking (2-consistency, 3-consistency, ...)

## Improvement: Ordering

Which variable should be assigned next?

- Minimum Remaining Values
- Most constrained variable & Fail-fast ordering

In what order should its values be tried?

- Least Constraining Value
- I.e., the one that rules out the fewest values in the remaining variables (affect as little as possible to others)

Why different strategies?

- Must face the difficulty raised by variable. (detect failure early)
- Then, try to avoid the difficulty raised by value. (try to avoid failure)

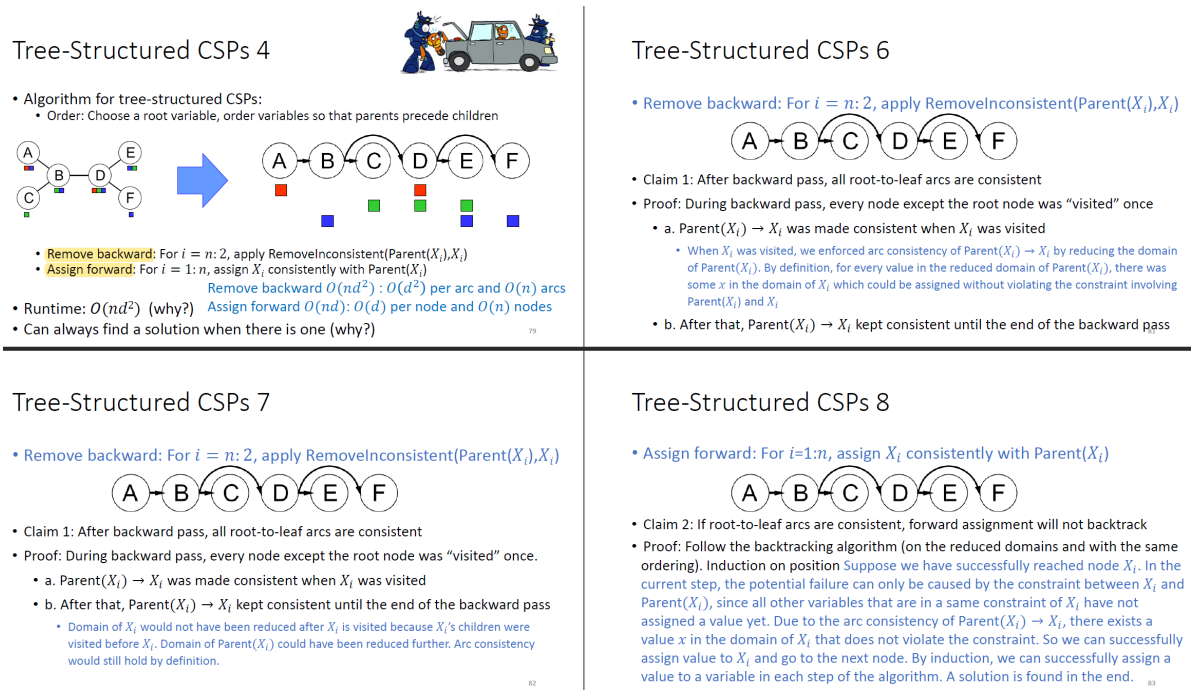
## Improvement: Structure

### Independent Subproblems

Connected components of constraint graph

### Tree-Structured

Theorem: if the constraint graph has no loops, the CSP can be solved in O(nd$^2$) time

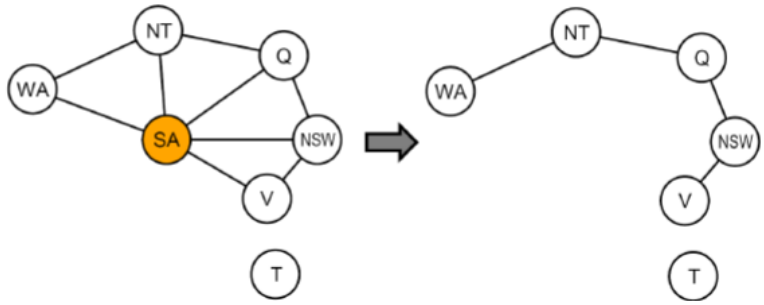


Q: What if there are cycles?

A: Assign forward may fail. We need to make sure the constraints involving the current node and <u>any parent node</u> is satisfied, which could be impossible.

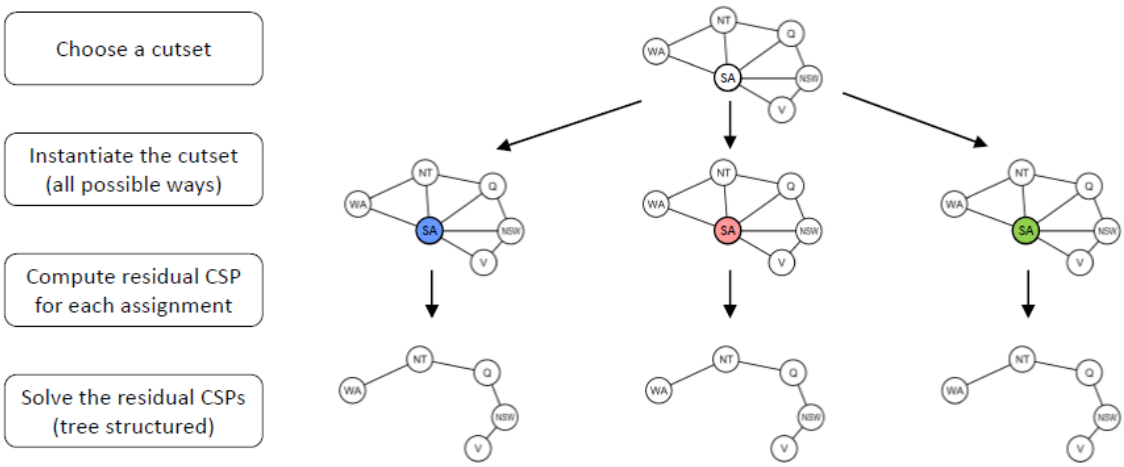
**Improving Structure (*)**

## Nearly Tree-Structured CSPs

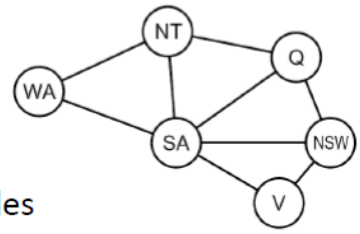



- Conditioning: instantiate a variable, prune its neighbors' domains
- Cutset conditioning: instantiate (in all ways) a set of variables such that the remaining constraint graph is a tree
- Cutset size c gives runtime O( ($d^c$) (n-c) $d^2$ ), very fast for small c

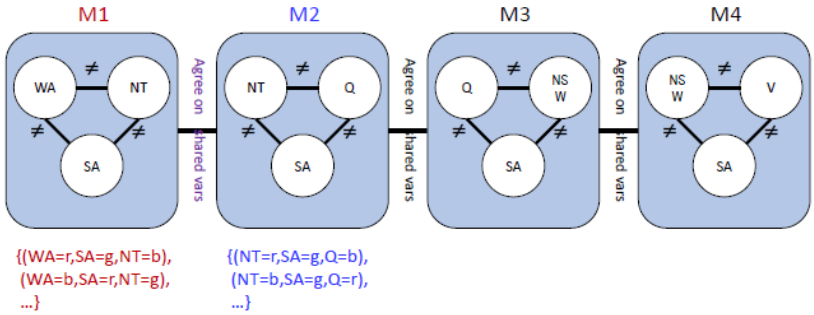
## Cutset Conditioning



## 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




91

## Non-binary CSPs (*)

## Constraint graph for non-binary CSPs

- Variable nodes: nodes to represent the variables
- Constraint nodes: auxiliary nodes to represent the constraints
- Edges: connects a constraint node and its corresponding variables

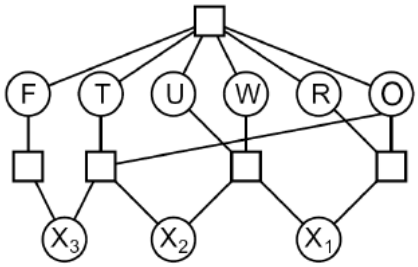
- Constraints:

  alldiff$(F, T, U, W, R, O)$

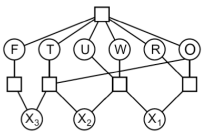
  $O + O = R + 10 \cdot X_1$

  $\cdots$

```
  T W O
+ T W O
-------
F O U R
```




### Solve non-binary CSPs




- Naïve search?
  - Yes!
- Backtracking?
  - Yes!
- Forward Checking?
  - Need to generalize the original FC operation
  - (nFC0) After a variable is assigned a value, find all constraints with only one unassigned variable and cross off values of that unassigned variable which violate the constraint
  - There exist other ways to do generalized forward checking

### Solve non-binary CSPs 2

- AC-3? Need to generalize the definition of AC and enforcement of AC
- Generalized arc-consistency (GAC)
  - A non-binary constraint is GAC iff for every value for a variable there exist consistent value combinations for all other variables in the constraint
  - Reduced to AC for binary constraints
- Enforcing GAC
  - Simple schema: enumerate value combination for all other variables
  - $O(d^k)$ on $k$-ary constraint on variables with domains of size $d$
- There are other algorithms for non-binary constraint propagation, e.g., (i,j)-consistency [Freuder, JACM 85]

## Local Search

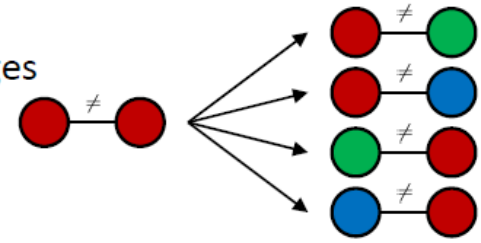
Can be applied to identification problems (e.g., CSPs), as well as some planning and optimization problems

Use a complete-state formulation.

Note: An assignment is complete means that all variables are assigned a value. An algorithm is complete means that it will output a solution if there exists one.

# Local Search vs Tree Search

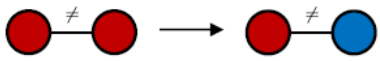
- Tree search keeps unexplored alternatives on the fringe (ensures completeness)
- Local search: improve a single option until you can't make it better (no fringe!)
- New successor function: local changes



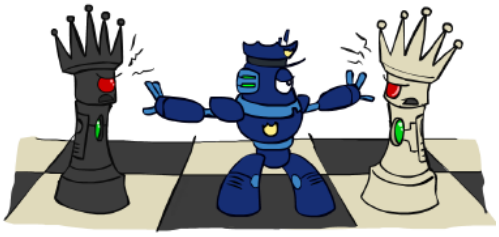
- Generally much faster and more memory efficient (but incomplete and suboptimal)

# Iterative Algorithms for CSPs

- 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
    - v.s., hill climb with h(x) = total number of violated constraints (break tie randomly)



## Hill Climbing

1. Start wherever
2. Repeat: move to the best neighboring state
3. Quit

# Hill Climbing (Greedy Local Search)

**function** HILL-CLIMBING(*problem*) **returns** a state that is a local maximum

  *current* ← MAKE-NODE(*problem*.INITIAL-STATE)
  **loop do**
    *neighbor* ← a highest-valued successor of *current*
    **if** neighbor.VALUE ≤ current.VALUE **then return** *current*.STATE
    *current* ← *neighbor*

Variants

- Random restart hill climbing

  “If at first you don’t succeed, try, try again."

- Stochastic hill climbing

  Choose randomly from the uphill moves , with probability dependent on the “steepness”

  Converge slower than steepest ascent, but may find better solutions

- First choice hill climbing

  Generate successors randomly (one by one) until a better one is found

  Suitable when there are too many successors to enumerate

### Simulated Annealing (*)

Combines random walk and hill climbing. Complete and efficient.

- Allow for downhill moves and make them rarer as time goes on
- Escape local maxima and reach global maxima

### Genetic Algorithms (*)

A variant of stochastic beam search

- Successors are generated by combining two parent states instead of modifying a single state

Content: State Representation & Fitness Function, Selection + Crossover + Mutation

For a specific problem, need to design these functions and operators

Successful use of genetic algorithms require careful engineering of the state representation!

### Summary

#### Local Search: Summary

- Maintain a constant number of current nodes or states, and move to “neighbors” or generate “offsprings” in each iteration
  - Do not maintain a search tree or multiple paths
  - Typically do not retain the path to the node

- Advantages
  - Use little memory
  - Can potentially solve large-scale problems or get a reasonable (suboptimal or almost feasible) solution

## Chapter 5: Adversarial Search

## Game Type

General Games

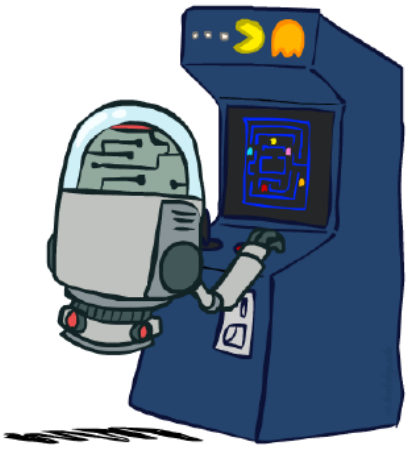
- Agents have independent utilities
- Cooperation, indifference, competition, shifting alliances, and more are all possible

Zero-Sum Games

- Agents have opposite utilities
- Adversarial, pure competition

### "Standard" Games

- Standard games are deterministic, observable, two-player, turn-taking, zero-sum
- Game formulation:
  - States: S (start at $s_0$)
  - Players: P={1...N} (usually take turns)
  - Actions: A (may depend on player / state)
  - Transition Function: SxA $\rightarrow$ S
  - Terminal Test: S $\rightarrow$ {t,f}
  - Terminal Utilities: SxP $\rightarrow$ R



- Solution for a player is a policy: S $\rightarrow$ A

17

How to model multi-agent game?

- Many single-agent trees

  Choose the best action for each agent independently
- Joint state/action spaces

  Think of one brain controlling many agents
- Centralized decision making

  Each agent proposes their actions and computer confirms the joint plan
- Alternate searching one agent at a time

## Minimax Search

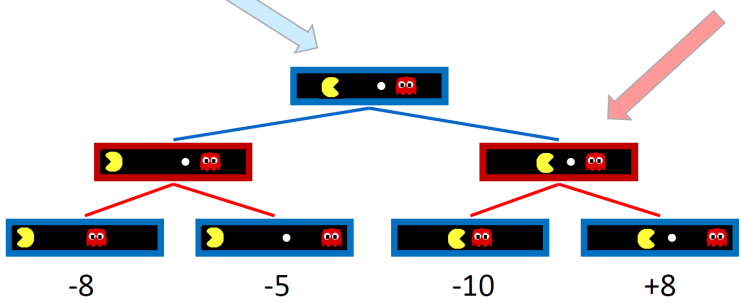
# Adversarial Game Trees: Minimax Values

States Under Agent's Control:

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

States Under Opponent's Control:

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$



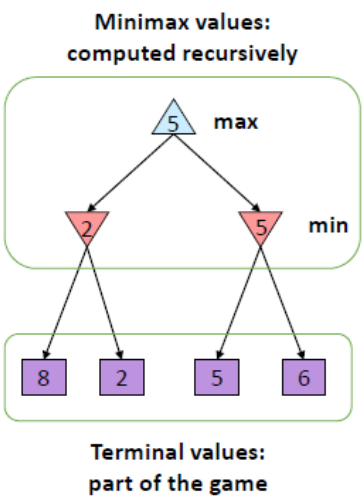

Terminal States:

$$V(s) = \text{known}$$



# Minimax Search

- Deterministic, zero-sum games:
  - Tic-tac-toe, chess, checkers
  - One player maximizes result
  - The other minimizes result
- Minimax search:
  - A state-space search tree
  - Players alternate turns
  - Compute each node's minimax value: the best achievable utility against a rational (optimal) adversary



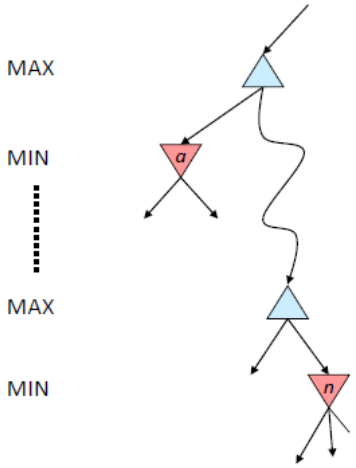

## Alpha-Beta Pruning

<u>The order of generation matters</u>: more pruning is possible if good moves come first

**Note**: Rational outcome is bounded in $[\alpha, \beta]$

# Game Tree Pruning: Alpha-Beta Pruning

- General configuration (MIN version)
  - We're computing the MIN-VALUE at some node *n*
  - We're looping over *n*'s children
  - *n*'s estimate of the childrens' min is dropping
  - Who cares about *n*'s value? MAX
  - Let *a* be the best value that MAX can get at any choice point along the current path from the root
  - If *n* becomes worse than *a*, MAX will avoid it, so we can stop considering *n*'s other children (it's already bad enough that it won't be played)




- MAX version is symmetric



# Alpha-Beta Implementation

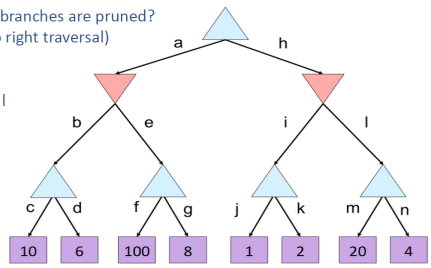
α: MAX's best option on path to root
β: MIN's best option on path to root

```
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
```

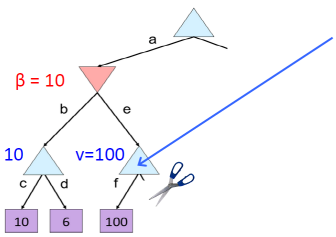
## Quiz 2

Which branches are pruned?
(Left to right traversal)
A) e, l
B) g, l
C) g, k, l
D) g, n






## Quiz 2 - 2

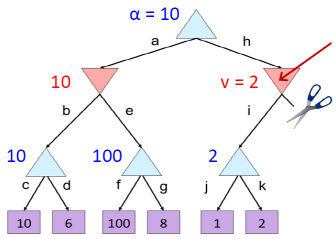



α: MAX's best option on path to root
β: MIN's best option on path to root

```
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
```



## Quiz 2 - 3

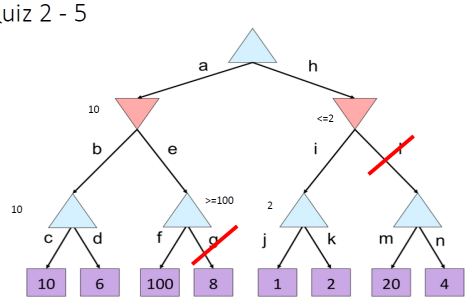



α: MAX's best option on path to root
β: MIN's best option on path to root

```
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
```

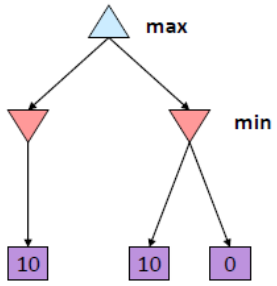


## Quiz 2 - 5



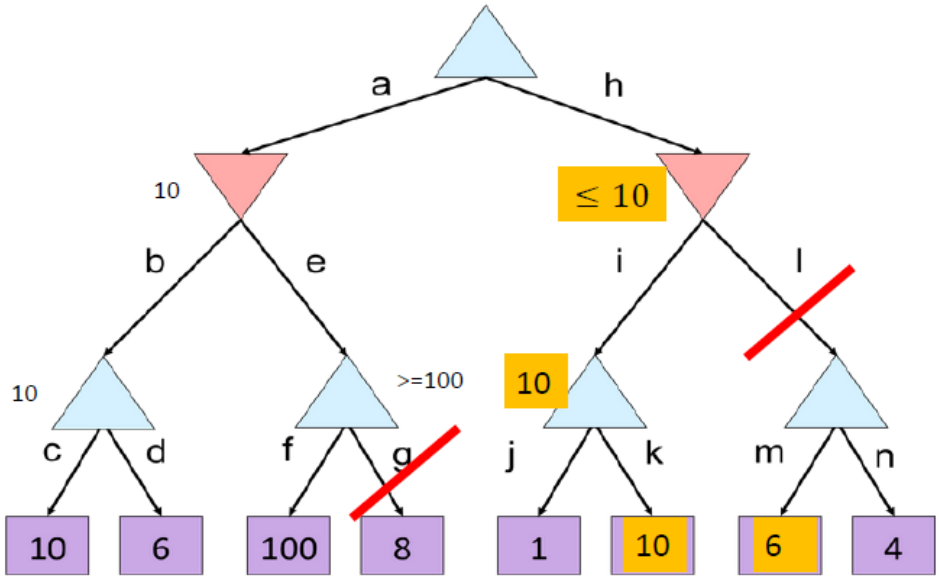
# Alpha-Beta Pruning Properties

- This pruning has no effect on minimax value computed for the root!
- Values of intermediate nodes might be wrong
  - Important: children of the root may have the wrong value
  - So the most naïve version won't let you do action selection
- Good child ordering improves effectiveness of pruning
- With "perfect ordering":
  - Time complexity drops to $O(b^{m/2})$
  - Doubles solvable depth!
  - Chess: 1M nodes/move => depth=8, respectable
  - Full search of complicated games, is still hopeless…
- This is a simple example of metareasoning (computing about what to compute)





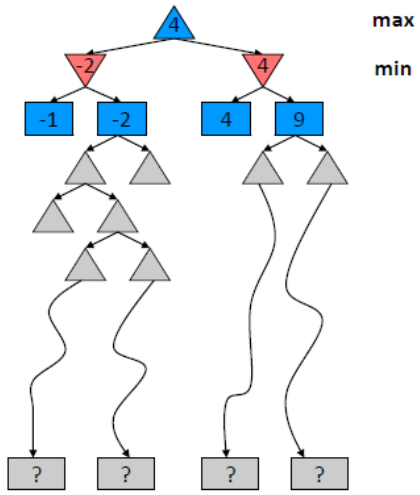

# Alpha-Beta Pruning: Action Selection Failure




## Bounded Lookahead

# Depth-limited search

- Problem: In realistic games, cannot search to leaves!
- Solution: Depth-limited search
  - Instead, search only to a limited depth in the tree
  - Replace terminal utilities with an evaluation function for non-terminal positions
- Example:
  - Suppose we have 100 seconds, can explore 10K nodes / sec
  - So can check 1M nodes per move
  - For chess, $b \approx 35$ so reaches about depth 4 – not so good
  - α-β reaches about depth 8 – decent chess program
- Guarantee of optimal play is gone
- More plies makes a BIG difference
- Use iterative deepening for an anytime algorithm



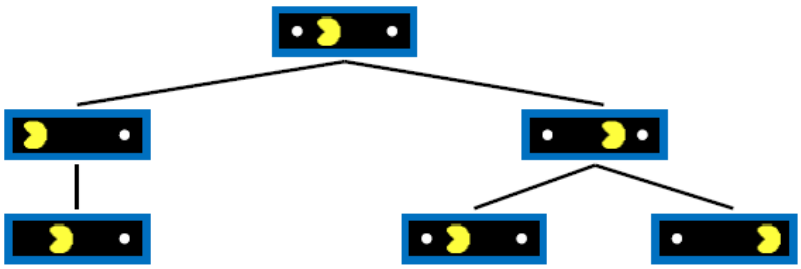

Depth matters:

- The deeper in the tree the evaluation function is buried, the less the quality of the evaluation function matters
- An important example of the tradeoff between complexity of features and complexity of computation

**Evaluation Functions**: score non-terminals

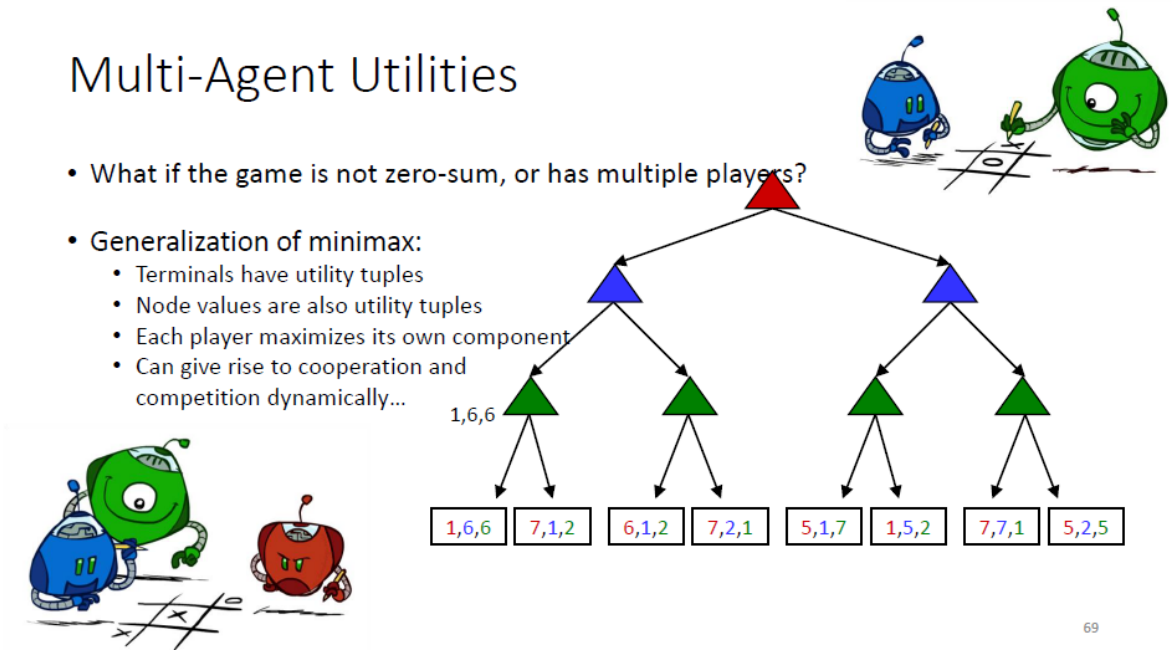
Ideal function: returns the actual minimax value of the position

# Why Pacman Starves



- A danger of replanning agents!
  - He knows his score will go up by eating the dot now (west, east)
  - He knows his score will go up just as much by eating the dot later (east, west)
  - There are no point-scoring opportunities after eating the dot (within the horizon, two here)
  - Therefore, waiting seems just as good as eating: he may go east, then back west in the next round of replanning!

## Generalized Minimax




## Modeling Assumptions

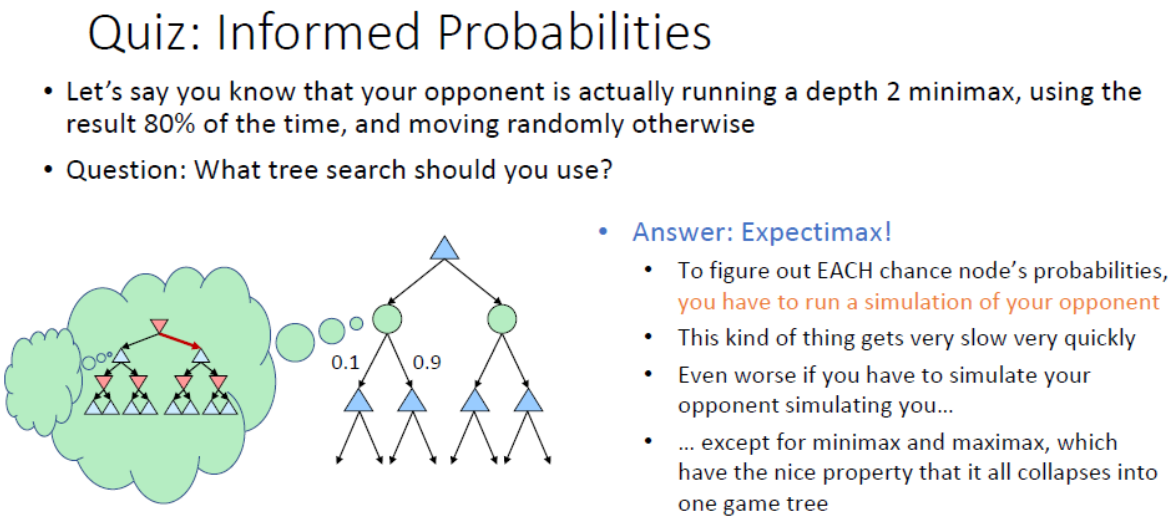
What if your opponent isn't playing optimally?

Q: Why not minimax?

A: Worst case reasoning is too conservative.

Idea: Add chance node and calculate the expected utilities

Note: Pruning is not allowed on chance node.




Pessimism: Assuming the worst case when it's not likely

Optimism: Assuming chance when the world is adversarial

## Utilities

# Utilities

- Utilities are functions from outcomes (states of the world) to real numbers that describe an agent's preferences

- Where do utilities come from?
  - In a game, may be simple (+1/-1)
  - Utilities summarize the agent's goals
  - Theorem: any "rational" preferences can be summarized as a utility function
- We hard-wire utilities and let behaviors emerge
  - Why don't we let agents pick utilities?
  - Why don't we prescribe behaviors?

99

A rational agent should chose the action that maximizes its expected utility, given its knowledge

Problems:

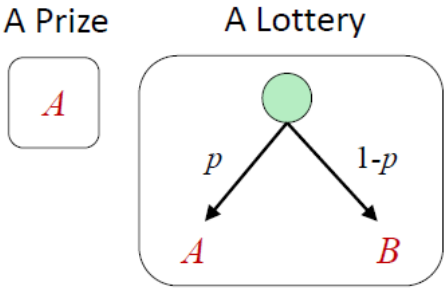
- Where do utilities come from?
- How do we know such utilities even exist?
- How do we know that averaging even makes sense?
- What if our behavior (preferences) can't be described by utilities?

For average case expectimax reasoning, we need magnitudes to be meaningful.

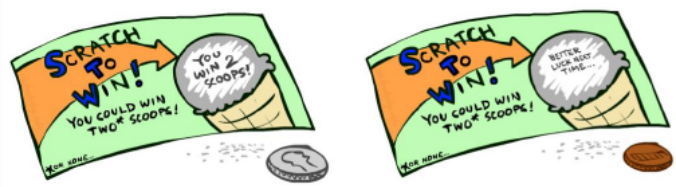
# Preferences

- An agent must have preferences among:
  - Prizes: $A$, $B$, etc.
  - Lotteries: situations with uncertain prizes

$$L = [p, A;\ (1-p), B]$$




- Notation:
  - Preference: $A \succ B$
  - Indifference: $A \sim B$

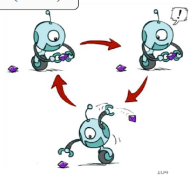



## Rational Preferences

- We want some constraints on preferences before we call them rational, such as:

Axiom of Transitivity: $(A \succ B) \wedge (B \succ C) \Rightarrow (A \succ C)$

- For example: an agent with intransitive preferences can be induced to give away all of its money
  - If B > C, then an agent with C would pay (say) 1 cent to get B
  - If A > B, then an agent with B would pay (say) 1 cent to get A
  - If C > A, then an agent with A would pay (say) 1 cent to get C



## Rational Preferences 2

- The Axioms of Rationality

Orderability
$(A \succ B) \vee (B \succ A) \vee (A \sim B)$
Transitivity
$(A \succ B) \wedge (B \succ C) \Rightarrow (A \succ C)$
Continuity
$A \succ B \succ C \Rightarrow \exists p\ [p, A;\ 1-p, C] \sim B$
Substitutability
$A \sim B \Rightarrow [p, A;\ 1-p, C] \sim [p, B; 1-p, C]$
Monotonicity
$A \succ B \Rightarrow$
$(p \geq q \Leftrightarrow [p, A;\ 1-p, B] \succeq [q, A;\ 1-q, B])$

- Theorem: Rational preferences imply behavior describable as maximization of expected utility

105

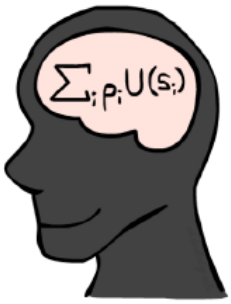
# MEU Principle

- Theorem [Ramsey, 1931; von Neumann & Morgenstern, 1944]
  - Given any preferences satisfying these constraints, there exists a real-valued function U such that:

$$U(A) \geq U(B) \;\Leftrightarrow\; A \succeq B$$

$$U([p_1, S_1;\; \ldots\; ;\; p_n, S_n]) = \sum_i p_i U(S_i)$$

  - i.e. values assigned by U preserve preferences of both prizes and lotteries!

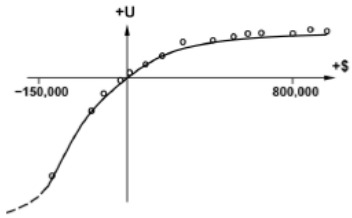



- Maximum expected utility (MEU) principle:
  - Choose the action that maximizes expected utility
  - Note: an agent can be entirely rational (consistent with MEU) without ever representing or manipulating utilities and probabilities
  - E.g., a lookup table for perfect tic-tac-toe, a reflex vacuum cleaner

### Human Utilities (*)

# Human Utilities: Money

- Money does not behave as a utility function, but we can talk about the utility of having money (or being in debt)
- Given a lottery L = [p, $X; (1-p), $Y]
  - The expected monetary value EMV(L) is p*X + (1-p)*Y
  - U(L) = p*U($X) + (1-p)*U($Y)
  - Typically, U(L) < U( EMV(L) )
  - In this sense, people are risk-averse
  - When deep in debt, people are risk-prone
    - When $X, Y < 0$, U(L) > U( EMV(L) )

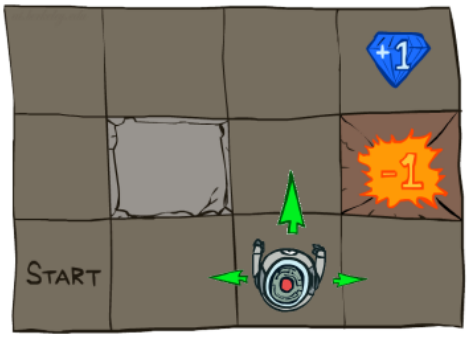





# Chapter 6: Markov Decision Process (MDP)

## Markov Decision Process

# Markov Decision Processes




- An MDP is defined by:
  - A set of states $s \in S$
  - A set of actions $a \in A$
  - A transition function $T(s, a, s')$
    - Probability that a from s leads to s', i.e., $P(s' | s, a)$
    - Also called the model or the dynamics
  - A reward function $R(s, a, s')$
    - Sometimes just $R(s)$ or $R(s')$
  - A start state
  - Maybe a terminal state

- MDPs are non-deterministic search problems
  - One way to solve them is with expectimax search

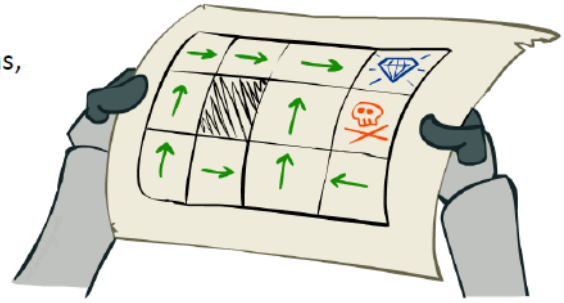
"Markov" generally means that given the present state, the future and the past are independent.

For Markov decision processes, "Markov" means action outcomes depend only on the current state.

# Policies

- In deterministic single-agent search problems, we wanted an optimal plan, or sequence of actions, from start to a goal

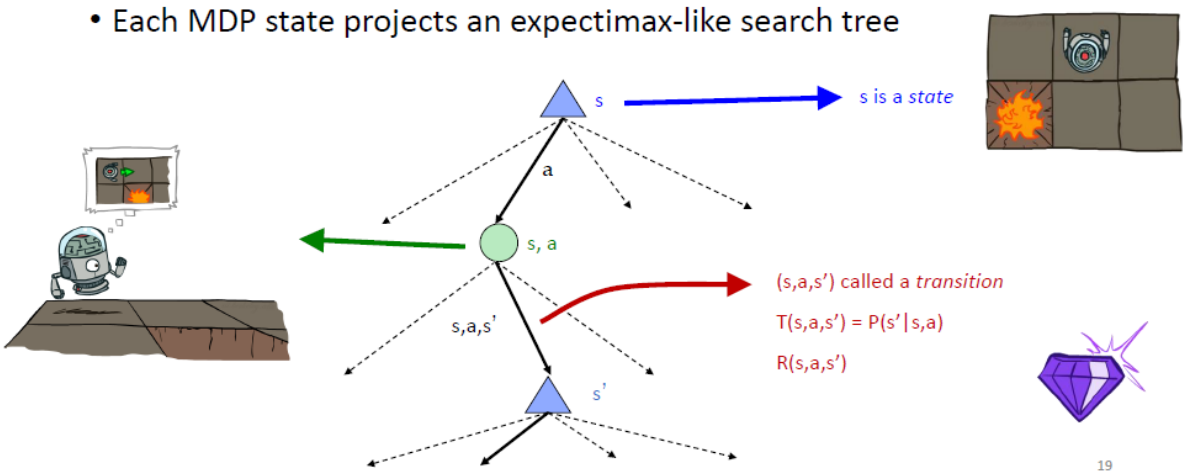
- For MDPs, we want an optimal policy $\pi^*: S \rightarrow A$
  - A policy $\pi$ gives an action for each state
  - An optimal policy is one that maximizes expected utility if followed
  - An explicit policy defines a reflex agent



Optimal policy when $R(s, a, s') = -0.03$ for all non-terminals s

# MDP Search Trees

- Each MDP state projects an expectimax-like search tree






Utilities of Sequences: **Discounting**

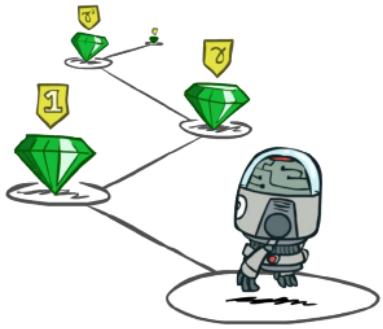
- It's reasonable to maximize the sum of rewards
- It's also reasonable to prefer rewards now to rewards later

- Also helps our algorithms converge

## Utilities of Sequences: Stationary Preferences

- Theorem: if we assume stationary preferences:

$$[a_1, a_2, \ldots] \succ [b_1, b_2, \ldots]$$
$$\Updownarrow$$
$$[r, a_1, a_2, \ldots] \succ [r, b_1, b_2, \ldots]$$



- Then: there are only two ways to define utilities
  - Additive utility: $U([r_0, r_1, r_2, \ldots]) = r_0 + r_1 + r_2 + \cdots$
  - Discounted utility: $U([r_0, r_1, r_2, \ldots]) = r_0 + \gamma r_1 + \gamma^2 r_2 \cdots$

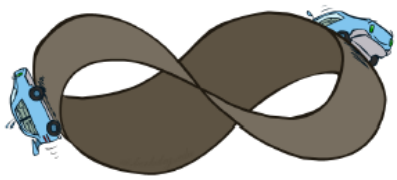
Note: $U_\gamma + U_{\gamma'}$ is not correct

Example:

$$(U_{0.9} + U_{0.5})\left(\frac{3}{4}, 0, 0, \ldots\right) > (U_{0.9} + U_{0.5})(0, 1, 0, \ldots)$$
$$(U_{0.9} + U_{0.5})\left(r, \frac{3}{4}, 0, \ldots\right) < (U_{0.9} + U_{0.5})(r, 0, 1, 0, 0, \ldots)$$

## Infinite Utilities?!

- Problem: What if the game lasts forever? Do we get infinite rewards?
- Solutions:
  - Finite horizon: (similar to depth-limited search)
    - Terminate episodes after a fixed T steps (e.g. life)
    - Gives nonstationary policies (π depends on time left)



  - Discounting: use 0 < γ < 1

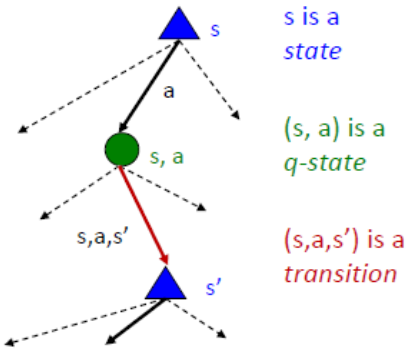
$$U([r_0, \ldots r_\infty]) = \sum_{t=0}^{\infty} \gamma^t r_t \leq R_{\max}/(1-\gamma)$$

    - Smaller γ means smaller "horizon" – shorter term focus
  - Absorbing state: guarantee that for every policy, a terminal state will eventually be reached (like "overheated" for racing)

## Solving MDPs

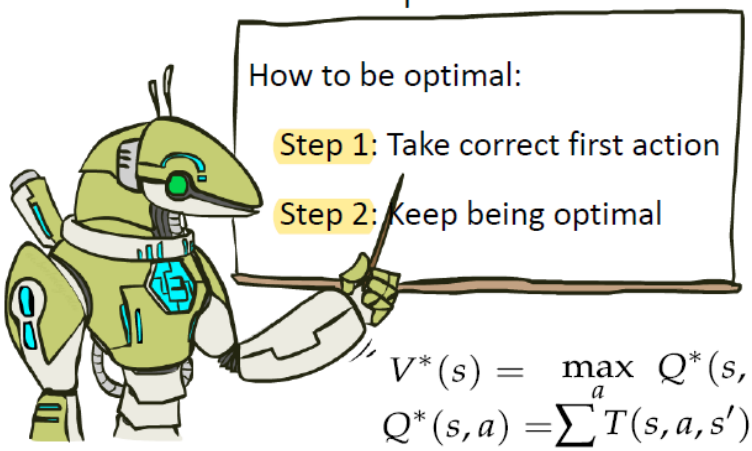
# Optimal Quantities

- The value (utility) of a state s:
  - V*(s) = expected utility starting in s and acting optimally
- The value (utility) of a q-state (s,a):
  - Q*(s,a) = expected utility starting out having taken action a from state s and (thereafter) acting optimally




- The optimal policy:
  - π*(s) = optimal action from state s

## The Bellman Equations




$$V^*(s) = \max_a Q^*(s,a)$$

$$Q^*(s,a) = \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$

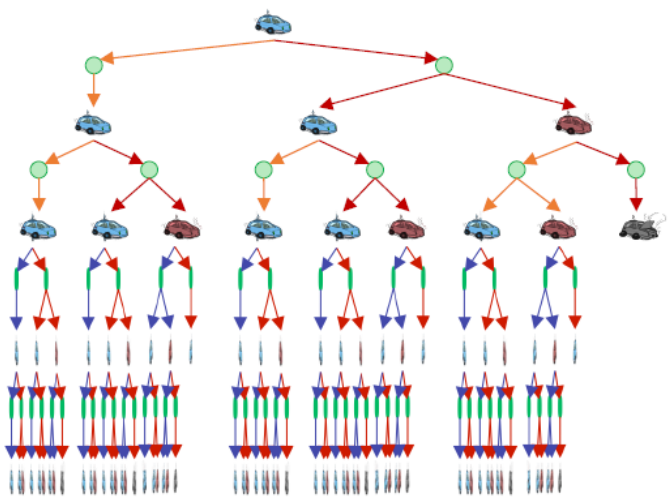
$$V^*(s) = \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$



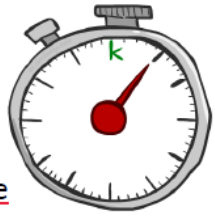
**Expectimax**

# Racing Search Tree 3

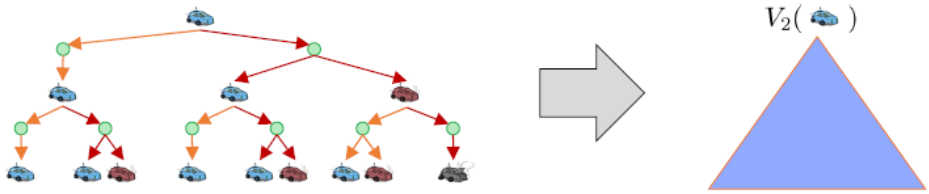
- We're doing way too much work with expectimax!
- Problem: States are repeated
  - Idea: Only compute needed quantities once
- Problem: Tree goes on forever
  - Idea: Do a depth-limited computation, but with increasing depths until change is small
  - Note: deep parts of the tree eventually don't matter if γ < 1



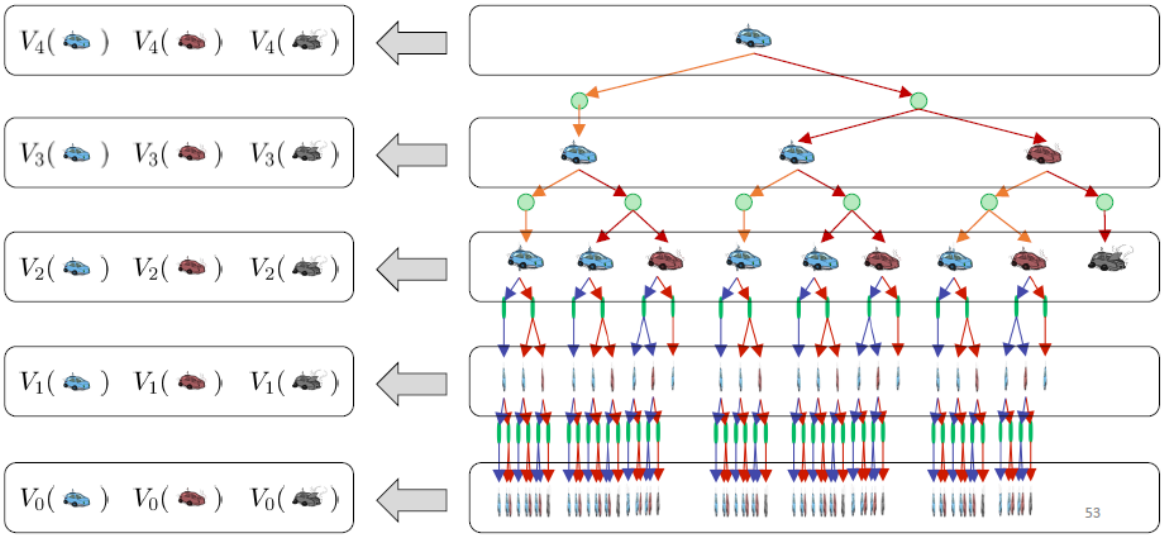
# Time-Limited Values




- Key idea: time-limited values
- Define $V_k(s)$ to be the optimal value of s if the game ends in k more time steps
  - Equivalently, it's what a depth-k expectimax would give from s




# Time-Limited Values: Computing



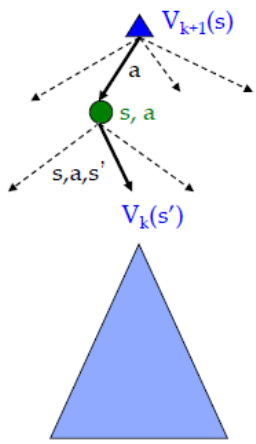

**Value Iteration**

# Value Iteration

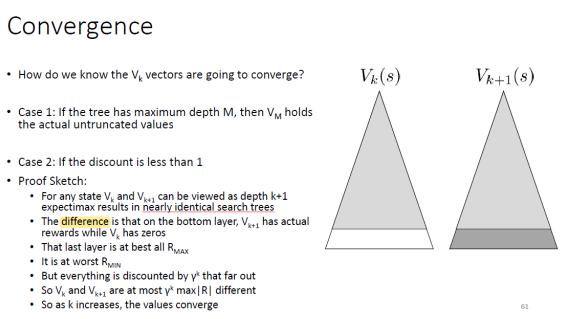
- Start with $V_0(s) = 0$: no time steps left means an expected reward sum of zero
- Given vector of $V_k(s)$ values, do one ply of expectimax from each state:

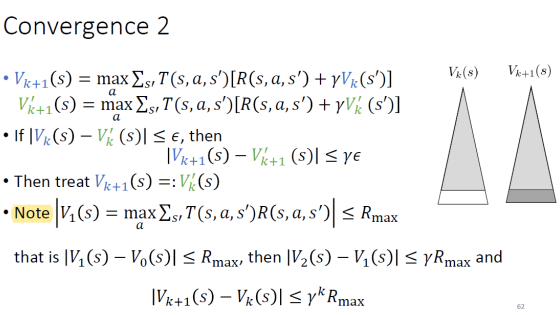
$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_k(s')\right]$$

- Repeat until convergence, which yields V*




- Complexity of each iteration: $O(S^2A)$
- Theorem: will converge to unique optimal values
  - Basic idea: approximations get refined towards optimal values
  - Policy may converge long before values do




Supplement:

Assume $V_{k+1}(s) \geq V'_{k+1}(s)$. Then, we have

$$V_{k+1}(s) = Q_{k+1}(s,a) \geq V'_{k+1}(s) = Q'_{k+1}(s,a') \geq Q'_{k+1}(s,a)$$
$$Q_{k+1}(s,a) \geq Q'_{k+1}(s,a') \geq Q'_{k+1}(s,a) \geq Q_{k+1}(s,a) - \gamma\epsilon$$

Value iteration is just a <u>fixed point solution method</u>

**Policy Extraction**: Computing Actions from Values

$$\pi^*(s) = \arg\max_a Q^*(s,a) = \arg\max_a \sum_{s'} T\left(s,a,s'\right)\left[R\left(s,a,s'\right) + \gamma V^*\left(s'\right)\right]$$

<u>Important lesson</u>: actions are easier to select from q values than v values!

## Policy Iteration

Problems with VI

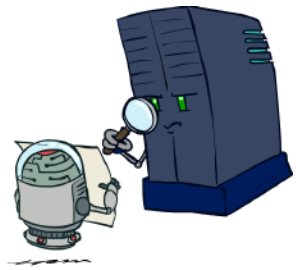
- Slow. $O(S^2A)$
- The "max" at each state rarely changes
- The policy often converges long before the values

# Policy Iteration

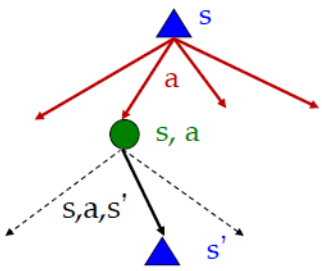
- Alternative approach for optimal values:
  - Step 1: Policy Evaluation: calculate utilities for some fixed policy (not optimal utilities!) until convergence
  - Step 2: Policy Improvement: update policy using one-step look-ahead with resulting converged (but not optimal!) utilities as future values
  - Repeat steps until policy converges

- This is Policy Iteration
  - It's still optimal!
  - Can converge (much) faster under some conditions

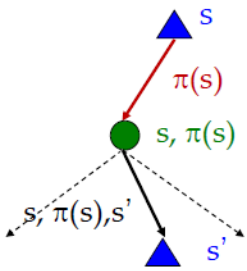
# Policy Evaluation: Fixed Policies



Do the optimal action




Do what π says to do




- Expectimax trees max over all actions to compute the optimal values
- If we fix some policy π(s), then the tree would be simpler – only one action per state
  - … though the tree's value would depend on which policy we fixed

73

Utilities for a Fixed Policy

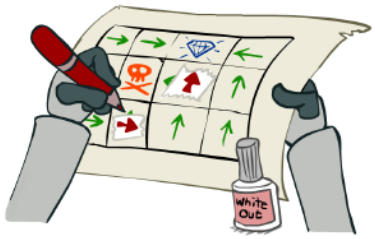
$V^{\pi}(s)$ = expected total discounted rewards starting in s and following $\pi$

$$V^{\pi}(s) = \sum_{s'} T\left(s, \pi(s), s'\right)\left[R\left(s, \pi(s), s'\right) + \gamma V^{\pi}\left(s'\right)\right]$$

Implementation:

1. Turn recursive Bellman equations into updates. (like value iteration, $O(S^2)$ per iteration)
2. Solve with linear system solver.

# Policy Iteration




- Evaluation: For fixed current policy $\pi$, find values with policy evaluation:
  - Iterate until values converge:

$$V_{k+1}^{\pi_i}(s) \leftarrow \sum_{s'} T(s, \pi_i(s), s')\left[R(s, \pi_i(s), s') + \gamma V_k^{\pi_i}(s')\right]$$

- Improvement: For fixed values, get a better (why? exercise) policy using policy extraction
  - One-step look-ahead:

$$\pi_{i+1}(s) = \arg\max_a \sum_{s'} T(s, a, s')\left[R(s, a, s') + \gamma V^{\pi_i}(s')\right]$$

### Policy Improvement

- Consider a deterministic policy, $a = \pi(s)$
- We can *improve* the policy by acting greedily

$$\pi'(s) = \underset{a \in \mathcal{A}}{\operatorname{argmax}}\, q_\pi(s, a)$$

- This improves the value from any state $s$ over one step,

$$q_\pi(s, \pi'(s)) = \max_{a \in \mathcal{A}} q_\pi(s, a) \geq q_\pi(s, \pi(s)) = v_\pi(s)$$

- It therefore improves the value function, $v_{\pi'}(s) \geq v_\pi(s)$

$$\begin{aligned} v_\pi(s) &\leq q_\pi(s, \pi'(s)) = \mathbb{E}_{\pi'}\left[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s\right] \\ &\leq \mathbb{E}_{\pi'}\left[R_{t+1} + \gamma q_\pi(S_{t+1}, \pi'(S_{t+1})) \mid S_t = s\right] \\ &\leq \mathbb{E}_{\pi'}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 q_\pi(S_{t+2}, \pi'(S_{t+2})) \mid S_t = s\right] \\ &\leq \mathbb{E}_{\pi'}\left[R_{t+1} + \gamma R_{t+2} + \ldots \mid S_t = s\right] = v_{\pi'}(s) \end{aligned}$$

### Policy Improvement (2)

- If improvements stop,

$$q_\pi(s, \pi'(s)) = \max_{a \in \mathcal{A}} q_\pi(s, a) = q_\pi(s, \pi(s)) = v_\pi(s)$$

- Then the Bellman optimality equation has been satisfied

$$v_\pi(s) = \max_{a \in \mathcal{A}} q_\pi(s, a)$$

- Therefore $v_\pi(s) = v_*(s)$ for all $s \in \mathcal{S}$
- so $\pi$ is an optimal policy

## Summary

# Summary of Two Methods for Solving MDPs

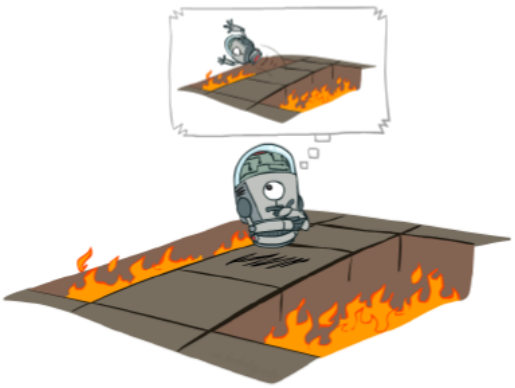
- Value iteration + policy extraction
  - Step 1: Value iteration: calculate values for all states by running one ply of the Bellman equations using values from previous iteration **until convergence**
  - Step 2: Policy extraction: compute policy by running one ply of the Bellman equations using values from value iteration
- Policy iteration
  - Step 1: Policy evaluation: calculate values for some fixed policy (not optimal values!) **until convergence**
  - Step 2: Policy improvement: update policy by running one ply of the Bellman equations using values from policy evaluation
  - Repeat steps until policy converges

# Value Iteration vs. Policy Iteration

- Both value iteration and policy iteration compute the same thing (all optimal values)
- In value iteration:
  - Every iteration updates both the values and (implicitly) the policy
  - We don't track the policy, but taking the max over actions implicitly recomputes it
- In policy iteration:
  - We do several passes that update utilities with fixed policy (each pass is fast because we consider only one action, not all of them)
  - After the policy is evaluated, a new policy is chosen (slow like a value iteration pass)
  - The new policy will be better (or we're done)
- Both are dynamic programs for solving MDPs

# Chapter 7: Reinforcement Learning

# Offline (MDPs) vs. Online (RL)



Offline Solution

Online Learning

# What Just Happened?

- That wasn't planning, it was learning!
  - Specifically, reinforcement learning
  - There was an MDP, but you couldn't solve it with just computation
  - You needed to actually act to figure it out

- Important ideas in reinforcement learning that came up
  - Exploration: you have to try unknown actions to get information
  - Exploitation: eventually, you have to use what you know
  - Regret: even if you learn intelligently, you make mistakes
  - Sampling: because of chance, you have to try things repeatedly
  - Difficulty: learning can be much harder than solving a known MDP

# Reinforcement Learning

- What if we didn't know $P(s'|s,a)$ and $R(s,a,s')$?

Value iteration: $$V_{k+1}(s) = \max_a \sum_{s'} \cancel{P(s'|s,a)}[\cancel{R(s,a,s')} + \gamma V_k(s')], \quad \forall s$$

Q-iteration: $$Q_{k+1}(s,a) = \sum_{s'} \cancel{P(s'|s,a)}[\cancel{R(s,a,s')} + \gamma \max_{a'} Q_k(s',a')], \quad \forall s,a$$

Policy extraction: $$\pi_V(s) = \operatorname*{argmax}_a \sum_{s'} \cancel{P(s'|s,a)}[\cancel{R(s,a,s')} + \gamma V(s')], \quad \forall s$$

Policy evaluation: $$V^\pi_{k+1}(s) = \sum_{s'} \cancel{P(s'|s,\pi(s))}[\cancel{R(s,\pi(s),s')} + \gamma V^\pi_k(s')], \quad \forall s$$

Policy improvement: $$\pi_{new}(s) = \operatorname*{argmax}_a \sum_{s'} \cancel{P(s'|s,a)}[\cancel{R(s,a,s')} + \gamma V^{\pi_{old}}(s')], \quad \forall s$$

Basic idea:

- Receive feedback in the form of <u>rewards</u>
- Must (learn to) act so as to maximize expected rewards
- All learning is based on <u>observed</u> samples of outcomes!

# Reinforcement Learning -- Overview

- Passive Reinforcement Learning (= how to learn from experiences)
  - Model-based Passive RL
    - Learn the MDP model from experiences, then solve the MDP
  - Model-free Passive RL
    - Forego learning the MDP model, directly learn V or Q:
      - Value learning – learns value of a fixed policy; 2 approaches: Direct Evaluation & TD Learning
      - Q learning – learns Q values of the optimal policy (uses a Q version of TD Learning)
- Active Reinforcement Learning (= agent also needs to decide how to collect experiences)
  - Key challenges:
    - How to efficiently explore?
    - How to trade off exploration <> exploitation
  - Applies to both model-based and model-free.
    we'll cover only in context of Q-learning

20

## Passive Reinforcement Learning

Key: how to learn from experiences

### Model-based Passive RL

# Model-Based Reinforcement Learning

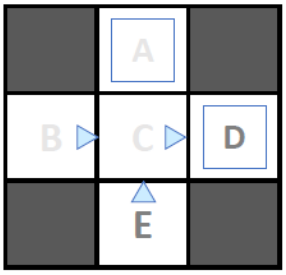
- Model-Based Idea:
  - Learn an approximate model based on experiences
  - Solve for values as if the learned model were correct

- Step 1: Learn empirical MDP model
  - Count outcomes s' for each s, a
  - Normalize to give an estimate of $\hat{T}(s, a, s')$
  - Discover each $\hat{R}(s, a, s')$ when we experience (s, a, s')

- Step 2: Solve the learned MDP
  - For example, use value iteration, as before

# Example: Model-Based RL

**Input Policy π**




*Assume: γ = 1*

**Observed Episodes (Training)**

Episode 1

B, east, C, -1
C, east, D, -1
D, exit, x, +10

Episode 2

B, east, C, -1
C, east, D, -1
D, exit, x, +10

Episode 3

E, north, C, -1
C, east, D, -1
D, exit, x, +10

Episode 4

E, north, C, -1
C, east, A, -1
A, exit, x, -10

**Learned Model**

$\hat{T}(s, a, s')$

T(B, east, C) = 1.00
T(C, east, D) = 0.75
T(C, east, A) = 0.25
...

$\hat{R}(s, a, s')$

R(B, east, C) = -1
R(C, east, D) = -1
R(D, exit, x) = +10
...

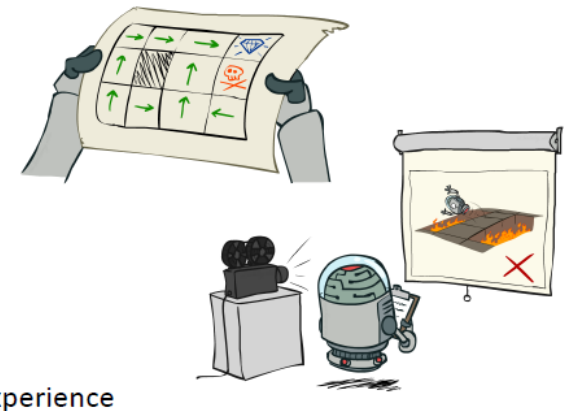
## Model-free Passive RL

Forego learning the MDP model, directly learn V or Q

### Value Learning

## Passive Model-Free Reinforcement Learning

- Simplified task: policy evaluation
  - Input: a fixed policy π(s)
  - You don't know the transitions T(s,a,s')
  - You don't know the rewards R(s,a,s')
  - Goal: learn the state values



- In this case:
  - Learner is "along for the ride"
  - No choice about what actions to take
  - Just execute the policy and learn from experience
  - This is NOT offline planning! You actually take actions in the world

### Direct Evaluation (MC Method)

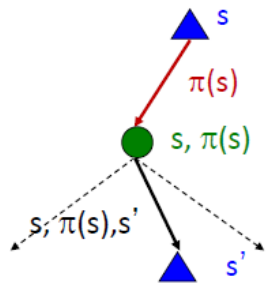
- Act according to $\pi$
- Every time you visit a state, write down what the sum of discounted rewards turned out to be
- Average those samples

Problems:

- It wastes information about state connections
- Each state must be learned separately

### TD-Learning

# Why Not Use Policy Evaluation?




- Simplified Bellman updates calculate V for a fixed policy:
  - Each round, replace V with a one-step-look-ahead layer over V

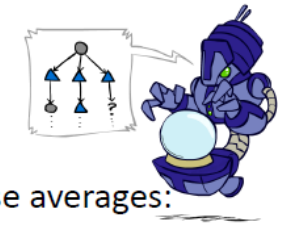
$$V_0^\pi(s) = 0$$

$$V_{k+1}^\pi(s) \leftarrow \sum_{s'} T(s,\pi(s),s')[R(s,\pi(s),s') + \gamma V_k^\pi(s')]$$

  - This approach fully exploited the connections between the states
  - Unfortunately, we need T and R to do it!

- Key question: how can we do this update to V without knowing T and R?
  - In other words, how do we take a weighted average without knowing the weights?

# Sample-Based Policy Evaluation?



- We want to improve our estimate of V by computing these averages:

$$V_{k+1}^\pi(s) \leftarrow \sum_{s'} T(s,\pi(s),s')[R(s,\pi(s),s') + \gamma V_k^\pi(s')]$$

- Idea: Take samples of outcomes s' (by doing the action!) and average

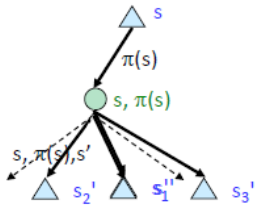
$$sample_1 = R(s,\pi(s),s_1') + \gamma V_k^\pi(s_1')$$

$$sample_2 = R(s,\pi(s),s_2') + \gamma V_k^\pi(s_2')$$

$$\ldots$$

$$sample_n = R(s,\pi(s),s_n') + \gamma V_k^\pi(s_n')$$

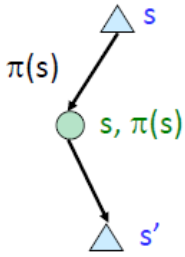
$$V_{k+1}^\pi(s) \leftarrow \frac{1}{n}\sum_i sample_i$$




*Almost! But we can't rewind time to get sample after sample from state s*



# Temporal Difference Value Learning




- Big idea: learn from every experience!
  - Update V(s) each time we experience a transition (s, a, s', r)
  - Likely outcomes s' will contribute updates more often

- Temporal difference learning of values
  - Policy still fixed, still doing evaluation!
  - Move values toward value of whatever successor occurs: running average

Sample of V(s): $sample = R(s,\pi(s),s') + \gamma V^\pi(s')$

Update to V(s): $V^\pi(s) \leftarrow (1-\alpha)V^\pi(s) + (\alpha)sample$

Same update: $V^\pi(s) \leftarrow V^\pi(s) + \alpha(sample - V^\pi(s))$

## Gradient Descent View

$f(x) = \frac{1}{2}(y-x)^2$

$\frac{df}{dx} = -(y-x)$

- Goal: find $x$ that minimizes $f(x)$

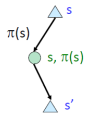
1. Start with initial guess, $x_0$
2. Update $x$ by taking a step in the direction that $f(x)$ is changing fastest (in the negative direction) with respect to x:
   $x \leftarrow x - \alpha \nabla_x f$, where $\alpha$ is the step size or learning rate
3. Repeat until convergence

- TD goal: find value(s), V, that minimizes difference between sample(s) and V

$V \leftarrow V - \alpha \nabla_V Error$ $\qquad Error(V) = \frac{1}{2}(sample - V)^2$

## Gradient Descent View 2

- Big idea: learn from every experience!
  - Update V(s) each time we experience a transition (s, a, s', r)
  - Likely outcomes s' will contribute updates more often
- Temporal difference learning of values
  - Policy still fixed, still doing evaluation!
  - Move values toward value of whatever successor occurs: running average




Sample of V(s): $sample = r + \gamma V^\pi(s')$

Update to V(s): $V^\pi(s) \leftarrow (1-\alpha) V^\pi(s) + (\alpha)\, sample$

Same update: $V^\pi(s) \leftarrow V^\pi(s) + \alpha\,[sample - V^\pi(s)]$

Same update: $V^\pi(s) \leftarrow V^\pi(s) - \alpha \nabla Error$ $\qquad Error = \frac{1}{2}(sample - V^\pi(s))^2$

# Exponential Moving Average

- Exponential moving average
  - The running interpolation update: $V_n = (1-\alpha)V_{n-1} + \alpha x_n$ with $V_1 = x_1$
  - Makes recent samples more important
    $$V_n = \alpha x_n + \alpha(1-\alpha)x_{n-1} + \cdots + \alpha(1-\alpha)^{n-2}x_2 + (1-\alpha)^{n-1}x_1$$
  - Forgets about the past (distant past values were wrong anyway)
- Decreasing learning rate (alpha) can give converging averages
  - Note $V_n = \alpha_n x_n + (1-\alpha_n)\alpha_{n-1}x_{n-1} + \cdots$
    $+(1-\alpha_n)(1-\alpha_{n-1}) \cdot \cdots \cdot (1-\alpha_3)\alpha_2 x_2$
    $+(1-\alpha_n)(1-\alpha_{n-1}) \cdot \cdots \cdot (1-\alpha_3)(1-\alpha_2)x_1$

Problems: CAN NOT turn values into a policy.

Idea: learn Q values, not values

Makes action selection model free too!

**Q Learning**

Q-Value Iteration:

- Start with $Q(s,a) = 0$, which we know is right
- Given $Q_k$, calculate the depth $k+1$ q-values for all q-states:

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma \max_{a'} Q_k(s',a')\right]$$

Q-Learning:

- Receive a sample (s, a, s', r)
- Consider your new sample estimate: (no longer policy evaluation)

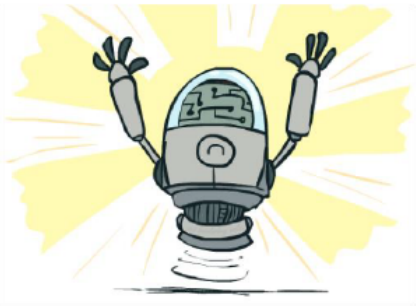
$$\text{sample} = R(s,a,s') + \gamma \max_{a'} Q(s',a')$$

- Incorporate the new estimate into a running average:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + (\alpha)[\text{ sample }]$$

# Q-Learning Properties

- Amazing result: Q-learning converges to optimal policy -- even if you're acting suboptimally!
- This is called off-policy learning
- Caveats:
  - You have to explore enough
  - You have to eventually make the learning rate small enough
  - ... but not decrease it too quickly
  - Basically, in the limit, it doesn't matter how you select actions (!)

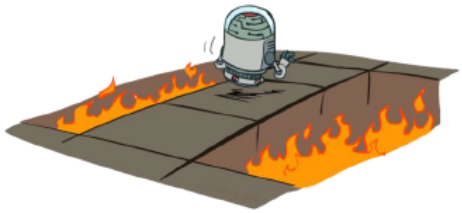


## Active Reinforcement Learning

Key: agent needs to decide how to collect experiences

# Active Reinforcement Learning

- Full reinforcement learning: optimal policies (like value iteration)
  - You don't know the transitions T(s,a,s')
  - You don't know the rewards R(s,a,s')
  - You choose the actions now
  - Goal: learn the optimal policy / values



- In this case:
  - Learner makes choices!
  - Fundamental tradeoff: exploration vs. exploitation
  - This is NOT offline planning! You actually take actions in the world and find out what happens...

### How to Explore

# How to Explore?

- Several schemes for forcing exploration
  - Simplest: random actions (ε-greedy)
    - Every time step, flip a coin
    - With (small) probability ε, act randomly
    - With (large) probability 1-ε, act on current policy
  - Problems with random actions?
    - You do eventually explore the space, but keep thrashing around once learning is done
    - One solution: lower ε over time
    - Another solution: exploration functions

# Exploration Functions

A commonly used 'exploration function' is $f(u,n) = u + c\sqrt{\log(1/\delta)/n}$, which is derived by Chernoff-Hoeffding inequality and $\delta$ is confidence level

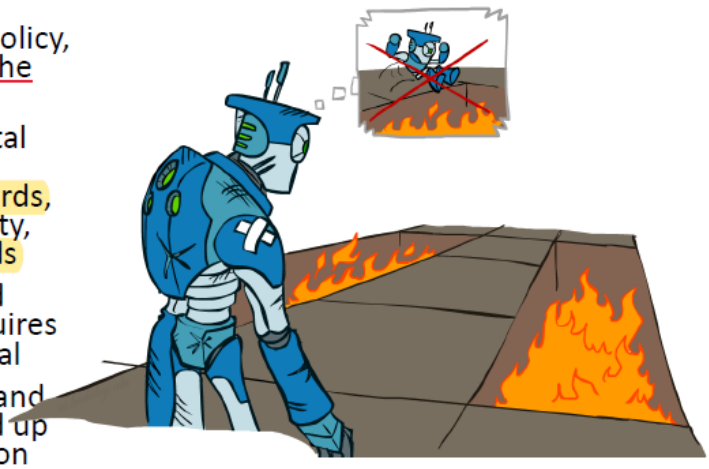
- When to explore?
  - Random actions: explore a fixed amount
  - Better idea: explore areas whose badness is not (yet) established, eventually stop exploring
- Exploration function
  - Takes a value estimate u and a visit count n, and returns an optimistic utility, e.g. $f(u,n) = u + k/n$

    Regular Q-Update: $Q(s,a) \leftarrow_\alpha R(s,a,s') + \gamma \max_{a'} Q(s',a')$
  - Modified Q-Update: $Q(s,a) \leftarrow_\alpha R(s,a,s') + \gamma \max_{a'} f(Q(s',a'), N(s',a'))$
  - Action selection: Use $a \leftarrow \text{argmax}_a Q(s,a)$
  - Note: this propagates the "bonus" back to states that lead to unknown states as well!

**Regret (*)**

# Regret

- Even if you learn the optimal policy, you still make mistakes along the way!
- Regret is a measure of your total mistake cost: the difference between your (expected) rewards, including youthful suboptimality, and optimal (expected) rewards
- Minimizing regret goes beyond learning to be optimal – it requires optimally learning to be optimal
- Example: random exploration and exploration functions both end up optimal, but random exploration has higher regret



# Regret 2

- Cumulative regret, i.e., for episodic MDP with fixed horizon

$$R(T) = \sum_{t=1}^{T} (V^*(s_t) - V^{\pi_t}(s_t))$$

  where $s_t$ is the starting state of the $t$-th interaction game
- The algorithm is learning if the average regret converges, i.e. $\frac{R(T)}{T} \to 0$, or equivalently $R(T) = o(T)$
- Smaller order of $R(T)$ means faster learning speed
- Worst-case regret bound $R(T) = \Omega(\sqrt{T})$, which holds for a fixed game with arbitrary transitions and arbitrary (bounded) rewards

# Approximate Reinforcement Learning (*)

Key: handle large state spaces

## Approximate Q Learning

Basic Q Learning keeps a table of all q values

- Too many states to visit them all in training
- Too many states to hold the q tables in memory

Instead, we want to generalize:

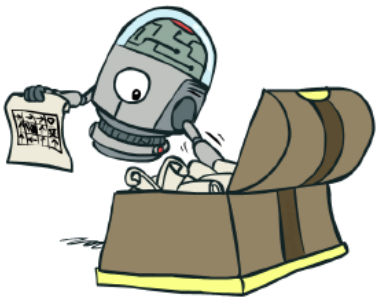
- Learn about some small number of training states from experience
- Generalize that experience to new, similar situations

Idea: Feature Based Representations + Parameterized Value Functions

Note: Limiting capacity to avoid overfitting.

## Policy Search

### Policy Search



- Problem: often the feature-based policies that work well (win games, maximize utilities) aren't the ones that approximate V / Q best
  - E.g. some value functions have probably horrible estimates of future rewards, but they still produced good decisions
  - Q-learning's priority: get Q-values close (modeling)
  - Action selection priority: get ordering of Q-values right (prediction)
  - We'll see this distinction between modeling and prediction again later in the course
- Solution: learn policies that maximize rewards, not the values that predict them
- Policy search: start with an ok solution (e.g. Q-learning) then fine-tune by hill climbing on feature weights

### Policy Search 2

- Simplest policy search:
  - Start with an initial linear value function or Q-function
  - Nudge each feature weight up and down and see if your policy is better than before
- Problems:
  - How do we tell the policy got better?
  - Need to run many sample episodes!
  - If there are a lot of features, this can be impractical
- Better methods exploit lookahead structure, sample wisely, change multiple parameters…

## Summary

### MDPs and RL

**Known MDP: Offline Solution**

| Goal | Technique |
|---|---|
| Compute V*, Q*, π* | Value / policy iteration |
| Evaluate a fixed policy π | Policy evaluation |

**Unknown MDP: Model-Based**

| Goal *use features to generalize | Technique |
|---|---|
| Compute V*, Q*, π* | VI/PI on approx. MDP |
| Evaluate a fixed policy π | PE on approx. MDP |

**Unknown MDP: Model-Free**

| Goal *use features to generalize | Technique |
|---|---|
| Compute V*, Q*, π* | Q-learning |
| Evaluate a fixed policy π | Value Learning |

# Chapter 8: Regression

## Linear Regression

Use linear relationship to approximate the function of Y on X

Mean squared error (MSE): $\frac{1}{n}\sum(Y_i - \hat{Y}_i)^2$

### How to get linear model with minimal MSE

- MSE for model parameter $\theta$:

$$J(\theta) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \theta^\top x_i)^2$$

- Find an estimator $\hat{\theta}$ to minimize $J(\theta)$
- $y = \theta^\top x + b + \varepsilon$. Then we can write $x' = (1, x^1, \dots, x^d)$, $\theta = (b, \theta_1, \dots, \theta_d)$, then $y = \theta^\top x' + \varepsilon$

- Note that $J(\theta)$ is a convex function in $\theta$, so it has a unique minimal point

## $J(\theta)$ is convex

Check it by yourself !

- $f(x) = (y - x)^2 = (x - y)^2$ is convex in $x$

- $g(\theta) = f(\theta^\top x)$

$$
\begin{aligned}
& g\big((1-t)\theta_1 + t\theta_2\big) \\
&= f\left((1-t){\theta_1}^\top x + t{\theta_2}^\top x\right) \\
&\le (1-t) f\big({\theta_1}^\top x\big) + t f\big({\theta_2}^\top x\big) \\
&= (1-t) g(\theta_1) + t g(\theta_2)
\end{aligned}
$$

Convexity of f

- The sum of convex functions is convex
- Thus $J(\theta)$ is convex

Q: How to deal with non-linear relationships?

A: Taylor Expansion can be of some help. $\phi(x) = \left(1, x, x^2, \ldots, x^d\right)$

### Normal Equation

## Minimal point (Normal equation)

- $\frac{\partial J(\theta)}{\partial \theta} = \frac{2}{N}\sum_{i=1}^{N}(\theta^\top x_i - y_i)x_i = \frac{2}{N}\sum_{i=1}^{N}\left(x_i x_i^\top \theta - x_i y_i\right)$
- Letting the derivative be zero

$$\left(\sum_{i=1}^{N} x_i x_i^\top\right)\theta = \sum_{i=1}^{N} x_i y_i$$

- If we write $X = \begin{bmatrix} x_1^\top \\ \vdots \\ x_N^\top \end{bmatrix} = \begin{bmatrix} x_1^1 & \cdots & x_1^d \\ & \vdots & \\ x_N^1 & \cdots & x_N^d \end{bmatrix}$, $y = \begin{bmatrix} y_1 \\ \vdots \\ y_N \end{bmatrix}$, then

$$X^\top X\theta = X^\top y$$

**Geometric interpretation**

Using the combination of $\tilde{x}_i$ (column vector) to approximate the projection of y at their plane.

- $N = 3, d = 2$

$$\mathbf{X} = \begin{pmatrix} 1 & 2 \\ 1 & -2 \\ 1 & 2 \end{pmatrix}, \quad \mathbf{y} = \begin{pmatrix} 8.8957 \\ 0.6130 \\ 1.7761 \end{pmatrix}$$

$$\underset{\hat{\mathbf{y}} \in \mathrm{span}(\{\tilde{\mathbf{x}}_1, \ldots, \tilde{\mathbf{x}}_D\})}{\mathrm{argmin}} \quad \|\mathbf{y} - \hat{\mathbf{y}}\|_2.$$

column vectors in X

- $\hat{\theta} = (X^\top X)^{-1} X^\top y$
- $\hat{y} = X\hat{\theta} = X(X^\top X)^{-1} X^\top y$

Projection
hat matrix (put a "hat" on y)

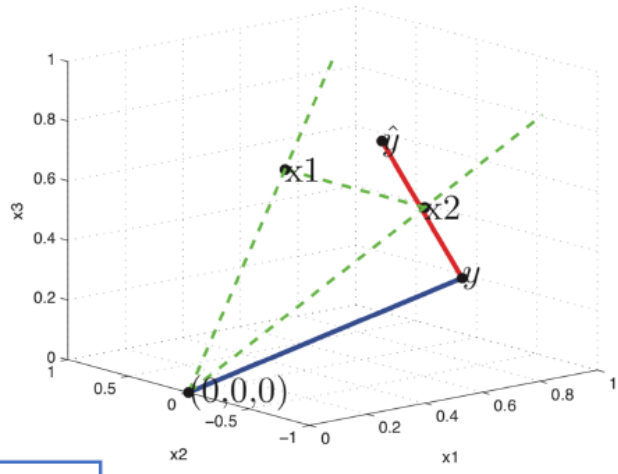



Figure credit: Kevin Murphy

18

## Gradient Methods

Motivation: large dataset

$$\theta_{\text{new}} \leftarrow \theta_{\text{old}} - \eta \frac{\partial \mathcal{L}(\theta)}{\partial \theta}$$

- Batch gradient descent
- Stochastic gradient descent

  Faster learning & Uncertainty in learning
- Mini-batch gradient descent

- A combination of batch GD and stochastic GD
- Split the whole dataset into *K* mini-batches

$$\{1, 2, 3, \ldots, K\}$$

- For each mini-batch *k*, perform one-step BGD towards minimizing

$$J^{(k)}(\theta) = \frac{1}{N_k} \sum_{i=1}^{N_k} (y_i - f_\theta(x_i))^2$$

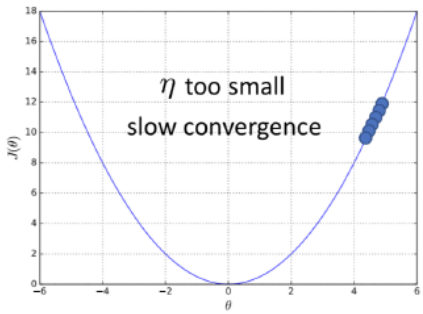
- Update $\theta_{\text{new}} = \theta_{\text{old}} - \eta \frac{\partial J^{(k)}(\theta)}{\partial \theta}$ for each mini-batch

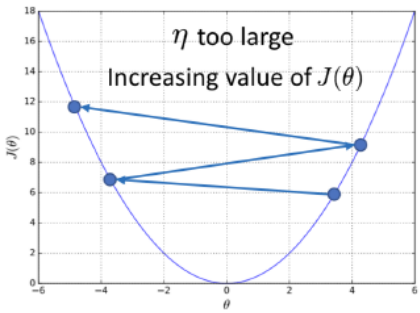
  - Good learning stability (BGD)
  - Good convergence rate (SGD)
  - Easy to parallelized

**Learning rate**

# Learning rate

$$\theta_{\text{new}} = \theta_{\text{old}} - \eta \frac{\partial J(\theta)}{\partial \theta}$$




- The initial point may be too far away from the optimal solution, which takes much time to converge

- May overshoot the minimum
- May fail to converge
- May even diverge

- To see if gradient descent is working, print out $J(\theta)$ for each or every several iterations. If $J(\theta)$ does not drop properly, adjust $\eta$

## Probabilistic View of MSE

# Probabilistic view

- Assume for each sampled $(x, y) \sim D$,
$$y = \theta^\top x + \varepsilon$$
where $\varepsilon$ is Gaussian noise and $\varepsilon \sim \mathcal{N}(0, \sigma^2)$, or equivalently, $y \sim \mathcal{N}(\theta^\top x, \sigma^2)$

- The linear regression estimator $\hat{\theta}$ is the maximal likelihood estimator (MLE) of the data

MLE view

- Given dataset $S$, find $\theta$ to maximize the likelihood of $S$, which is
$$\mathbb{P}[S|\theta] = \prod_{i=1}^{N} \mathbb{P}[y_i|x_i, \theta]$$
- $\hat{\theta} = \text{argmax}_\theta \mathbb{P}[S|\theta]$
$$= \text{argmax}_\theta \log \mathbb{P}[S|\theta]$$
$$= \text{argmax}_\theta \sum_{i=1}^{N} \log \mathbb{P}[y_i|x_i, \theta]$$
- $\mathbb{P}[y_i|x_i, \theta] = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y_i - \theta^\top x_i)^2}{2\sigma^2}}$ Gaussian distribution

36

MLE view (cont.)

- $\hat{\theta} = \text{argmax}_\theta \sum_{i=1}^{N} \log \mathbb{P}[y_i|x_i, \theta]$ $\mathbb{P}[y_i|x_i, \theta] = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y_i - \theta^\top x_i)^2}{2\sigma^2}}$
$$= \text{argmax}_\theta \sum_{i=1}^{N} \log \frac{1}{\sqrt{2\pi\sigma^2}} - \frac{(y_i - \theta^\top x_i)^2}{2\sigma^2}$$
$$= \text{argmax}_\theta - \sum_{i=1}^{N} (y_i - \theta^\top x_i)^2$$
$$= \text{argmin}_\theta \sum_{i=1}^{N} (y_i - \theta^\top x_i)^2$$ residual sum of squares (RSS)
$$= \text{argmin}_\theta \frac{1}{N} \sum_{i=1}^{N} (y_i - \theta^\top x_i)^2$$

## Regularization

Best model is to minimize both the <u>bias</u> and the <u>variance</u> (i.e. different data generate different model)

Ordinary least squares (OLS):

- Unbiased
- Can have huge variance
    1. Multi collinearity among data

2. Too many predictor variables

Solution: Reduce variance at the cost of introducing some bias

**Note**: Ridge = L2 norm, Lasso = L1 norm

## Ridge regression

- Regularization with L2 norm

$$L_{Ridge} = (y - X\theta)^2 + \lambda \|\theta\|_2^2$$

- $\lambda \to 0, \hat{\theta}_{Ridge} \to \hat{\theta}_{OLS}$
- $\lambda \to \infty, \hat{\theta} \to 0$
- As $\lambda$ becomes larger, the variance decreases but the bias increases
- $\lambda$: Trade-off between bias and variance
  - Choose by cross-validation
- Ridge regression decreases the complexity of a model but does not reduce the number of variables (compared to other regularization like Lasso)

## Solution of the ridge regression

- $\frac{\partial L_{Ridge}}{\partial \theta} = 2\sum_{i=1}^{N}(\theta^\top x_i - y_i)x_i + 2\lambda\theta$
- Letting the derivative be zero

$$\left(\lambda I + \sum_{i=1}^{N} x_i x_i^\top\right)\theta = \sum_{i=1}^{N} x_i y_i$$

- If we write $X = \begin{bmatrix} x_1^\top \\ \vdots \\ x_N^\top \end{bmatrix} = \begin{bmatrix} x_1^1 & \cdots & x_1^d \\ & \vdots & \\ x_N^1 & \cdots & x_N^d \end{bmatrix}, y = \begin{bmatrix} y_1 \\ \vdots \\ y_N \end{bmatrix}$, then

$$(\lambda I + X^\top X)\theta = X^\top y$$

$$\hat{\theta}_{\text{ridge}} = (\lambda I + X^\top X)^{-1} X^\top y$$

Recall the normal equation for OLS is $X^\top X\theta = X^\top y$

Always invertible

### Probabilistic View of L2 Regularization

Maximum A Posteriori (MAP)

$$P(\theta \mid X) = \frac{P(X \mid \theta) \times P(\theta)}{P(X)}$$

$$
\begin{aligned}
\hat{\theta}_{\text{MAP}} &= \arg\max P(\theta \mid X) \\
&= \arg\min -\log P(\theta \mid X) \\
&= \arg\min -\log P(X \mid \theta) - \log P(\theta) + \log P(X) \\
&= \arg\min -\log P(X \mid \theta) - \log P(\theta)
\end{aligned}
$$

## Probabilistic view (MAP)

- Ridge regression estimator is an MAP estimator with Gaussian prior
- Suppose $\theta$ has the prior $P(\theta) = \mathcal{N}(0, \tau^2 I)$
- $\hat{\theta}_{\text{MAP}} = \text{argmax}_\theta \sum_{i=1}^{N} \log \mathbb{P}[y_i | x_i, \theta] + \log P(\theta)$

$$f_{\mathbf{X}}(x_1, \ldots, x_k) = \frac{\exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right)}{\sqrt{(2\pi)^k|\boldsymbol{\Sigma}|}}$$

$$\mathbb{P}[y_i|x_i, \theta] = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y_i - \theta^\top x_i)^2}{2\sigma^2}}$$

$$
\begin{aligned}
&= \text{argmax}_\theta \sum_{i=1}^{N} -\frac{(y_i - \theta^\top x_i)^2}{2\sigma^2} - \frac{1}{2\tau^2}\|\theta\|_2^2 \\
&= \text{argmax}_\theta -\frac{1}{N}\sum_{i=1}^{N}(y_i - \theta^\top x_i)^2 - \lambda\|\theta\|_2^2 \\
&= \text{argmin}_\theta \frac{1}{N}\sum_{i=1}^{N}(y_i - \theta^\top x_i)^2 + \lambda\|\theta\|_2^2
\end{aligned}
$$

$\lambda = \sigma^2 / N\tau^2$

## Logistic Regression

Classification problem: predict the labels from many numerical features

- Given:
  - A description of an instance $x \in X$
  - A fixed set of categories: $C = \{c_1, c_2, \ldots, c_m\}$
- Determine:
  - The category of $x : f(x) \in C$ where $f(x)$ is a categorization function whose domain is $X$ and whose range is $C$
  - If the category set binary, i.e. $C = \{0, 1\}$ ({false, true}, {negative, positive}) then it is called binary classification

### Loss Function

## KL divergence

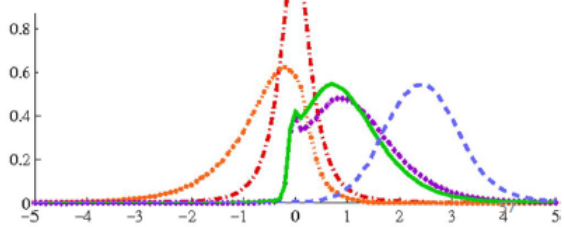
- Regression: mean squared error (MSE)
- Kullback-Leibler divergence (KL divergence)
  - Measure the dissimilarity of two probability distributions

$$
\begin{aligned}
\mathbb{KL}(p||q) &\triangleq \sum_{k=1}^{K} p_k \log \frac{p_k}{q_k} \\
&= \sum_k p_k \log p_k - \sum_k p_k \log q_k = -\mathbb{H}(p) + \mathbb{H}(p, q)
\end{aligned}
$$

Entropy

Cross entropy

Question:
Which one is more similar to normal distribution?



- Information inequality

$$\mathbb{KL}\left(p||q\right) \geq 0 \textit{ with equality iff } p = q.$$

- Entropy
  - $\mathbb{H}(X) \triangleq -\sum_{k=1}^{K} p(X=k)\log_2 p(X=k)$
  - Is a measure of the uncertainty
  - Discrete distribution with the maximum entropy is the uniform distribution

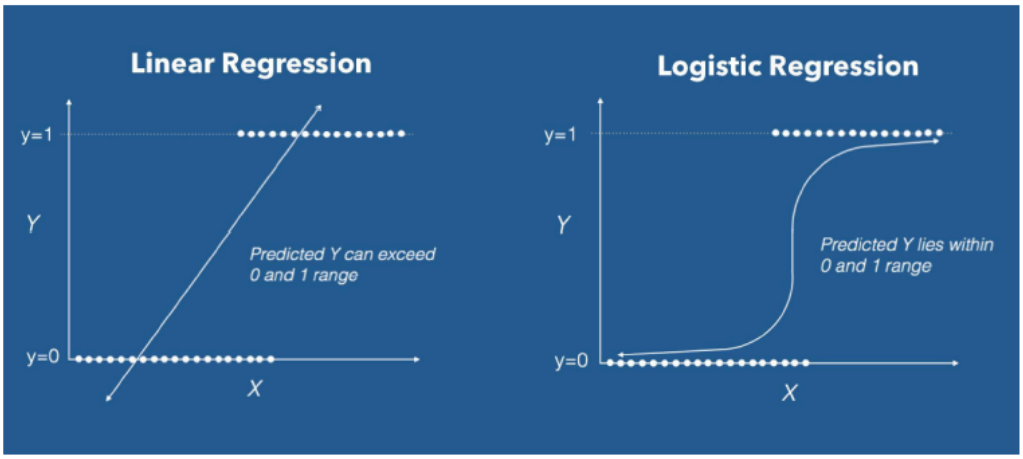
- Cross entropy
  - $\mathbb{H}(p,q) \triangleq -\sum_{k} p_k \log q_k$
  - Is the average number of bits needed to encode data coming from a source with distribution p when we use model q to define our codebook

**Cross entropy loss**

$$\begin{aligned}\mathcal{L} &= -\sum_{i=1}^{m} 1_{y=c_k} \log p_\theta\left(C_k \mid x\right) \\ &= -y\log p_\theta(1 \mid x) - (1-y)\log\left(1 - p_\theta(1 \mid x)\right)\end{aligned}$$

## Binary Classification

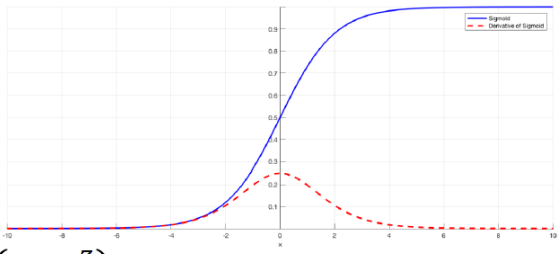
Binary classification: linear and logistic




Linear regression: $h_\theta(x) = \theta^\top x$

Logistic regression: $h_\theta(x) = \sigma(\theta^\top x) = \frac{1}{1+e^{-\theta^\top x}}$

# Properties for the sigmoid function




- $\sigma(z) = \frac{1}{1+e^{-z}}$
  - Bounded in (0,1)
  - $\sigma(z) \to 1$ when $z \to \infty$
  - $\sigma(z) \to 0$ when $z \to -\infty$
- $\sigma'(z) = \frac{d}{dz}\frac{1}{1+e^{-z}} = -(1+e^{-z})^{-2} \cdot (-e^{-z})$

$$= \frac{1}{1+e^{-z}}\frac{e^{-z}}{1+e^{-z}}$$
$$= \frac{1}{1+e^{-z}}\left(1 - \frac{1}{1+e^{-z}}\right)$$
$$= \sigma(z)\big(1-\sigma(z)\big)$$

# Logistic regression

- Binary classification

$$p_\theta(y=1|x) = \sigma(\theta^\top x) = \frac{1}{1+e^{-\theta^\top x}}$$
$$p_\theta(y=0|x) = \frac{e^{-\theta^\top x}}{1+e^{-\theta^\top x}}$$

- Cross entropy loss function

is also convex in $\theta$

$$\mathcal{L}(y, x, p_\theta) = -y \log \sigma(\theta^\top x) - (1-y)\log(1-\sigma(\theta^\top x))$$

- Gradient

$$\frac{\partial \mathcal{L}(y,x,p_\theta)}{\partial \theta} = -y\frac{1}{\sigma(\theta^\top x)}\sigma(z)(1-\sigma(z))x - (1-y)\frac{-1}{1-\sigma(\theta^\top x)}\sigma(z)(1-\sigma(z))x$$
$$= (\sigma(\theta^\top x) - y)x$$
$$\theta \leftarrow \theta + \eta(y - \sigma(\theta^\top x))x$$

$$\boxed{\frac{\partial \sigma(z)}{\partial z} = \sigma(z)(1-\sigma(z))}$$

$$\theta_{\text{new}} \leftarrow \theta_{\text{old}} - \eta\frac{\partial \mathcal{L}(\theta)}{\partial \theta}$$

67

**Threshold**

Threshold is trade-off.

$$\hat{y} = \begin{cases} 1, & p_\theta(y=1 \mid x) > h \\ 0, & \text{otherwise} \end{cases}$$

Precision = TP / (TP + FP)  (i.e. How many TP among predicted P samples?)

Recall = TP / (TP + FN)  (i.e. How many TP among original P samples?)

- Higher threshold

  More FN and less FP & Higher precision & Lower recall

- Lower threshold

  More FP and less FN & Lower precision & Higher recall

Note: The decision boundary is linear!

Note: logit = $\theta^\top x$

### Multi-Class Classification (*)

## Multi-Class Logistic Regression

- Class set $C = \{c_1, c_2, \ldots, c_m\}$
- Predicting the probability of $p_\theta(y = c_j|x)$

$$p_\theta(y = c_j|x) = \frac{e^{\theta_j^\top x}}{\sum_{k=1}^m e^{\theta_k^\top x}} \quad \text{for } j = 1, \ldots, m$$

- Softmax
  - Parameters $\theta = \{\theta_1, \theta_2, \ldots, \theta_m\}$
  - Can be normalized with m-1 groups of parameters

Learning on one instance $(x, y = c_j)$, maximize log-likelihood $\log p_\theta(y = c_j \mid x)$

# Chapter 9: Neural Network

## Activation Functions

### Sigmoid

$$\sigma(z) = \frac{1}{1 + e^{-z}} \in (0, 1)$$
$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$

However, saturated neurons make value vanished.

$$f(f(f(\cdots)))$$
$$f([0, 1]) \subseteq [0.5, 0.732)$$
$$f([0.5, 0.732)) \subseteq (0.622, 0.676)$$

### Tanh

$$\tanh(z) = \frac{\sinh(z)}{\cosh(z)} = \frac{e^z - e^{-z}}{e^z + e^{-z}} \in (-1, 1)$$
$$\tanh'(z) = 1 - \tanh^2(z)$$

Less likely to get “stuck” during training

### Rectified Linear Unit (ReLU)

$$\mathrm{ReLU}(z) = \max(0, z)$$
$$\sim f_{\text{softplus}}(x) = \log(1 + e^x)$$

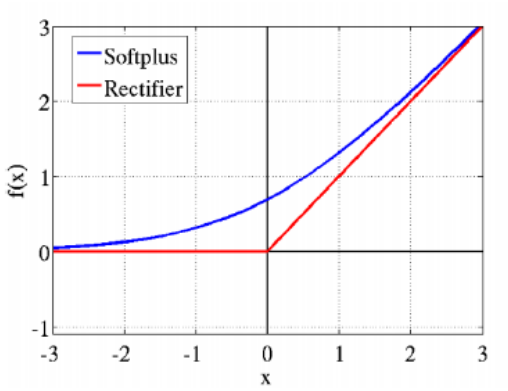
- ReLU’s gradient doesn't vanish as x increases
- Speed up training of neural networks (less computation)

- The gradient on positive portion is larger than sigmoid or tanh functions
  - Update more rapidly
  - The left “dead neuron” part can be ameliorated by Leaky ReLU

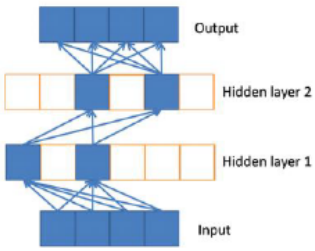
# ReLU activation function (cont.)

- ReLU function

$$\mathrm{ReLU}(z) = \max(0, z)$$




- The only non-linearity comes from the path selection with individual neurons being active or not
- It allows sparse representations:
  - for a given input only a subset of neurons are active

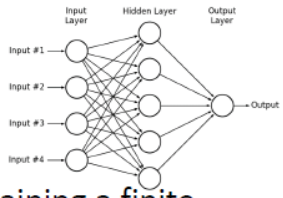



Sparse propagation of activations and gradients

20

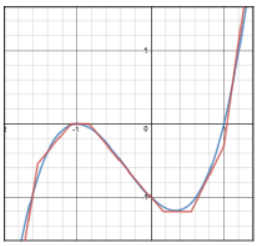
## Neural networks

# Universal approximation theorem



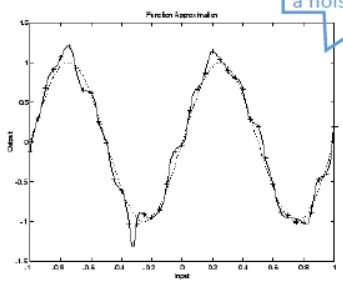

- A feed-forward network with a single hidden layer containing a finite number of neurons can approximate continuous functions

1-20-1 NN approximates a noisy sine function



$$n_1(x) = Relu(-5x - 7.7)$$
$$n_2(x) = Relu(-1.2x - 1.3)$$
$$n_3(x) = Relu(1.2x + 1)$$
$$n_4(x) = Relu(1.2x - .2)$$
$$n_5(x) = Relu(2x - 1.1)$$
$$n_6(x) = Relu(5x - 5)$$

$$Z(x) = -n_1(x) - n_2(x) - n_3(x) + n_4(x) + n_5(x) + n_6(x)$$



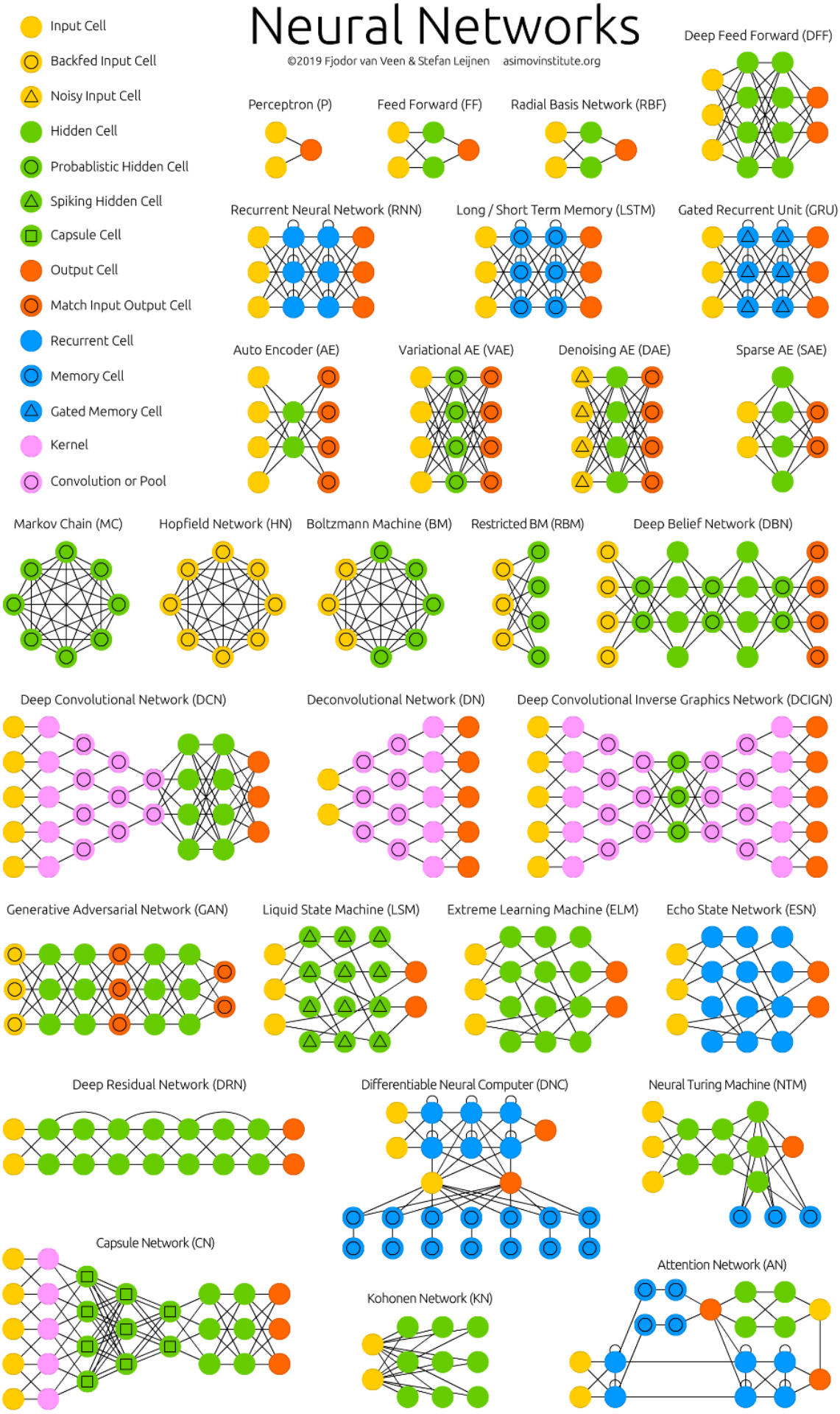
Hornik, Kurt, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators." *Neural networks* 2.5 (1989): 359-366

23

A mostly complete chart of

# Neural Networks

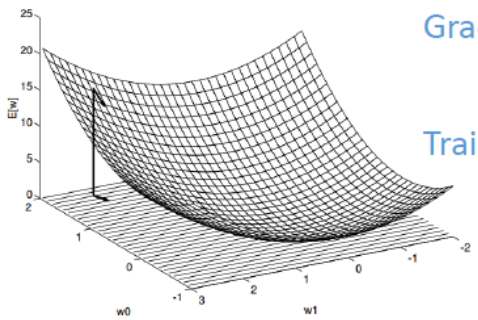
©2019 Fjodor van Veen & Stefan Leijnen asimovinstitute.org



## Backpropagation

Given the topology of the network (number of layers, number of neurons, their connections), find a set of weights to minimize the error function

# Gradient descent




Gradient

$$\nabla E[\vec{w}] \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots \frac{\partial E}{\partial w_n}\right]$$

Training Rule

$$\Delta \vec{w} = -\eta \nabla E[\vec{w}]$$

i.e.

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

Partial derivatives are the key

Update

$$w_i \leftarrow w_i + \Delta w_i$$

Key Idea: The chain rule.

# Backpropagation algorithms

- Error function $E = \frac{1}{2}\sum_k (y_k - t_k)^2$
- $\delta_k^{(2)} = (t_k - y_k) y_k (1 - y_k)$
- $\Rightarrow w_{k,j}^{(2)} \leftarrow w_{k,j}^{(2)} + \eta \delta_k^{(2)} h_j^{(1)}$
- $\delta_j^{(1)} = h_j^{(1)} \left(1 - h_j^{(1)}\right) \sum_k w_{k,j}^{(2)} \delta_k^{(2)}$
- $\Rightarrow w_{j,m}^{(1)} \leftarrow w_{j,m}^{(1)} + \eta \delta_j^{(1)} x_m$

- Activation function: sigmoid

Initialize all weights to small random numbers
Do until convergence

- For each training example:
  1. Input it to the network and compute the network output
  2. For each output unit $k$, $o_k$ is the output of unit $k$

     $$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$$

  3. For each hidden unit $j$, $o_j$ is the output of unit $j$

     $$\delta_j \leftarrow o_j(1 - o_j) \sum_{k \in \text{next layer}} w_{k,j} \delta_k$$

  4. Update each network weight, where $x_i$ is the output for unit $i$

     $$w_{j,i} \leftarrow w_{j,i} + \eta \delta_j x_i$$

## Using New Weights

# Calculation example (cont.)

- For each output unit $k$, $o_k$ is the output of unit $k$

$$\delta_k \leftarrow o_k(1-o_k)(t_k-o_k)$$

- For each hidden unit $j$, $o_j$ is the output of unit $j$

$$\delta_j \leftarrow o_j(1-o_j)\sum_{k \in next\ layer} w_{k,j}\delta_k$$

- Update each network weight, where $x_i$ is the input for unit $j$

$$w_{j,i} \leftarrow w_{j,i} + \eta\delta_j x_i$$

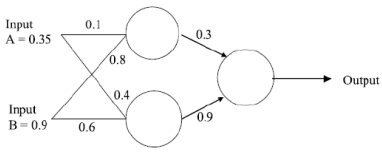
Answer:
(i)
Input to top neuron = (0.35x0.1)+(0.9x0.8)=0.755. Out = 0.68.
Input to bottom neuron = (0.9x0.6)+(0.35x0.4) = 0.68. Out = 0.6637.
Input to final neuron = (0.3x0.68)+(0.9x0.6637) = 0.80133. Out = 0.69.

(ii)
Output error δ=(t-o)(1-o)o = (0.5-0.69)(1-0.69)0.69 = -0.0406.

New weights for output layer
$w1^+$ = w1+(δ x input) = 0.3 + (-0.0406x0.68) = 0.272392.
$w2^+$ = w2+(δ x input) = 0.9 + (-0.0406x0.6637) = 0.87305.

Errors for hidden layers:
δ1 = δ x w1 = -0.0406 x 0.272392 x (1-o)o = $-2.406\text{x}10^{-3}$
δ2= δ x w2 = -0.0406 x 0.87305 x (1-o)o = $-7.916\text{x}10^{-3}$

New hidden layer weights:
$w3^+$=0.1 + (-2.406 x $10^{-3}$ x 0.35) = 0.09916.
$w4^+$ = 0.8 + (-2.406 x $10^{-3}$ x 0.9) = 0.7978.
$w5^+$ = 0.4 + (-7.916 x $10^{-3}$ x 0.35) = 0.3972.
$w6^+$ = 0.6 + (-7.916 x $10^{-3}$ x 0.9) = 0.5928.




(iii)
Old error was -0.19. New error is -0.18205. Therefore error has reduced.

## Using Old Weights

Note: Only the first layer is different.

# Calculation example (cont.)


- For each output unit $k$, $o_k$ is the output of unit $k$

$$\delta_k \leftarrow o_k(1-o_k)(t_k-o_k)$$

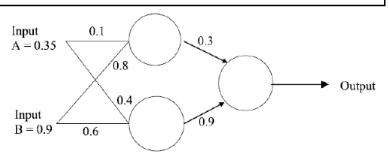
- For each hidden unit $j$, $o_j$ is the output of unit $j$

$$\delta_j \leftarrow o_j(1-o_j)\sum_{k \in next\ layer} w_{k,j}\delta_k$$

- Update each network weight, where $x_i$ is the input for unit $j$

$$w_{j,i} \leftarrow w_{j,i} + \eta\delta_j x_i$$


- Answer (i)
  - Input to top neuron $= 0.35 \times 0.1 + 0.9 \times 0.8 = 0.755$. Out=0.68
  - Input to bottom neuron $= 0.35 \times 0.4 + 0.9 \times 0.6 = 0.68$. Out= 0.6637
  - Input to final neuron $= 0.3 \times 0.68 + 0.9 \times 0.6637 = 0.80133$. Out= 0.69
- (ii) It is both OK to use new or old weights when computing $\delta_j$ for hidden units
  - Output error $\delta = (t-o)o(1-o) = (0.5-0.69) \times 0.69 \times (1-0.69) = -0.0406$
  - Error for top hidden neuron $\delta_1 = 0.68 \times (1-0.68) \times 0.3 \times (-0.0406) = -0.00265$
  - Error for top hidden neuron $\delta_2 = 0.6637 \times (1-0.6637) \times 0.9 \times (-0.0406) = -0.008156$
  - New weights for the output layer
  - $w_{o1} = 0.3 - 0.0406 \times 0.68 = 0.272392$
  - $w_{o2} = 0.9 - 0.0406 \times 0.6637 = 0.87305$
  - New weights for the hidden layer
  - $w_{1A} = 0.1 - 0.00265 \times 0.35 = 0.0991$
  - $w_{1B} = 0.8 - 0.00265 \times 0.9 = 0.7976$
  - $w_{2A} = 0.4 - 0.008156 \times 0.35 = 0.3971$
  - $w_{2B} = 0.6 - 0.008156 \times 0.9 = 0.5927$
- (iii)
  - Input to top neuron $= 0.35 \times 0.0991 + 0.9 \times 0.7976 = 0.7525$. Out=0.6797
  - Input to bottom neuron $= 0.35 \times 0.3971 + 0.9 \times 0.5927 = 0.6724$. Out= 0.662
  - Input to final neuron $= 0.272392 \times 0.6797 + 0.87305 \times 0.662 = 0.7631$. Out= 0.682
  - New error is $-0.182$, which is reduced compared to old error $-0.19$




## Misc (*)

## More on backpropagations

- It is gradient descent over entire network weight vectors
- Easily generalized to arbitrary directed graphs
- Will find a local, not necessarily global error minimum
  - In practice, often work well (can run multiple times)
- Often include weight momentum $\alpha$

$$\Delta w_{j,i}(n) \leftarrow \eta \delta_j x_i + \alpha \Delta w_{j,i}(n-1)$$

- Minimizes error over *training* examples
  - Will it generalize well to unseen examples?
- Training can take thousands of iterations, slow!
- Using network after training is very fast

### Overfitting (*)

## Avoid overfitting

- Penalize large weights: $$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2 + \boxed{\gamma \sum_{i,j} w_{ji}^2}$$
- Train on target slopes as well as values:

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} \left[ (t_{kd} - o_{kd})^2 + \mu \sum_{j \in inputs} \left( \frac{\partial t_{kd}}{\partial x_d^j} - \frac{\partial o_{kd}}{\partial x_d^j} \right)^2 \right]$$

- Weight sharing
  - Reduce total number of weights
  - Using structures, like convolutional neural networks
- Early stopping
- Dropout

# Chapter 10: Bayes Net Probabilistic Models

## Probability

Probabilistic Models: joint distribution over a set of random variables.

- Joint Distributions
- Marginal Distributions

  sub-tables which eliminate variables
- Conditional Distributions

  Conditional Probabilities: $P(a \mid b) = P(a, b)/P(b)$

  probability distributions over some variables given fixed values of others

  1. SELECT the joint probabilities matching the evidence (i.e. condition)

2. NORMALIZE the selection

## Probabilistic Inference

Compute a desired probability from other known probabilities (e.g. conditional from joint)

### Inference by Enumeration

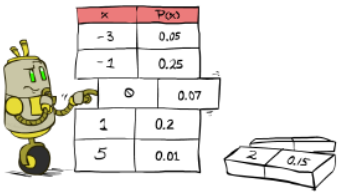
- General case:
  - Evidence variables: $E_1 \dots E_k = e_1 \dots e_k$
  - Query* variable: $Q$
  - Hidden variables: $H_1 \dots H_r$

$\left.\right\} X_1, X_2, \dots X_n$ *All variables*

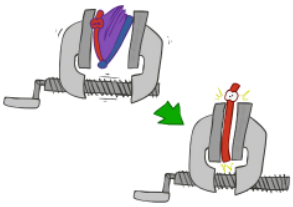
- We want: $P(Q|e_1 \dots e_k)$

** Works fine with multiple query variables, too*

- Step 1: Select the entries consistent with the evidence




- Step 2: Sum out H to get joint of Query and evidence



$$P(Q, e_1 \dots e_k) = \sum_{h_1 \dots h_r} \underbrace{P(Q, h_1 \dots h_r, e_1 \dots e_k)}_{X_1, X_2, \dots X_n}$$

- Step 3: Normalize

$$\times \frac{1}{Z}$$

$$Z = \sum_q P(Q, e_1 \cdots e_k)$$

$$P(Q|e_1 \cdots e_k) = \frac{1}{Z} P(Q, e_1 \cdots e_k)$$

### Answer Any Query from Joint Distributions 2

- Two tools to go from joint to query
- Joint: $P(H_1, H_2, Q, E)$
- Query: $P(Q \mid e)$

1. Definition of conditional probability

$$P(Q|e) = \frac{P(Q, e)}{P(e)}$$

2. Law of total probability (marginalization, summing out)

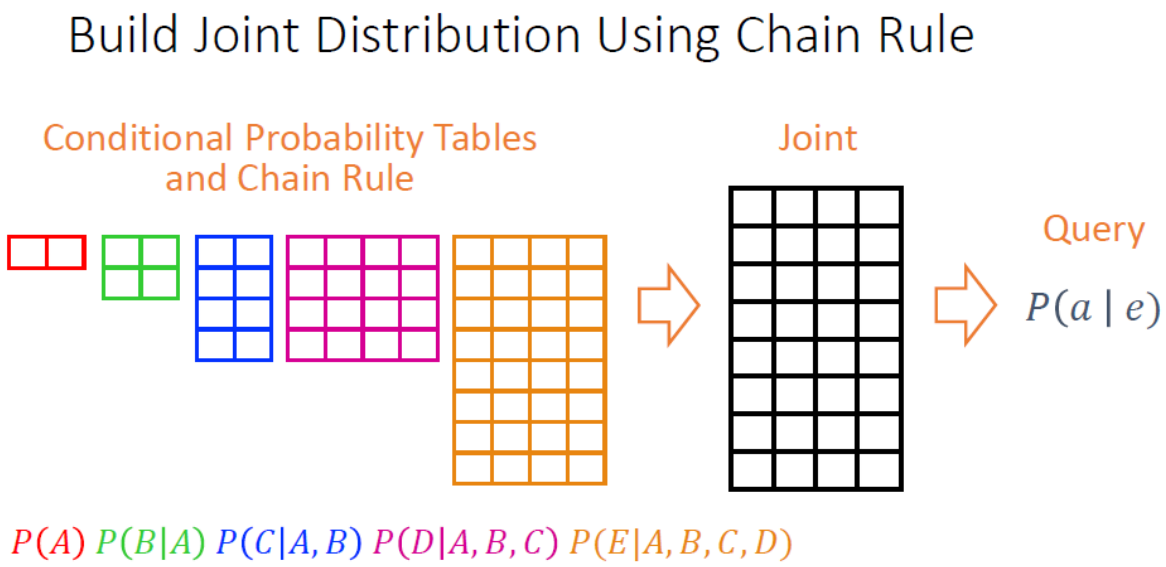
$$P(Q, e) = \sum_{h_1} \sum_{h_2} P(h_1, h_2, Q, e)$$

$$P(e) = \sum_q \sum_{h_1} \sum_{h_2} P(h_1, h_2, q, e)$$

Only need to compute $P(Q, e)$ then normalize

## Build Joint Distribution Using Chain Rule

$$P(x_1, x_2, x_3) = P(x_1) P(x_2 \mid x_1) P(x_3 \mid x_1, x_2)$$
$$P(x_1, x_2, \dots x_n) = \prod_i P(x_i \mid x_1 \dots x_{i-1})$$




## Conditional Independence

Independence: $P(x, y) = P(x)P(y)$

# Conditional Independence (cont.)

- Unconditional (absolute) independence very rare (why?)
- *Conditional independence* is our most basic and robust form of knowledge about uncertain environments.
- X is conditionally independent of Y given Z $\quad X \perp\!\!\!\perp Y | Z$

  if and only if:

  $$\forall x, y, z : P(x, y|z) = P(x|z)P(y|z)$$

  or, equivalently, if and only if

  $$\forall x, y, z : P(x|z, y) = P(x|z)$$

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

## Bayes' Nets

Problems with full joint tables

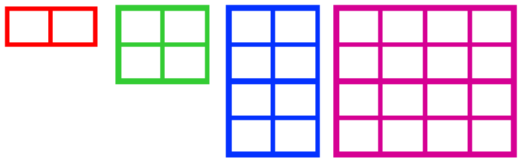
- Too big to represent explicitly
- Hard to learn (estimate) anything empirically about more than a few variables at a time

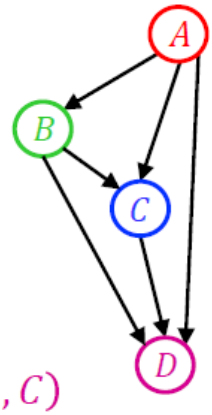
**Bayes' nets**: describe complex joint distributions (models) using simple, local distributions (conditional probabilities)

# Bayesian Networks

Bayes net

- One node per random variable, DAG
- One conditional probability table (CPT) per node:
  P(node | *Parents*(node) )






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

Encode joint distributions as product of conditional distributions on each variable

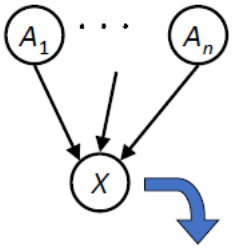
$$P(X_1, \dots, X_N) = \prod_i P(X_i|\ Parents(X_i))$$

# 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

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

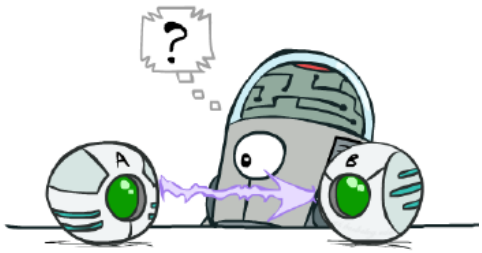
  - CPT: conditional probability table
  - Description of a noisy "causal" process




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

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

# Causality?




- When Bayes' nets reflect the true causal patterns:
  - Often simpler (nodes have fewer parents)
  - Often easier to think about
  - Often easier to elicit from experts
- BNs need not actually be causal
  - Sometimes no causal net exists over the domain (especially if variables are missing)
  - E.g. consider the variables *Traffic* and *Drips*
  - End up with arrows that reflect correlation, not causation
- What do the arrows really mean?
  - Topology may happen to encode causal structure
  - Topology really encodes conditional independence

$$P(x_i|x_1, \ldots x_{i-1}) = P(x_i|parents(X_i))$$

Bayes' nets implicitly encode joint distributions

- As a product of local conditional distributions
- Multiply all the relevant conditionals together

$$P(X_1 \ldots X_2) = \prod_i P(X_i \mid \text{ Parents } (X_i))$$

## Conditional Independence Semantics

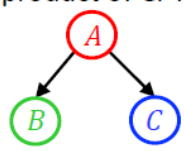
# Conditional Independence Semantics

- For the following Bayes nets, write the joint $P(A, B, C)$
  1. Using the chain rule (with top-down order A,B,C)
  2. Using Bayes net semantics (product of CPTs)

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

$P(A)\ P(B|A)\ P(C|B)$

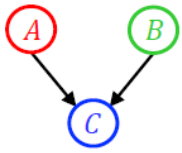
Assumption:
$P(C|A,B) = P(C|B)$
C is independent from A given B




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

$P(A)\ P(B|A)\ P(C|A)$

Assumption:
$P(C|A,B) = P(C|A)$
C is independent from B given A




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

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

Assumption:
$P(B|A) = P(B)$
A is independent from B given { }

## Causal Chain

### Causal Chains

- This configuration is a "causal chain"

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

- Guaranteed X independent of Z ?
- *No!*
  - One example set of CPTs for which X is not independent of Z is sufficient to show this independence is not guaranteed.
  - Example:
    - Low pressure causes rain causes traffic, high pressure causes no rain causes no traffic
    - In numbers:

      P( +y | +x ) = 1, P( -y | - x ) = 1,
      P( +z | +y ) = 1, P( -z | -y ) = 1

103

### Causal Chains 2

- This configuration is a "causal chain"

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

- Guaranteed X independent of Z given Y?

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

*Yes!*

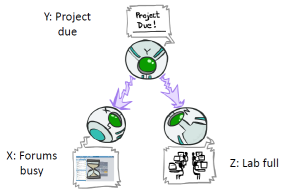
- Evidence along the chain "blocks" the influence

104

## Common Causes

### Common Causes

- This configuration is a "common cause"




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

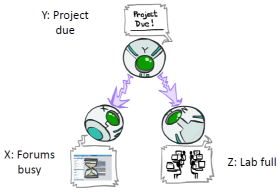
- Guaranteed X independent of Z ?
- *No!*
  - One example set of CPTs for which X is not independent of Z is sufficient to show this independence is not guaranteed.
  - Example:
    - Project due causes both forums busy and lab full
    - In numbers:

      P( +x | +y ) = 1, P( -x | -y ) = 1,
      P( +z | +y ) = 1, P( -z | -y ) = 1

105

### Common Cause 2

- This configuration is a "common cause"




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

- Guaranteed X and Z independent given Y?

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

*Yes!*

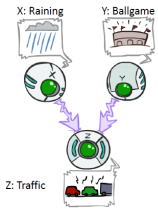
- Observing the cause blocks influence between effects

106

## Common Effect

### Common Effect

- Last configuration: two causes of one effect (v-structures)



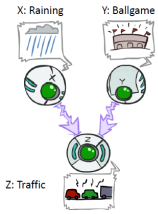

- Are X and Y independent?
  - *Yes*: the ballgame and the rain cause traffic, but they are not correlated
- Proof:

$$P(x, y) = \sum_z P(x, y, z)$$
$$= \sum_z P(x)P(y)P(z|x, y)$$
$$= P(x)P(y)\sum_z P(z|x, y)$$
$$= P(x)P(y)$$

107

### Common Effect 2

- Last configuration: two causes of one effect (v-structures)




- Are X and Y independent?
  - *Yes*: the ballgame and the rain cause traffic, but they are not correlated
  - (Proved previously)
- Are X and Y independent given Z?
  - *No*: seeing traffic puts the rain and the ballgame in competition as explanation.
- This is backwards from the other cases
  - Observing an effect activates influence between possible causes

108

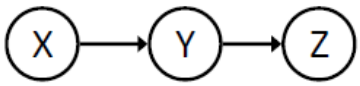
## Bayes Nets: Independence

### Bayes Nets: Assumptions

- Assumptions we are required to make to define the Bayes net when given the graph:
  $$P(x_i|x_1 \cdots x_{i-1}) = P(x_i|parents(X_i))$$
- Beyond those "chain rule → Bayes net" conditional independence assumptions
  - Often additional conditional independences
  - They can be read off the graph
- Important for modeling: understand assumptions made when choosing a Bayes net graph

# Independence in a BN

- Important question about a BN:
  - Are two nodes independent given certain evidence?
  - If yes, can prove using algebra (tedious in general)
  - If no, can prove with a counter example
  - Example:




  - Question: are X and Z necessarily independent?
    - Answer: no. Example: low pressure causes rain, which causes traffic.
    - X can influence Z, Z can influence X (via Y)
    - Addendum: they *could* be independent: how?

**General question**: in a given BN, are two variables independent (given evidence)?

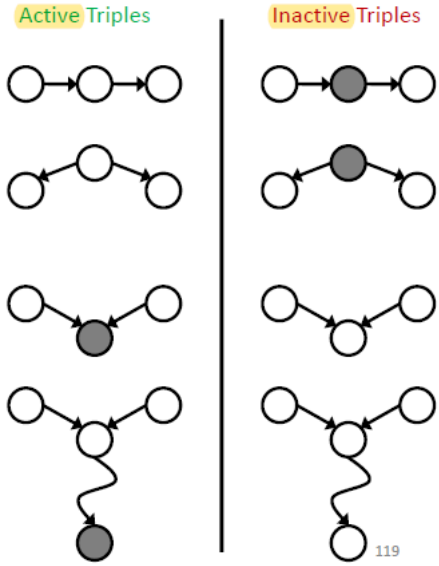
**Bayes Ball / D-separation Algorithm**

# 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



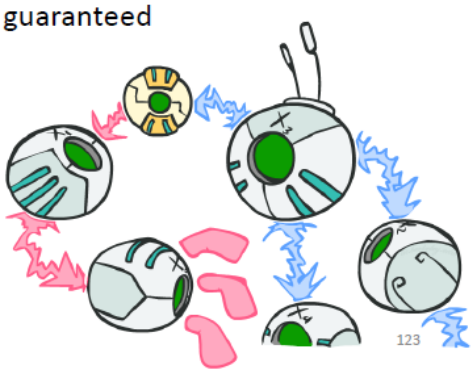
# More Variables

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

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

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

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

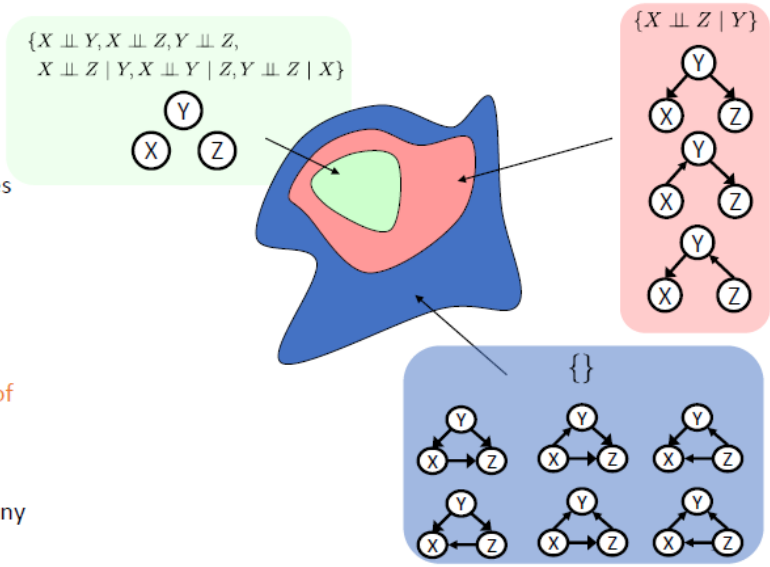



## Structure Implications (*)

Given a Bayes net structure, can run Bayes ball/d-separation algorithm to build a complete list of conditional independences.

# Topology Limits Distributions

- Given some graph topology G, only certain joint distributions can be encoded
- The graph structure guarantees certain (conditional) independences
- (There might be more independence)
- Adding arcs increases the set of distributions, but has several costs
- Full conditioning can encode any distribution




## Summary

## Bayes Nets Representation Summary

- Bayes nets compactly encode joint distributions (by making use of conditional independences!)
- Guaranteed independencies of distributions can be deduced from BN graph structure
- Bayes ball/D-separation gives precise conditional independence guarantees from graph alone
- A Bayes net's joint distribution may have further (conditional) independence that is not detectable until you inspect its specific distribution

# Chapter 11: Bayes Net Inference

## Bayes Rule

$$P(x \mid y) = \frac{P(y \mid x) \cdot P(x)}{P(y)}$$

Build posterior distribution with prior distribution and condition.

## Inference

### Inference by Enumeration

Multiple Join, Multiple Eliminate

## Inference by Enumeration: Procedural Outline

- Track objects called factors
- Initial factors are local CPTs (one per node)

$P(R)$

| | |
|---|---|
| +r | 0.1 |
| -r | 0.9 |

$P(T|R)$

| | | |
|---|---|---|
| +r | +t | 0.8 |
| +r | -t | 0.2 |
| -r | +t | 0.1 |
| -r | -t | 0.9 |

$P(L|T)$

| | | |
|---|---|---|
| +t | +l | 0.3 |
| +t | -l | 0.7 |
| -t | +l | 0.1 |
| -t | -l | 0.9 |

- Any known values are selected
  - E.g. if we know $L = +\ell$, the initial factors are

$P(R)$

| | |
|---|---|
| +r | 0.1 |
| -r | 0.9 |

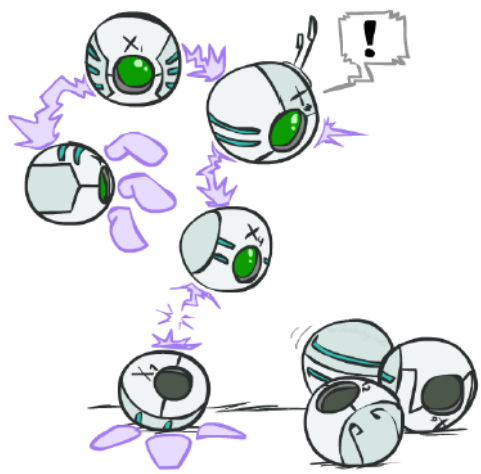
$P(T|R)$

| | | |
|---|---|---|
| +r | +t | 0.8 |
| +r | -t | 0.2 |
| -r | +t | 0.1 |
| -r | -t | 0.9 |

$P(+\ell|T)$

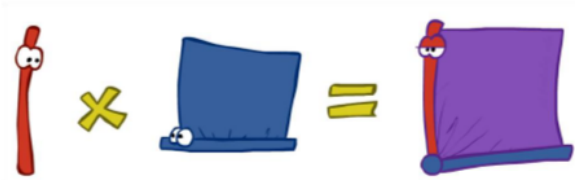
| | | |
|---|---|---|
| +t | +l | 0.3 |
| -t | +l | 0.1 |

- Procedure: Join all factors, then sum out all hidden variables

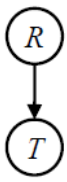


16

# Operation 1: Join Factors



- First basic operation: joining factors
- Combining factors:
  - Just like a database join
  - Get all factors over the joining variable
  - Build a new factor over the union of the variables involved
- Example: Join on R




$P(R) \times P(T|R) \rightarrow P(R,T)$

| | |
|---|---|
| +r | 0.1 |
| -r | 0.9 |

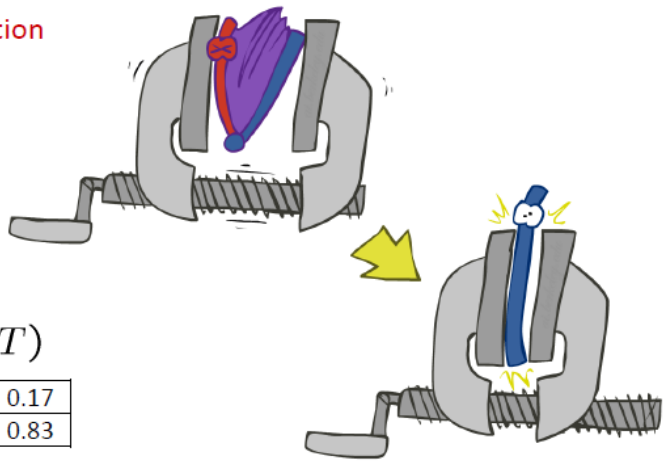
| | | |
|---|---|---|
| +r | +t | 0.8 |
| +r | -t | 0.2 |
| -r | +t | 0.1 |
| -r | -t | 0.9 |

| | | |
|---|---|---|
| +r | +t | 0.08 |
| +r | -t | 0.02 |
| -r | +t | 0.09 |
| -r | -t | 0.81 |

- Computation for each entry: pointwise products $\forall r, t: \quad P(r,t) = P(r) \cdot P(t|r)$

# Operation 2: Eliminate

- Second basic operation: marginalization
- Take a factor and sum out a variable
  - Shrinks a factor to a smaller one
  - A projection operation



- Example:

$P(R,T)$

| | | |
|---|---|---|
| +r | +t | 0.08 |
| +r | -t | 0.02 |
| -r | +t | 0.09 |
| -r | -t | 0.81 |

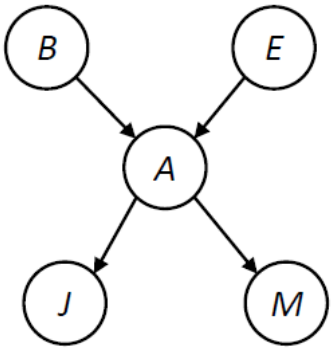
sum $R$ $\rightarrow$

$P(T)$

| | |
|---|---|
| +t | 0.17 |
| -t | 0.83 |

# Inference by Enumeration in Bayes Net

- Reminder of inference by enumeration:
  - Any probability of interest can be computed by summing entries from the joint distribution
  - Entries from the joint distribution can be obtained from a BN by multiplying the corresponding conditional probabilities




$$
\begin{aligned}
P(B \mid j, m) &= \alpha\, P(B, j, m) \\
&= \alpha \textstyle\sum_{e,a} P(B, e, a, j, m) \\
&= \alpha \textstyle\sum_{e,a} P(B)\, P(e)\, P(a \mid B, e)\, P(j \mid a)\, P(m \mid a)
\end{aligned}
$$

- So inference in Bayes nets means computing sums of products of numbers: sounds easy!!
- Problem: sums of *exponentially many* products!

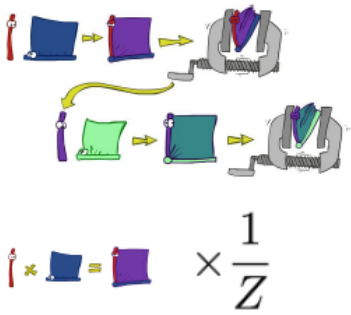
## Variable Elimination (VE)

Idea: Interleave joining and marginalizing!

Still NP-hard, but usually much faster than inference by enumeration.

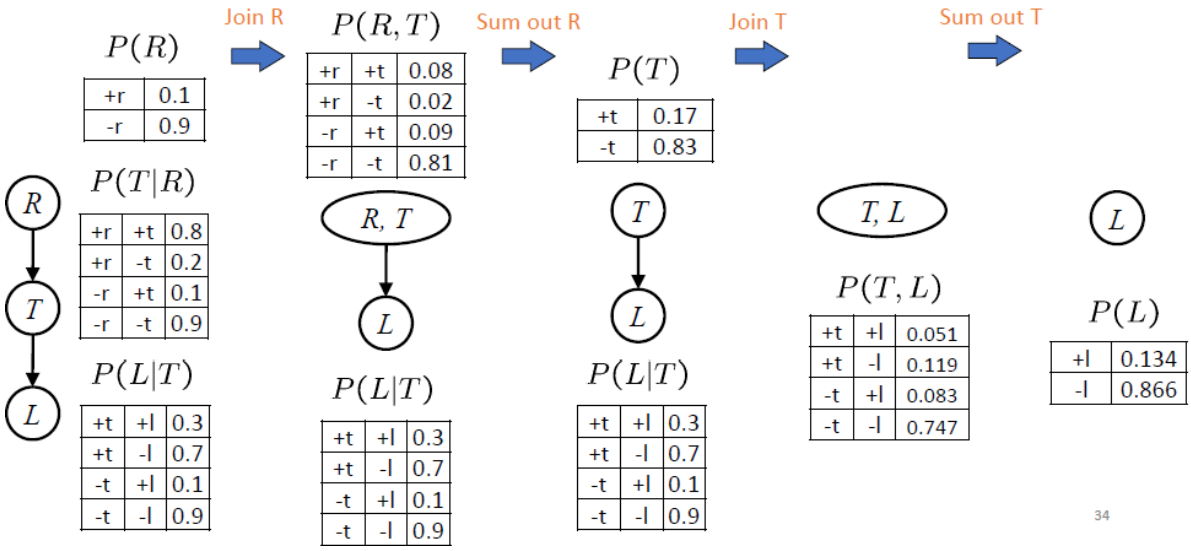
# General Variable Elimination

- Query: $P(Q|E_1 = e_1, \ldots E_k = e_k)$
- Start with initial factors:
  - Local CPTs (but instantiated by evidence)
- While there are still hidden variables (not Q or evidence):
  - Pick a hidden variable H
  - Join all factors mentioning H
  - Eliminate (sum out) H
- Join all remaining factors and normalize



$\times \frac{1}{Z}$

# Marginalizing Early! (aka VE)






# Another Variable Elimination Example

Query: $P(X_3|Y_1 = y_1, Y_2 = y_2, Y_3 = y_3)$

Start by inserting evidence, which gives the following initial factors:

$$P(Z), P(X_1|Z), P(X_2|Z), P(X_3|Z), P(y_1|X_1), P(y_2|X_2), P(y_3|X_3)$$

Eliminate $X_1$, this introduces the factor $f_1(y_1|Z) = \sum_{x_1} P(x_1|Z)P(y_1|x_1)$, and we are left with:

$$P(Z), P(X_2|Z), P(X_3|Z), P(y_2|X_2), P(y_3|X_3), f_1(y_1|Z)$$

Eliminate $X_2$, this introduces the factor $f_2(y_2|Z) = \sum_{x_2} P(x_2|Z)P(y_2|x_2)$, and we are left with:

$$P(Z), P(X_3|Z), P(y_3|X_3), f_1(y_1|Z), f_2(y_2|Z)$$

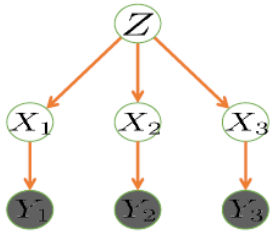
Eliminate $Z$, this introduces the factor $f_3(y_1, y_2, X_3) = \sum_z P(z)P(X_3|z)f_1(y_1|Z)f_2(y_2|Z)$, and we are left with:

$$P(y_3|X_3), f_3(y_1, y_2, X_3)$$

No hidden variables left. Join the remaining factors to get:

$$f_4(y_1, y_2, y_3, X_3) = P(y_3|X_3)\ f_3(y_1, y_2, X_3)$$

Normalizing over $X_3$ gives $P(X_3|y_1, y_2, y_3) = f_4(y_1, y_2, y_3, X_3)/\sum_{x_3} f_4(y_1, y_2, y_3, x_3)$

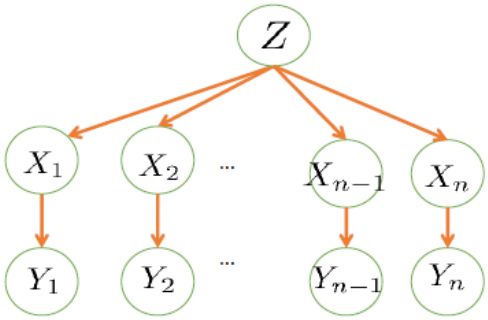



Computational complexity critically depends on the largest factor being generated in this process. Size of factor = number of entries in table. In example above (assuming binary) all factors generated are of size 2 --- as they all only have one variable (Z, Z, and $X_3$ respectively).

**Ordering**

# Variable Elimination Ordering

- For the query $P(X_n | y_1,\ldots,y_n)$ work through the following two different orderings as done in previous slide: Z, $X_1$, …, $X_{n-1}$ and $X_1$, …, $X_{n-1}$, Z. What is the size of the maximum factor generated for each of the orderings?




- Answer: $2^n$ versus 2 (assuming binary)
- In general: the ordering can greatly affect efficiency



# Worst Case Complexity?

- CSP:

$$(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3 \vee \neg x_4) \wedge (x_2 \vee \neg x_2 \vee x_4) \wedge (\neg x_3 \vee \neg x_4 \vee \neg x_5) \wedge (x_2 \vee x_5 \vee x_7) \wedge (x_4 \vee x_5 \vee x_6) \wedge (\neg x_5 \vee x_6 \vee \neg x_7) \wedge (\neg x_5 \vee \neg x_6 \vee x_7)$$

$P(X_i = 0) = P(X_i = 1) = 0.5$

$Y_1 = X_1 \vee X_2 \vee \neg X_3$

…

$Y_8 = \neg X_5 \vee X_6 \vee X_7$

$Y_{1,2} = Y_1 \wedge Y_2$

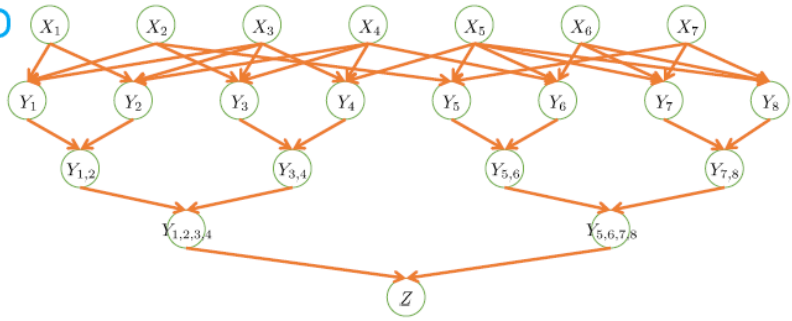
…

$Y_{7,8} = Y_7 \wedge Y_8$

$Y_{1,2,3,4} = Y_{1,2} \wedge Y_{3,4}$

$Y_{5,6,7,8} = Y_{5,6} \wedge Y_{7,8}$

$Z = Y_{1,2,3,4} \wedge Y_{5,6,7,8}$




- If we can answer P(z) equal to zero or not, we answered whether the 3-SAT problem has a solution
- Hence inference in Bayes' nets is NP-hard. No known efficient probabilistic inference in general



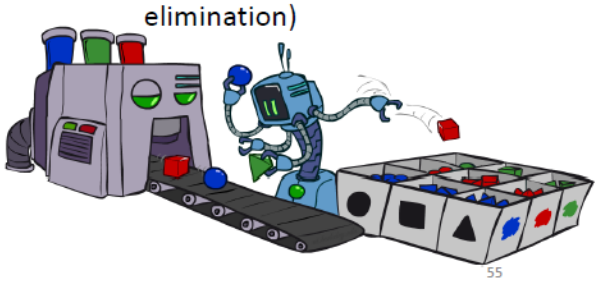
# "Easy" Structures: Polytrees

- A polytree is a directed graph with no undirected cycles
- For poly-trees you can always find an ordering that is efficient
  - Try it!!
- Cut-set conditioning for Bayes' net inference
  - Choose set of variables such that if removed only a polytree remains
  - Exercise: Think about how the specifics would work out!

## Sampling (*)

# Sampling

- Sampling is a lot like repeated simulation
  - Predicting the weather, basketball games, …
- Basic idea
  - Draw N samples from a sampling distribution S
  - Compute an approximate posterior probability
  - Show this converges to the true probability P
- Why sample?
  - Learning: get samples from a distribution you don't know
  - Inference: getting a sample is faster than computing the right answer (e.g. with variable elimination)

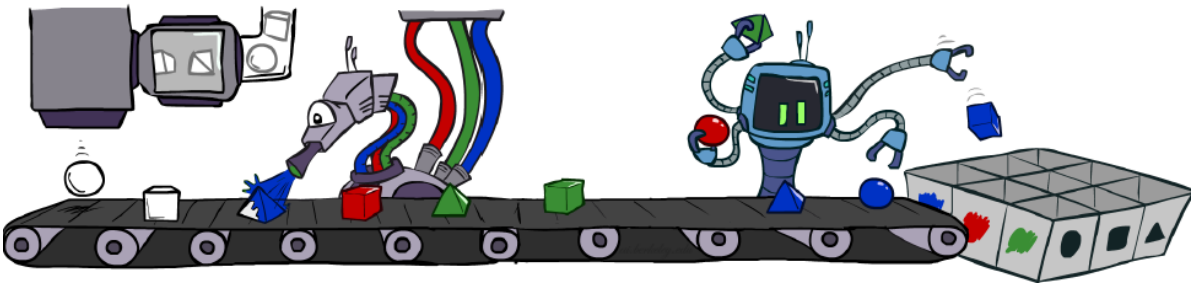


55

## Prior Sampling

# Prior Sampling: Algorithm

- For i = 1, 2, …, n in topological order
  - Sample $x_i$ from $P(X_i \mid \text{Parents}(X_i))$
- Return $(x_1, x_2, \ldots, x_n)$



# Prior Sampling

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

$$= S_{PS}(x_1, \ldots, x_n)$$

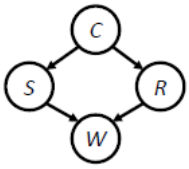
$$= P(x_1 \ldots x_n)$$

- i.e., the sampling procedure is consistent

## Rejection Sampling

# Rejection Sampling

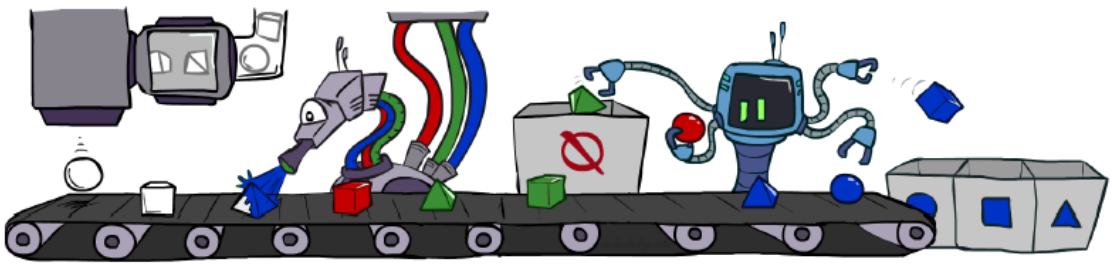
- Let's say we want P(C)
  - Just tally counts of C as we go




- Let's say we want P(C | +s)
  - Same thing: tally C outcomes, but ignore (reject) samples which don't have S=+s
  - This is called rejection sampling
  - We can toss out samples early!
  - It is also consistent for conditional probabilities (i.e., correct in the limit)

# Rejection Sampling: Algorithm

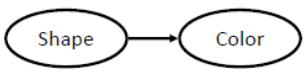
- Input: evidence instantiation
- For i = 1, 2, …, n in topological order
  - Sample $x_i$ from $P(X_i \mid Parents(X_i))$
  - If $x_i$ not consistent with evidence
    - Reject: return – no sample is generated in this cycle
- Return $(x_1, x_2, \ldots, x_n)$



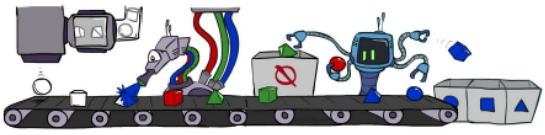
## Likelihood Reweighting

# Likelihood Weighting

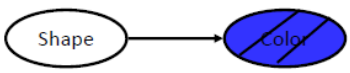
- Problem with rejection sampling:
  - If evidence is unlikely, rejects lots of samples
  - Consider P( Shape | blue )



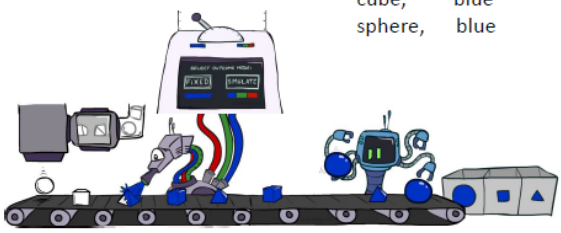

~~pyramid, green~~
~~pyramid, red~~
sphere, blue
~~cube, red~~
~~sphere, green~~



- Idea: fix evidence variables and sample the rest
  - Problem: sample distribution not consistent!
  - Solution: weight by probability of evidence given parents

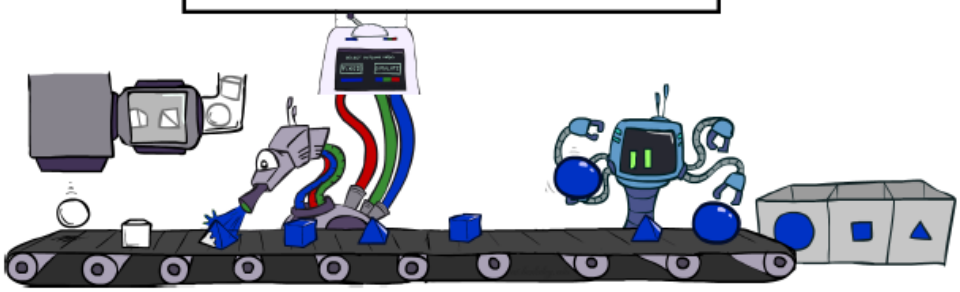



pyramid, blue
pyramid, blue
sphere, blue
cube, blue
sphere, blue



# Likelihood Weighting: Algorithm

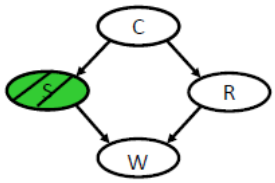
- Input: evidence instantiation
- w = 1.0
- for i = 1, 2, …, n in topological order
  - if $X_i$ is an evidence variable
    - $X_i$ = observation $x_i$ for $X_i$
    - Set w = w * $P(x_i \mid Parents(X_i))$
  - else
    - Sample $x_i$ from $P(X_i \mid Parents(X_i))$
- return $(x_1, x_2, \ldots, x_n)$,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



# Likelihood Weighting

- Sampling distribution if z sampled and e fixed evidence

$$S_{WS}(\mathbf{z}, \mathbf{e}) = \prod_{i=1}^{l} P(z_i|\text{Parents}(Z_i))$$




- Now, samples have weights

$$w(\mathbf{z}, \mathbf{e}) = \prod_{i=1}^{m} P(e_i|\text{Parents}(E_i))$$

- Together, weighted sampling distribution is consistent

$$S_{\text{WS}}(z, e) \cdot w(z, e) = \prod_{i=1}^{l} P(z_i|\text{Parents}(z_i)) \prod_{i=1}^{m} P(e_i|\text{Parents}(e_i))$$
$$= P(\mathbf{z}, \mathbf{e})$$

Likelihood weighting is helpful

- We have taken evidence into account as we generate the sample

  e.g. here, W's value will get picked based on the evidence values of S
- More of our samples will reflect the state of the world suggested by the evidence

Problem:

- Evidence influences the choice of <u>downstream</u> variables , but not <u>upstream</u> ones

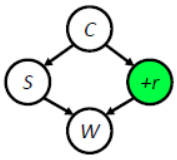
  e.g. C isn't more likely to get a value matching the evidence

## Gibbs Sampling

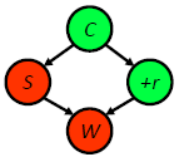
Gibbs sampling belongs to a family of sampling methods called Markov chain Monte Carlo (MCMC)

### Gibbs Sampling: Example P( S | +r)

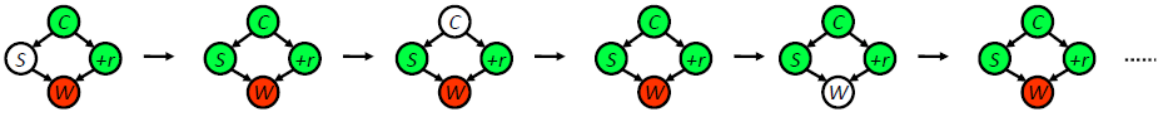
- Step 1: Fix evidence
  - R = +r




- Step 2: Initialize other variables
  - Randomly



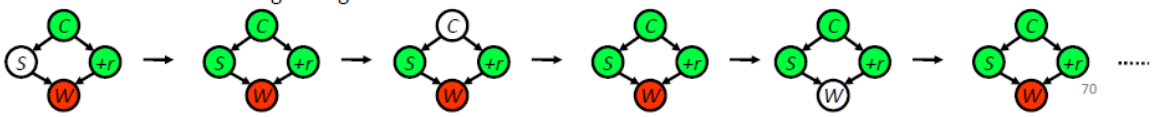

- Steps 3: Repeat
  - Choose a non-evidence variable X
  - Resample X from P( X | all other variables)*




Sample from $P(S|+c,-w,+r)$ Sample from $P(C|+s,-w,+r)$ Sample from $P(W|+s,+c,+r)$

### Gibbs Sampling

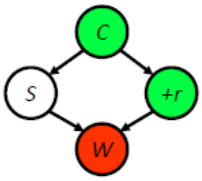
- Procedure
  - Keep track of a full instantiation $x_1, \ldots, x_n$
  - Start with an arbitrary instantiation consistent with the evidence
  - Sample one variable at a time, conditioned on all the rest, but keep evidence fixed
  - Keep repeating this for a long time
- Property
  - In the limit of repeating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




70

### Resampling of One Variable

- Sample from P(S | +c, +r, -w)




$$
\begin{aligned}
P(S|+c,+r,-w) &= \frac{P(S,+c,+r,-w)}{P(+c,+r,-w)} \\
&= \frac{P(S,+c,+r,-w)}{\sum_s P(s,+c,+r,-w)} \\
&= \frac{P(+c)P(S|+c)P(+r|+c)P(-w|S,+r)}{\sum_s P(+c)P(s|+c)P(+r|+c)P(-w|s,+r)} \\
&= \frac{P(+c)P(S|+c)P(+r|+c)P(-w|S,+r)}{P(+c)P(+r|+c)\sum_s P(s|+c)P(-w|s,+r)} \\
&= \frac{P(S|+c)P(-w|S,+r)}{\sum_s P(s|+c)P(-w|s,+r)}
\end{aligned}
$$

- Many things cancel out – only CPTs with S remain!
- More generally: only CPTs that have resampled variable need to be considered, and joined together

# Reference


1. https://shuaili8.github.io/Teaching/CS445/index.html
2. http://ai.berkeley.edu/lecture_slides.html
3. http://ai.berkeley.edu/exams.html
4. https://karpathy.github.io/2019/04/25/recipe/
5. https://ermongroup.github.io/cs228-notes/inference/sampling/
6. https://inst.eecs.berkeley.edu/~cs188/fa21/
7. https://inst.eecs.berkeley.edu/~cs188/fa19/final_prep/


# Demos

1. https://inst.eecs.berkeley.edu/~cs188/fa19/assets/demos/csp/csp_demos.html
2. http://playground.tensorflow.org/

# HW & Lab Recap

Refer to `homework solutions` for more details.

# Appendix: Matrix Gradient

1. https://zhuanlan.zhihu.com/p/24709748
2. https://zhuanlan.zhihu.com/p/263777564