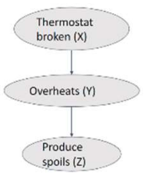
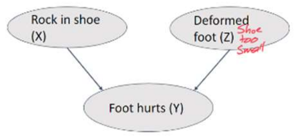
Bayes Nets:

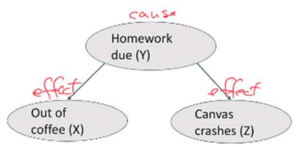
- Diagnostic: Observing an effect leads to competition between possible causes
- Causal Chain: What about X and Z, given Y?
  - Important bayes Net Question: Are two nodes independent given certain evidence?
    - If Yes – can prove using algebra
    - If No – can prove using a counterexample



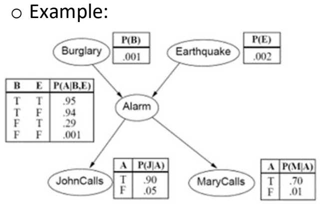
- Example:
  - Are X and Z necessarily independent?
  - No! X certainly influences Y, which influences Z
  - Also knowledge of Z influences beliefs about X (through Y)
  - This is a causal chain
- Common Effect: Where a single effect has two possible causes:



- Example:
  - Are X and Z independent?
  - Yes! But X and Z are not conditionally independent given Y.
- Common Cause: Another canonical case; Two effects, from the same cause



- Example:
  - Are X and Z independent?
  - Not necessarily
  - X and Z are not necessarily independent given Y
- Enumeration:
  - Query: What we want the posterior probability of given some evidence. $X = b$ (Burglery)
  - Observed Variables (Evidence): The variables we are given an assignment of (the data). $E = [+j, +m]$ (John and Mary calling)
  - Unobserved Variables (Hidden): The non-evidence, non-query variables. $y = [E, A]$ (Earthquake, and Alarm)
  - Model: Agent knows something about how the known variables relate to the unknown variables
  - Example:



- Suppose we know the alarm has gone off, and both John and Mary called to warn us. What is the probability that we have been burgled?
- $P(+b \mid +j, +m) = ?$
  - Calculation by **enumeration:**

$$P(B \mid j, m) = \frac{P(B, j, m)}{P(j, m)} \Rightarrow \alpha = \frac{1}{P(j, m)} \Rightarrow P(B \mid j, m) = \alpha P(B, j, m)$$

  - **With this in mind:**
    - $P(B \mid j, m) = \alpha P(B, j, m) = \alpha \sum_a P(B, j, m \mid a)\, P(a) = \alpha \sum_a P(B, j, m, a)$ **do this for all the hidden variables**
    - You should result in the following: $\alpha \sum_e \sum_a P(B, j, m, a, e)$ **See what we did there?**
    - Then we essentially take the conditional independence of the Bayes net which is this thing:

$$P(B \mid j, m) = \alpha \sum_e \sum_a \prod_{i=1}^{n} P(x_i \mid \text{parents}(X_i))$$
$$= \alpha \sum_e \sum_a P(B)P(e)P(a \mid B, e)P(j \mid a)P(m \mid a)$$
$$= \alpha P(B) \sum_e P(e) \sum_a P(a \mid B, e)P(j \mid a)P(m \mid a)$$

- Which we then apply to the example resulting in the following:
- $P(B \mid j, m) = \alpha P(B) \sum_e P(e) \sum_a P(a \mid B, e) P(j \mid a) P(m \mid a)$
- You then have to literally account for ALL possibilities:

P(B,J,M) =
.001 x .002 x .95 x .90 x .7 +
.001 x .998 x .94 x .9 x .7 +
.001 x .002 x .05 x .05 x .01 +
.001 x .998 x .06 x .05 x .01

- Then proceed to divide said properties by all the properties by $P(j = T, m = T)$
- You should result in the following:

$$P(Burglary = T \mid JohnCalls = T, MaryCalls = T) = \frac{P(Burglary = T, JohnCalls = T, MaryCalls = T)}{P(JohnCalls = T, MaryCalls = T)}$$

$$= \frac{P(Burglary = T, Earthquake = T \text{ or } F, Alarm = T \text{ or } F, JohnCalls = T, MaryCalls = T)}{P(Burglary = T \text{ or } F, Earthquake = T \text{ or } F, Alarm = T \text{ or } F, JohnCalls = T, MaryCalls = T)}$$

$$= \frac{P(+B,+E,+A,+J,+M) + P(+B,+E,-A,+J,+M) + P(+B,-E,+A,+J,+M) + P(+B,-E,-A,+J,+M)}{P(+B,+E,+A,+J,+M) + P(+B,+E,-A,+J,+M) + P(+B,-E,+A,+J,+M) + P(+B,-E,-A,+J,+M) + P(-B,+E,+A,+J,+M) + P(-B,+E,-A,+J,+M) + P(-B,-E,+A,+J,+M) + P(-B,-E,-A,+J,+M)}$$

$$= \frac{(0.001)(0.002)(0.95)(0.90)(0.70) + (0.001)(0.002)(1-0.95)(0.05)(0.01) + (0.001)(1-0.002)(0.94)(0.90)(0.70) + (0.001)(1-0.002)(1-0.94)(0.05)(0.01)}{\ldots + (1-0.001)(0.002)(0.29)(0.90)(0.70) + (1-0.001)(0.002)(1-0.29)(0.05)(0.01) + (1-0.001)(1-0.002)(0.001)(0.90)(0.70) + (1-0.001)(1-0.002)(1-0.001)(0.05)(0.01)}$$

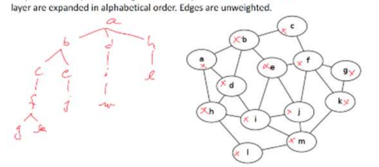
$$= \frac{0.0000012 + 0.000000001 + 0.0005910156 + 0.00000002994}{0.0000012 + 0.000000001 + 0.0005910156 + 0.00000002994 + \ldots} = \frac{0.00059224654}{[illegible]} = 0.284173$$

- Some enumeration examples:
  - $P(+b \mid +a) =$
    - Calculate P(a) first: $(.001 * .002 * .95) + (.001 * .998 * .94) + (.999 * .002 * .29) + (.999 * .998 * .001) = 0.0025$
    - Top: $(.001 * .95 * .002) + (.001 * .94 * .998) = .00094$
    - $0.00094/.0025 = .376$
- Sampling:
  - For large BN, computationally intractable to calculate exact joint distributions
    - Generate random sample u from uniform distance [0, 1)
    - Convert sample to an outcome for a given distribution
  - Example: A very important survey was given out to several hundred people walking along Pearl Street asking what their childhood pet was. Below is the resulting probability distro of that survey: (Cat: .28, Dog: .37, Pig: 0.11, Fish: 0.18, Rock 0.06)
    - Generate a random list, in this case is given: [0.11, 0.97, 0.45, 0.16, 0.37, 0.76, 0.55, 0.42, 0.99, 0.84, 0.02, 0.29, 0.63, 0.69, 0.18]
    - We then take that random list and map each random value to the designated animal that it is associated with:
      - Cat [0, .27], Dog [28, 64], Pig [65, 76], Fish [76, 94], Rock [94, 100]
    - Then we calculate the sample distribution. Taking the length of each list (Cat, Dog, Pig, etc.) and dividing it by the total of the animal's probabilities in this case is 100 giving each sample distribution on each value.

BFS, DFS, UCS

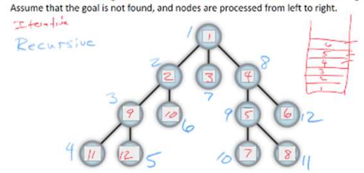
- Agents: Actions, percepts (e.g. robot vacuum)
- States: Discrete configuration of the environment (position of agents, dirty and clean cells)
- Search Algorithms: Find problem solution – sequence of actions to go from current state to goal state
  - A search problem consists of: State space, Transition Model, Actions, Initial State, Goal test, Solution
- Uninformed Search: No additional information about states beyond that in the problem definition
- Informed Search: Some idea of which non-goal states are "more promising" than others.
- Breadth First Search: Search across the tree before searching deeper into the tree. **BFS doesn't work on weighted graphs**
- Depth First Search: Search deeper into the tree before searching across. Implementation determines nodes explored; iterative and recursive versions exist.
- Uniform Cost Search: BFS strategy with additional logic
- A*: BFS strategy, informed, heuristic to identify more promising solutions
- Things to think about:
  - Completeness: Algorithm will find a solution if it exists
  - Optimality: Algorithm will find optimal solution
  - Time Complexity: # of operation completed in worst case
  - Space complexity: Memory needs
- BFS Example:

Example: Build a search tree from the nodes in the graph according to the order in which they would be expanded using BFS to find a path from a to k. Assume that nodes within a layer are expanded in alphabetical order. Edges are unweighted.



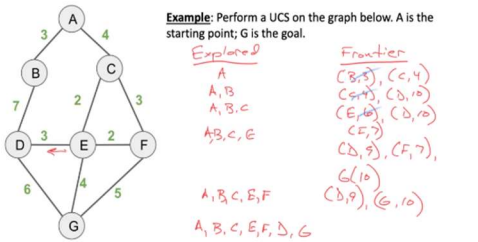
  - Complete: Yes
  - Optimal: Yes in unweighted graph, no otherwise
  - Time Complexity: $O(b^d)$
  - Space Complexity: Assumes need to store every node in the explored set: $O(b^{d-1})$ and every node on the frontier: $O(b^d)$ so $O(b^d)$
  - Keep in mind that b is the branching factor and d is total layer
- DFS Example:

Example: Number the nodes in the search tree according to the order in which they would be added to visited using DFS. Show both iterative and recursive versions of the algorithm. Assume that the goal is not found, and nodes are processed from left to right.



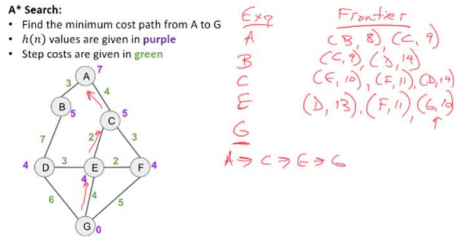
  - Complete: Yes if the solution exists
  - Optimal: No, not even in unweighted graphs
  - Time complexity: Worst case scenario: $O(b^m)$
  - Space complexity: Worst case scenario: $O(mb)$, best case $O(b^m)$. m is the maximal depth of m layers. Potential failure in infinite state spaces
- Uniform Cost Search Example:
  - Expand out in contours, where least cost dictates which nodes we explore.
  - Eventually we will find a path to the goal – but the search is not directed.

- BFS Strategy
- Expand cheapest node first (lowest path cost)
- Frontier is a priority queue
- Cost function sets priority

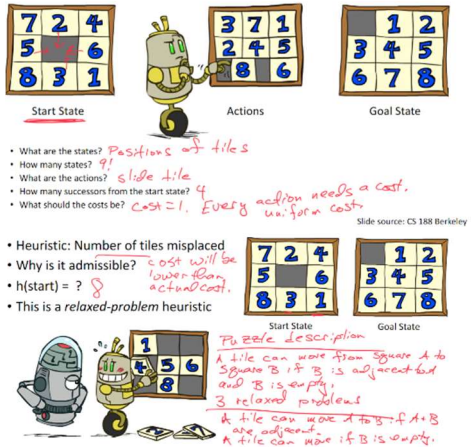


- Goal test occurs when node is selected for expansion
- Because se know we've taken the cheapest path to get there, UCS is optimal if all edge weights > 0
- Also complete because it's a more general form of BFS (which is complete)
- Can get stuck if there are sequences of no-cost actions. Optimality requires positive edge weights
- Worst case in time and space complexity: $O(b^{1+\lfloor \frac{C*}{\epsilon} \rfloor})$
  - C* is cost of optimal solution and € is minimal action cost
- Potential inefficiency: Explores in every direction

• A* Example:
- Key Difference:
  - Uniform Cost Search: $f(n) = g(n)$ (cost to get to n)
  - Greedy: $f(n) = h(n)$ (estimated cost to get from n to goal)
  - A*: $f(n) = g(n) + h(n)$ (estimated cost of cheapest sol.)
- Consistent: For every node n and successor n' of n, generated by some action a, the estimated cost of reaching the goal from n is no greater than the step cost from n to n', plus the estimated cost of reaching the goal from n'
- A heuristic is **admissible** – an admissible heuristic is one that never overestimates the cost to reach the goal
- The heuristic is **consistent** if, for every node n and every successor n' or n generated by any action a, the estimated cost of reaching the goal from n is no greater than the step cost of getting to n' plus the estimated cost of reaching the goal from n': $h(n) \leq c(n, a, n') + h(n')$
- A* is **optimally efficient** for any given heuristic: No other optimal algorithm is guaranteed to expand fewer nodes than A*
  - Recall: A* expands all nodes with $f(n) < C*$, where C* is the cost of the optimal solution path
  - Any algorithm that does not expand all nodes with this risks missing a better solution path
- A* (graph) is optimal if the heuristic h(n) is consistent
  - If h(n) is consistent, then the values of f(n) along any path are nondecreasing
  - Whenever A* selects a node n for expansion, the optimal path to that node has been found
- If it's so great why do we use other search algos?
  - Number of nodes to expand along the goal contour is still exponential in depth of solution/length of solution path
  - It only also works only when domain is fully observable, known, deterministic, and static



• Heuristics:
- Educated guess about solution quality using domain knowledge
- Using a heuristic can help solve a problem more quickly
- We want the heuristic to be admissible, but we need to keep in mind that the lower heuristic is, the more nodes A* expands making the algo slower in the process
- We come with heuristics using the following:
  - Generate heuristic from relaxed problems (same problem but with fewer constraints
  - Generate heuristics from sub-problems
  - Learning heuristics from experience (learn through multiple trials usually thousands)
- Class example (8 tile problem):



- Class example(Checkers/Chess):
  - Are there relaxed problems in the game that could be used in generating heuristics?:
    - Chess: Knight moves in L, King has to move out check. One piece moves per player per turn, King can be attacked, minimum # of pieces must present, etc.

• Local Search:
- Use only the current node/state and consider only moves to neighbor states. Super memory efficient (only cares about current state). Used widely for optimization and find the best state according to some objective function
- Global Maximum: Refers to the highest possible value that a function can attain
- Local maximum: The highest value that a function takes on within a specific range of its domain
- Random Restarts: Used to avoid getting stuck in local optima

• Minimax with Alpha-Beta Pruning: Alpha = Max, Beta = Min
- Step 1: Alternate between Max and Min (MAX ALWAYS STARTS FIRST)
- Step 2: Initialize alpha and beta as empty values. Technically they are negative and positive INFTY, but for the sake of simplicity let's say they are empty.
- Step 3: Traverse into the inner most left leaf and compare the values based on whether it is a MAX or MIN. Since value, alpha, and beta are all empty. Simply place the inner most left child into the value slot.
- Step 4: The next thing we do is determine whether or not the value that we've just found is lesser/greater than beta/alpha. Pass Beta or Alpha along to the parent node.
  - IF MIN: Lesser than and compute for BETA
  - IF MAX: Greater than and compute for ALPHA
- Step 5: Once you are at the parent node, evaluate Alpha or Beta again, and pass that value along to the next child or the parent's parent.
- Step 6: We should now have an alpha and a beta when we evaluate the next child. Evaluate the following:
  - IF MAX: Is the value of the child you are currently evaluating, greater than or equal to the beta?
    - IF IT IS: There's no need to evaluate further, because we know the parent is gonna evaluate the other node anyways, so there's no need to evaluate any further
  - IF MIN: Is the value of the child you are currently evaluating, less than or equal to the alpha?
    - The same is true for MIN. Just prune it with ALPHA.

• The steps are essentially the same now. Wash rinse repeat until you reach the desired outcome with MiniMax.