Lecture 1, Jan 9, 2024

Rational Agents

- An *agent* is anything that perceives the *environment* through *sensors* and acts upon the environment through *actuators*; e.g. humans and robots are both agents
  - The *agent function* maps from percept histories/sequences to actions: $f : P^* \mapsto A$
  - The agent *program* runs on the physical *architecture* to perform f
  - The *transition model* is a function $s' = T(s, a)$ that maps the current state and an action to the next state
  - Example: for a robot that mops the room:
    * Environment: the location of the robot and the status of each location (clean and dirty)
    * Percept: current location and the status of the location
    * Actions: move around and mop the current location
    * The agent's function would map sequences of percepts to actions, as in the figure below

Percepts Sequence	Action_1	Action_2
[A, clean]	Right	Mop
[A, dirty]	Clean	Right
[B, clean]	Left	Mop
[B, dirty]	Clean	Left
[A, clean],[A, clean]	Right	Clean
[A, clean],[A, dirty]	Clean	Right

Figure 1: Example agent function for the mop robot.

- A *rational* agent is an agent that does the "right thing", based on all the information it has access to
  - Rational agents are not omniscient; the "right thing" is conditioned on the information and resources the agent can access
  - To define this, we define some *performance measure*, an objective criterion for measuring the success of the agent's behaviour
- Properties of task environments:
  - A task is *fully observable* if sensors provide access to the complete state of the environment at all times; otherwise it is *partially observable*
  - A task is *deterministic* if the next state of the environment is completely determined by the current state and the agent's action; otherwise it is *stochastic*
  - A task is *dynamic* if the environment can change while the agent is deliberating; otherwise it is *dynamic*
  - A task is *discrete* if the number of states, percepts, actions is finite; otherwise it is *continuous*
  - A task is *single-agent* if the agent operates by itself; otherwise it is *multi-agent*
- Types of agent programs:
  - *Simple reflex*: actions only depend on percept
  - *Model-based reflex*: action depend on internal state (based on percept history), model of the world, and percept
  - *Goal-based*: action depends on current state, percepts, model of the world, and tries to achieve a desired goal
  - *Utility-based*: tries to achieve multiple conflicting goals; uses a weighted combination of goals

Goal-Based Agents

- Example: finding the shortest route between two locations
  - States: locations
  - Actions: moving between locations

  - Transition model: taking current location and direction that we move in, outputting the new location
  - Goal test: whether we are at the location we want to go
  - Cost function: length of route
  - In general, we want to keep the action simple, and restrict what we can do in the transition model

Search Algorithms

- Many problems can be modelled as having an initial node, a successor function $S(x)$ giving the set of new nodes from a node x in a single action, the goal test function $G(x)$, and the action cost function $C(x; a; y)$ giving the cost of moving from x to y using action a
  - A state represents a physical configuration, while a node is a part of a search tree
  - Each node includes the state, parent node, action, and path cost
  - Two different nodes are allowed to represent the same state!
  - In most settings, representing the entire graph in memory is impractical, so implicit representations that only keep a part of the graph at a time are used
- To solve this problem, we can start at the initial node and keep searching until we reach a goal node
  - The *frontier* is the set of all nodes that we have seen but haven't explored
    - At initialization this is just the initial node
  - At each iteration we can choose a node from the frontier, explore it, and add its neighbours to the frontier
  - *Tree search algorithm* don't store information about visited states, so can end up in cycles
  - *Graph search algorithms* keep track of visited nodes so explored nodes are not revisited (aka cycle checking)
- How do we evaluate an algorithm?
  - *Completeness*: whether the algorithm always finds a solution, if one exists
  - *Optimality*: is the solution least-cost?
  - *Time complexity*: how long does it take to find a solution?
  - *Space complexity*: how many nodes do we need to store in memory?
- To quantify the problem we use the following parameters:
  - Branch width b: maximum number of successors on each node
    - Unless otherwise stated, assuming that this is finite, i.e. at every state there are a finite number of states we can go to
  - Depth d: depth of shallowest goal node
    - This is usually finite
  - Max depth m: max depth of any node from the start node
    - This is often infinite (but countably infinite) – e.g. if the workspace of the robot is the entirety of Mars
  - We don't know the number of nodes in advance so instead of we use these parameters, since they are local properties
- Note worst-case scenario analysis does not capture the graph structure; performance in the real world is often highly problem-dependent, so the best algorithm will also be
- *Uninformed* search algorithms use only the problem input; no domain information is used
  - The problem is represented either explicitly or implicitly as graphs
  - Includes BFS, DFS, uniform-cost search

Breadth First Search

- Explores nodes in order of their discovery, using a FIFO queue
- Completeness: yes; even for infinitely large graphs, as long as b and d are finite, the goal will eventually be reached
- Time complexity: $O(b^{d+1})$
  - At each node we explore at most b new nodes
- Space complexity: $O(b^{d+1})$

Algorithm 1 Breadth First Search: FindPathToGoal(u)

```
1. F(Frontier)← Queue(u)                    // FIFO
2. E(Explored)← u
3. while F is not empty do
4.     u ← F.pop()
5.     for all children v of u do
6.         if GoalTest(v) then
7.             return path(v)
8.         else
9.             if v ∉ E then
10.                E.add(v)
11.                F.push(v)
12. return Failure
```

Figure 2: BFS algorithm.

  - This is the max size of the frontier
- Optimality: no; the result is not optimal in cost, but it is optimal in the number of state transitions
  - Note simply replacing the queue by a priority queue based on cost would not work by itself since the algorithm still returns too early and does not update node costs
  - Making the appropriate modifications, we have uniform cost search

Uniform Cost Search

Algorithm 3 Uniform Cost Search(UCS): FindPathToGoal(u)

```
1. F(Frontier)← PriorityQueue(u)            // lowest cost node out first
2. E(Explored)← u
3. ĝ[u] ← 0
4. while F is not empty do
5.     u ← F.pop()
6.     if GoalTest(u) then
7.         return path(u)
8.     E.add(u)
9.     for all children v of u do
10.        if v ∉ E then
11.            if v ∈ F then
12.                ĝ[v] = min(ĝ[v], ĝ[u] + c(u, v))
13.            else
14.                F.push(v)
15.                ĝ[v] = ĝ[u] + c(u, v)
16. return Failure
```

Figure 3: UCS algorithm.

- Explores nodes in order of cost
- Completeness: yes, if b is finite and if all edge weights are greater than equal to some positive ϵ
- Optimality: yes; every time we pop some node u, we can guarantee that the path we found to u is optimal, provided weights are positive
  - This can be proven by induction
- Time complexity: $O(b^{1+\frac{C^*}{\epsilon}})$ where C^* is the optimal cost
  - Since the optimal cost is C^* and the cost so far increases by at least ϵ at every step, we take at

most $\frac{C^*}{\epsilon} + 1$ steps

- Space complexity: $O(b^{1+\frac{C^*}{\epsilon}})$ by the same logic

Depth First Search

Algorithm 4 Depth First Search(DFS)-TreeSearch: FindPathToGoal(u)

```
1. F(Frontier)← Stack(u)
2. while F is not empty do
3.     u ← F.pop()
4.     if GoalTest(u) then
5.         return path(u)
6.     for all children v of u do
7.         F.push(v)
8. return Failure
```

Figure 4: DFS tree search algorithm.

- Explores the deepest discovered but unexplored node first
- To minimize memory usage, we don't store the set of explored nodes (i.e. use a tree search)
- Completeness: only if the search space is finite
- Optimality: no
- Time complexity: $O(b^{m+1})$
- Space complexity: $O(bm)$
- How do we remedy the loss of completeness?
  - Depth limited search (DLS): restricting the depth of the search to a certain cutoff
    - This is complete only if d is less than or equal to the cutoff
    - But we don't know d beforehand
  - Iterative deepening search (IDS): iteratively increasing the cutoff depth, if a complete search is performed and no goal is found

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

1. if b is finite.
2. If b is finite and step cost $\geq \varepsilon$

Figure 5: Summary of search algorithms.

Lecture 2, Jan 16, 2024

Informed Algorithms – A^* Search Algorithm

- Suppose that we have some kind of knowledge about the system, in the form of an evaluation/heuristic function $h(u)$ which gives an estimate of the distance of the goal from u; we can use this in UCS to speed up the algorithm
  - In the priority queue, instead of using $\hat{g}(u)$ as the pop order, use $\hat{f}(u) = \hat{g}(u) + h(u)$
  - This will help the algorithm always choose nodes in the direction of the goal and not explore unnecessary nodes
- This modified algorithm is the A^* search algorithm

Algorithm 6 A* Algorithm: FindPathToGoal(u)

1. F(Frontier) ← PriorityQueue(u) // This should be implemented with $\hat{f}$ minimum
2. E(Explored) ← $\{u\}$
3. Initialize $\hat{g}$: $\hat{g}[u] \leftarrow 0$ $\hat{g}[v] = \infty$. $\forall v \neq u$
4. **while** F is not empty **do**
5. $v \leftarrow F$.pop()
6. **if** GoalTest (v) **then**
7. **return** path (v)
8. E.add (u)
9. **for** all successors v of u **do**
10. **if** v not in E **then**
11. **if** v in F **then**
12. $\hat{g}[v] = min(\hat{g}[v], \hat{g}[u] + c(u, v))$
13. $\hat{f}[v] = h[v] + \hat{g}[v]$
14. **else**
15. F.push(v)
16. $\hat{g}[v] = \hat{g}[u] + c(u, v)$
17. $\hat{f}[v] = h[v] + \hat{g}[v]$
18. **return** Failure

Figure 6: A^* search algorithm.

- Is this still optimal?
  - In our proof of optimality, we relied on having $g(u) \leq \cdots \leq g(v_{k-1}) \leq g(v_k) \leq g(w)$ for a path to the goal of $u, v_1, \ldots, v_k, w$
  - We need to show that if $i < j$, then $f(v_i) \leq f(v_j) \iff g(v_i) + h(v_i) \leq g(v_j) + h(v_j)$
  - Note $c(v_i, v_j) \geq g(v_j) - g(v_i)$ where c is the cost
    - This means a sufficient condition is $h(v_i) \leq h(v_j) + c(v_i, v_j)$
- This property is known as *consistency*: $\forall v_i, v_j, h(v_i) \leq h(v_j) + c(v_i, v_j)$ where v_j is a successor of v_i
  - If h is consistent, then A^* is optimal; we can prove this in the same way that we proved UCS optimal
- h is *admissible* if it satisfies $\forall v_i, h(v_i) \leq C^*(v_i)$ where $C^*(v) = g(w) - g(v)$ which is the true (optimal) cost to reach the goal from v
  - Theorem: if h is admissible, then A^* with tree search (i.e. no cycle checking) is optimal
    - Note this only works with tree search because of it possibly searching the same node multiple times
    - If we do cycle checking, then if we popped a node in the wrong order, then we will never take the correct path to it again
- Admissibility is generally a weaker property than consistency; if $h(w) = 0$, then consistency implies admissibility

Definition

Consistency: The evaluation function h is *consistent* if it satisfies

$$\forall v_i, v_j, h(v_i) \leq h(v_j) + c(v_i, v_j)$$

where v_j is a successor of v_i and $c(v_i, v_j)$ is the cost of moving from v_i to v_j, i.e. for each step we move backwards, the value of the heuristic at the further node is less than or equal to the value at the closer node plus the cost of the step.

Definition

Admissibility: The evaluation function h is *admissible* if it satisfies

$$\forall v_i, h(v_i) \leq C^*(v_i) = g(w) - g(v_i)$$

where $C^*(v)$ is the true optimal cost to reach the goal from v, i.e. the value of the heuristic is always less than or equal to the true cost of reaching the goal.

- Now we have requirements for h, how do we choose one?
  - Euclidean distance (L2 norm) is one option
    - Since it satisfies the triangle inequality, it is admissible and consistent
  - Manhattan distance (L1 norm) can be used in the case of certain movement restrictions
    - This is admissible with restrictions that we can only move along the axes
- A problem with fewer restrictions on the actions is a *relaxed problem*
  - The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem
  - We can always ignore certain constraints so that the problem is easily solvable, and use this as the heuristic
  - Example: the 8-puzzle
    - Tiles can only move into neighbouring tiles if that tile is empty – this is the restriction that makes it hard
    - In one relaxed problem we can move a tile to an adjacent tile always
      - We can select $h_1(n)$ to be the number of misplaced tiles
    - In another relaxed problem we can move any tile to any other tile
      - We can select $h_2(n)$ to be the total Manhattan distance between the current configuration and the desired configuration
- Unfortunately, there is no recipe to generate consistent heuristics
  - This is one reason to prefer tree search
- What if the priority queue is based on h instead of f?
  - This is called greedy best-first search and it is no longer optimal
  - However, this is faster

Lecture 3, Jan 23, 2024

Local Search and Optimization

- So far we have looked at problems where we want to find the minimum cost path to the goal; the goal itself may be known and the path is the desired solution
- In some situations the path we take is irrelevant, and we just want to find the goal
- Example: the N-queens problem: place N chess queens on an $N \times N$ chessboard such that no two queens can attack each other (i.e. no two queens share the same row, column, or diagonal)
- Every column must have exactly one queen, so we place one queen in each column and only move queens along columns

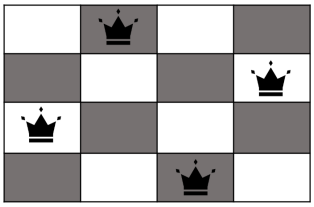

Figure 7: Solution to the N-queens problem for $N = 4$.

  - In general, we start from some random position and try to move to a better position
- For all such problems we define the following:
  - S: set of all states
  - $N(s)$: neighbours of the state $s \in S$ (i.e. all states reachable from s in one step)
  - Val(s): value of the state $s \in S$
    * This should reflect the "quality" of the state, i.e. how close it is to the goal
    * For the N-queens problem, this could be the number of pairs of queens that can attack each other
    * We want Val(w) = 0 where w is the goal, so the problem becomes minimizing Val(s) until we reach 0

Hill Climb Algorithm

- A simple strategy would be to always take steps that improve the value of the state in hopes of eventually reaching the goal; this leads to the *hill climb* algorithm
  - This is a type of *local search*, since at each step we aim to improve the local situation we're in

Algorithm 1 $HillClimb(S)$

```
1. minVal ← val(S)
2. minState ← {}                    //Only 1 thing to track
3. for each u in N(S) do
4.     if val(u) < minVal then
5.         minVal = val(u)
6.         minState = u
7. return minState
```

Algorithm 2 $SolveNQueens(initialState)$

```
1. S ← initialState
2. while val(S)! = 0 do
3.     S ← HillClimb(S)
4. return S
```

Figure 8: Hill climb algorithm.

- The hill climb algorithm is very simple and uses a minimal amount of memory since it only keeps track of the current state
  - However, hill climb is susceptible to getting stuck in local minima
  - Since it only allows moves to better positions, if the goal is locked behind a worse position, it will never be reached
  - We want an algorithm that allows making a "mistake" (moving to a state with higher value) but still stays mostly on track to the goal

Simulated Annealing

- We can use the *simulated annealing* algorithm, where transitions to states that raise the value are allowed, and the probability of such transitions is dependent on the difference in value

Algorithm 3 SimulatedAnnealing($initialState$)

1. $C \leftarrow initialState$
2. **for** $t = 0$ to ∞ **do**
3. $\quad C' \leftarrow PickRandomNeighbour(C)$
4. $\quad T \leftarrow Schedule(t)$ //Assume that $val(Goal) = 0$
5. $\quad$ **if** $val(C') = 0$ **then**
6. $\quad\quad$ **return** C'
7. $\quad$ **if** $val(C') < val(C)$ **then**
8. $\quad\quad C \leftarrow C'$
9. $\quad$ **else**
10. $\quad\quad C \leftarrow C'$ with Probability $\propto exp\left(\frac{val(C)-val(C')}{K_B T}\right)$

Figure 9: Simulated annealing algorithm.

- At each step, we pick a random neighbour C' and look at its value; if the value is lower, then the transition is always allowed; if the value is higher, then the transition is allowed with probability $e^{-\frac{\text{Val}(C')-\text{Val}(C)}{kT}}$
  - This is inspired by the annealing process in material physics
  - Transitions to states that raise the value are allowed, but the more the value is raised, the less likely the transition is to occur
  - T is a function of time, known as the *cooling schedule*, typically a decreasing function
    * Initially, the "temperature" is high, so the probability remains high regardless of the value difference, so the state can freely jump around
    * As time goes on we lower the temperature, making transitions to worse states increasingly unlikely
    * The cooling schedule is application dependent
  - The algorithm terminates when we reach $\text{Val}(C) = 0$; for some problems the value of the optimum might not be known, in which case we terminate when $T = 0$
- Simulated annealing does not always reach the solution (i.e. it is incomplete), but it is often effective for a variety of problems

Lecture 4, Jan 30, 2024

Constraint Satisfaction Problems (CSPs)

Definition

Constraint Satisfaction Problem: A *CSP* comprises of 3 components:
1. A set of variables $X = \{ x_1, x_2, \ldots, x_n \}$
2. A set of domains for each variable: D: $\{ D_{x_1}, D_{x_2}, \ldots, D_{x_3} \}$
3. A set of constraints C relating the variables

The problem is to find a value for each of the variables in its domain that satisfies all the constraints.

- Example: For the 4-queens problem:
  - Variables: $\{ x_1, x_2, x_3, x_4 \}$
  - Domain: for each variable: $D_{x_i} = \{ 1, 2, 3, 4 \}$

  - Constraint: NoAttack(x_i, x_j) (true if queen x_i can attack x_j)
    * We can express this in a table, giving the value of NoAttack for every combination of x_i, x_j
- In general CSPs are NP-hard – no polynomial time solution exists
- But we can use heuristics to do better in a lot of the problems that arise in real life

Backtracking Search

- Assign each of the variables some value, and then check if the constraints are satisfied
- If the constraints are not satisfied, revert the last variable that we assigned – this is the process of *backtracking*
- If no value of the last variable works, then we go back one more variable and pick another value for that one, and so on
- This is essentially a brute force search if we pick values for the variables sequentially
  - However we can also use heuristics to aid our search

Algorithm 1 BacktrackingSearch(*prob,assign*)

```
1. if AllVarsAssigned(prob,assign) then
2.   if IsConsistent(assign) then
3.     return assign
4.   else
5.     return failure
6. var←PickUnassignedVar(prob,assign)
7. for value∈ OrderDomainValue(var,prob,assign) do
8.   assign← assign ∪ (var = value)
9.   result ← BacktrackingSearch(prob,assign)
10.  if result != failure then return result
11.  assign ← assign \ (var=value)
12. return failure
```

Figure 10: Basic backtracking search algorithm.

- The pseudocode above is a template for the backtracking search algorithm
  - Each level of the recursion picks a variable to set, goes through all values of that variable and checks if any of them work
  - We can specify different implementations for PickUnassignedVariable and OrderDomain
- One simple improvement we can make is to only assign variables to values that satisfy all the constraints
  - Otherwise we would do a lot of meaningless searches when we pick a value that violates a constraint and keeps assigning others
  - e.g. for N-queens, check that the new queen cannot attack any previous queens before placing it
- Can we do better and reduce our backtracks even more?
  - Every time we assign a variable, it reduces the domain that the other variables can take based on constraints
  - The domain reduction (restrictions) on the other variables can propagate to even more variables
  - Every time we do an assignment, we call the inference function, which restricts the domain further based on the constraints and the new variable value
    * This can look at multiple constraints at the same time but often we stick to just 1

Algorithm 2 BacktrackingSearch(*prob*,*assign*)

```
1. if AllVarsAssigned(prob,assign) then
2. var←PickUnassignedVar(prob,assign)
3. for value∈ OrderDomainValue(var,prob,assign) do
4.     if ValIsConsistentWithAssignment(value,assign) then
5.         assign← assign ∪ (var = value)
6.         result ← BacktrackingSearch(prob,assign)
7.         if result != failure then return result
8.         assign ← assign \ (var=value)
9. return failure
```

Figure 11: Improved backtracking search algorithm.

Algorithm 3 BacktrackingSearch_with_Inference(*prob*,*assign*)

```
1. if AllVarsAssigned(prob,assign) then
2. var←PickUnassignedVar(prob,assign)
3. for value∈ OrderDomainValue(var,prob,assign) do
4.     if ValIsConsistentWithAssignment(value,assign) then
5.         assign← assign ∪ (var = value)
6.         inference← Infer(var,prob,assign)
7.         if inference != failure then
8.             assign← assign ∪ inference
9.             result ← BacktrackingSearch(prob,assign)
10.            if result != failure then return result
11.        assign ← assign \ {(var=value) ∪ inference
12. return Failure
```

Figure 12: Improved backtracking with inference.

Lecture 5, Feb 6, 2024

Inference and Heuristics for CSPs

- What data structures should we use for infer and assign in the algorithm?
  - For the inference use unordered list of tuples of the form $(x \notin S)$
  - For the assignments use unordered list of tuples of the form $(x = v)$ or $(x \notin S)$
- The ComputeDomain$(x, \text{assign}, \text{inference})$ returns a set S such that the effective domain of x is S, i.e. x can only be in S
  - This just simply looks at whether x has already been assigned a value (if so, return that value), otherwise it looks at the inference for x and gives all values that x is not forbidden to have
- The basic idea of infer is: for a certain value of the current variable, compute the effective domains of all other variables that would satisfy all constraints relating to this variable, and if any of the domains are empty, then the current variable cannot be this value
  - We need to go over each constraint where the current variable appears to limit the effective domains of all other variables
  - If the effective domain of any variable becomes empty, fail immediately
  - Every single time the effective domain of any variable changes, we need to examine all other variables and restrict their domains based on constraints relating to the variable whose domain changed

Algorithm 4 **Infer** (prob, var, assign) function of Algorithm 3

```
1. inference ← ∅
2. varQueue ← [var]
3. while varQueue is not empty do
4.     y ← varQueue.pop()
5.     for each constraint C ∈ prob where y ∈ Vars(C) do
6.         for all x ∈ Vars(C) \ y do
7.             S ← ComputeDomain (x, assign, inference)
8.             for each value v ∈ S do
9.                 if no valid value exists ∀var ∈ Vars(C) \ x s.t. C[x = v] is satisfied then
10.                    inference ← inference ∪ (x ∉ v)
11.            T ← ComputeDomain (x, assign, inference)
12.            if T = ∅ then return failure
13.            if S ≠ T then varQueue. add(x)
14. return inference
```

Figure 13: The infer algorithm.

- There are many places in the above algorithm where we can insert heuristics etc to improve the runtime
- Infer is very expensive to run, so we might want to limit the depth of the inference to reduce the overall computational cost
- In *forward checking*, we only check the effect of assigning the current variable
  - Delete line 13/14 so we don't add anything to the queue
  - For many problems, this is not effective enough – we want to infer something more
- Another heuristics is to find inference for variables that only have one valid value in their domain
  - On line 13 change the test to $|T| = 1$
  - This can be extended to any value depending on the problem, e.g. only inferring if $|T| < n$
- There are 2 more places where we can introduce heuristics:
  - PickUnassignedVar
    - Minimum remaining value heuristic: pick the unassigned variable with the smallest effective

domain size, so we backtrack quickly if it fails
  - OrderDomainValue
    * Least constraining value heuristic: pick a value for the variable that rules out the least domain for other variables
- A special variant of CSP is the *boolean satisfiability problem* (B-SAT) where all the variables can only be 0 or 1
  - In the case of 3-SAT, all constraints specify relations between 3 variables
    * This can be expressed in terms of a logical expression over the 3 variables that must be true
  - For SAT, the only way we can infer is when all variables except one in a constraint has an assigned value
  - 3-SAT and above are NP-complete
- Another variant is the *binary CSP* where every constraint is defined over two variables
  - This is also in general NP-complete
- The 2-SAT problem is a combination of the two where every variable can only be 2 values, and every constraint is over two variables
  - This is one of the only variants of the problem that has a polynomial time solution

Lecture 6, Feb 13, 2024

Adversarial Games

- Consider an adversarial game where two players take turns, with one player trying to maximize the value of the final state of the game while the other tries to minimize it
  - The value of a game state, known as the *utility*, is not known until we reach a terminal state
  - All possible outcomes of the game can be modelled as a *game tree*
    * At each level, each possible action the player may take leads to a new subtree, until the terminal states as leaves
  - At each step the MAX player will try to take an action to maximize the value of the next state, while the MIN player will try to minimize it
  - If we assume that both players are playing optimally, then we can recursively assign a value to each state:
    * If it is a terminal state, take its utility
    * If the MAX player is playing, take the maximum of the values of the next states
    * If the MIN player is playing, take the minimum of the values of the next state
  - Then at each step, we simply take make the move that leads to the max or min value of the next state, depending on the player
  - This is the *minimax* algorithm
- $$\text{MiniMax}(s) = \begin{cases} \text{Utility}(s) & s \text{ is a terminal state} \\ \max_{a \in A} \text{MiniMax}(\text{Result}(s, a)) & \text{MAX player} \\ \min_{a \in A} \text{MiniMax}(\text{Result}(s, a)) & \text{MIN player} \end{cases}$$
- However, this requires that we recursively search the entire game tree to calculate the value of a state
  - The number of states grows exponentially as the game gets longer
  - This is usually impractical, e.g. in chess there are on the order of 10^{120} states
- We can optimize the search and prune subtrees that can't possibly give us a better result
  - As the max player, after we've searched one of the successor states, we have a lower bound for the value of the current state
  - We can pass this lower bound to the next level of the search, which will be the min player
  - Now as the min player, searching a successor state will give an upper bound for the current state's value
  - If this upper bound is lower than the lower bound from the max player, then there's no more point searching this state any further – the value of this state will always end up being lower than the previous states searched in the max layer, so this move will never be taken

Algorithm 2 MinValue(S)

1. **if** TERMINAL(S)=True **then return** UTILITY(S)
2. $v \leftarrow \infty$
3. **for** all action a in ACTIONS(S) **do**
4. $\quad v \leftarrow \min(v, \text{MaxValue}(\text{RESULT}(S,a)))$
5. **return** v

Algorithm 3 MaxValue(S)

1. **if** TERMINAL(S)=True **then return** UTILITY(S)
2. $v \leftarrow -\infty$
3. **for** all action a in ACTIONS(S) **do**
4. $\quad v \leftarrow \max(v, \text{MinValue}(\text{RESULT}(S,a)))$
5. **return** v

Figure 14: The minimax algorithm for computing the value of a state.

- The above intuition gives the *alpha-beta pruning* algorithm:
  - Let α be the highest value of all searched successors so far, from the MAX player's perspective
  - Let β be the lowest value of all searched successors so far, from the MIN player's perspective
  - Initialize $\alpha = -\infty$ and $\beta = \infty$
  - For the MIN player, if the value of any successor state is less than α, then we can stop searching (since this state will always have less value than α, so the MAX player will never consider it)
  - For the MAX player, if the value of any successor state is more than β, then we can stop searching (since this state will always have greater value than β, so the MIN player will never consider it)
- Plain minimax explores all $O(b^d)$ states
  - With α-β pruning we explore $O(b^{\frac{d}{2}})$ in the best case, but still $O(b^d)$ in the worst case
  - For randomized evaluation of states we get $O(b^{\frac{3}{4}d})$ on average
- This can make a huge difference but is still not enough in many cases
  - e.g. for chess we would still have 10^{60} states to check in the best case of α-β pruning
- For many games, we can create heuristics to estimate the value of a non-terminal game state, so we don't have to search until the end of the tree
  - We can construct a heuristic as $h(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s)$ where the w_i are weights and $f_i(s)$ are certain features of the game state
    - e.g. for chess features can be the number of pieces each side has left
  - We only search until a certain depth, and then we compute the heuristic if we haven't yet reached the terminal state
  - Today heuristics can be created using machine learning methods
  - A good heuristic should:
    - Evaluate terminal states in the same way as the true utility function would
    - Not take too long to compute
    - Favour states highly correlated with the chances of winning, for non-terminal states

Lecture 7, Feb 27, 2024

Bayesian Networks

- So far we have only discussed deterministic, fully observable task environments
- Partially observable or stochastic environments can be modelled with probability

Algorithm 5 MinValue(S,α,β)

1. **if** TERMINAL(S)=True **then return** UTILITY(S)
2. $v \leftarrow \infty$
3. **for** all action a in ACTIONS(S) **do**
4. $\quad v \leftarrow \min(v, \text{MaxValue}(\text{RESULT}(S,a),\alpha,\beta))$
5. $\quad$ **if** $v \leq \alpha$ then return v
6. $\quad \beta \leftarrow \min(v,\beta)$
7. **return** v

Algorithm 6 MaxValue(S,α,β)

1. **if** TERMINAL(S)=True **then return** UTILITY(S)
2. $v \leftarrow -\infty$
3. **for** all action a in ACTIONS(S) **do**
4. $\quad v \leftarrow \max(v, \text{MinValue}(\text{RESULT}(S,a),\alpha,\beta))$
5. $\quad$ **if** $v \geq \beta$ **then return** v // Pruning of branch
6. $\quad \alpha \leftarrow \max(\alpha,v)$
7. **return** v

Figure 15: The α-β pruning algorithm.

- Often we have multiple models of our state, and then based on evidence, we classify which model is correct (or which one we're most likely to be in)
- How can we store conditional probabilities efficiently?
  - If we have n variables each taking 2 values, to store the conditional probability over all combinations of variables we'd need 2^n entries
  - Not all variables may be dependent on each other; how can we take advantage of this?
- A *Bayesian network* is a probabilistic graphical model representing a set of variables and their conditional dependence via a directed acyclic graph (DAG)
  - In the DAG, an edge $A \to B$ denotes that B is conditionally dependent on A
  - Traversing the DAG gives us a chain of dependence between events
  - Often human intuition is used to determine which events have a causal relationship
  - At each node, we store a conditional probability table for the probability of the event at the node, given all its parents
    - The table has an entry for every combination of its parents' values
  - If a node has no parents, we simply store the absolute probability of that event (not conditioned on anything)
- We don't need to collect data about every possible event from the same sample, i.e. we may compute probabilities separately, using different datasets, for different nodes
  - However we always assume that whatever sample we take is representative of the population
  - This means we can combine different studies
- Bayesian networks allow compact representation of probability distributions
  - For a network over n nodes, if a node has at max q parents, then the space complexity is $O(n \cdot 2^q)$, which is often significantly less than 2^n
- The crucial assumption of Bayesian networks is the *Bayesian Network Law*: for any node, given its parents, its probability is completely independent of its non-descendants; i.e. nothing that came before it matters except for its parents
  - Note: v is a descendant of u if there is a directed path from u to v
  - Note that the probability of a node can still depend on its descendants
  - This also works even if we don't give the direct parents, as long as the probability of all the direct

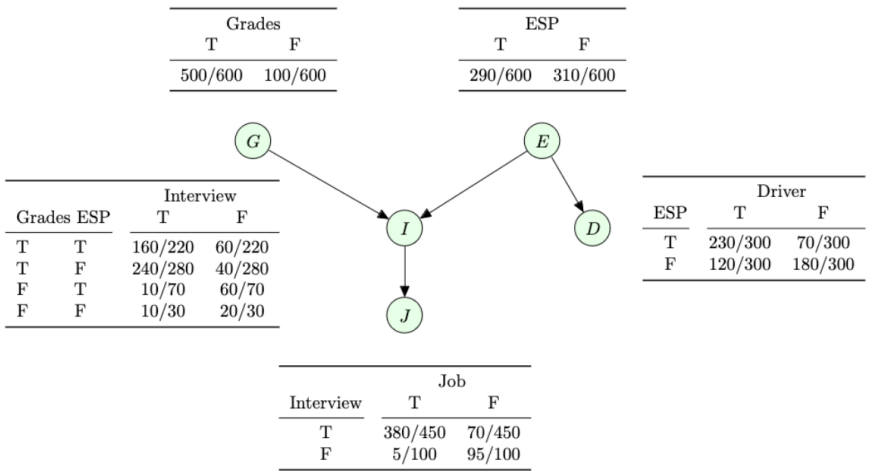

Figure 16: Example Bayesian network.

parents can be computed from the grandparents given
  - e.g. in the above graph, $P(I|G,E,D) = P(I|G,E)$ since I is not a descendant of D
- Example: in the graph below:
  - A and E are not independent
  - A and E, given B, are independent
  - A and E, given G, C, are independent
  - A and E, given G only, are not independent

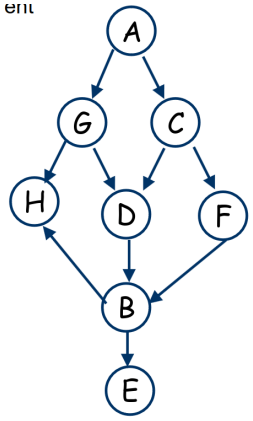

Figure 17: Example DAG network.

- For any set of events, $P(x_1, x_2, \ldots, x_n) = \prod_{i=1}^{n} P(x_i|x_{i+1}, \ldots, x_n)$
$$= P(x_1|x_2, \ldots, x_n)P(x_2|x_3, \ldots, x_n)\ldots P(x_{n-1}|x_n)P(x_n)$$
  - Given the Bayesian network law, we can simplify these terms significantly by taking out all the variables except for the direct parents
  - e.g. $P(G, I, J, E, D) = P(J, I, G, D, E)$
$$= P(J|I, G, D, E)P(I|G, D, E)P(G|D, E)P(D|E)P(E)$$
$$= P(J|I)P(I|G, E)P(G)P(D|E)P(E)$$
  - This means that we can always compute $P(x_1, \ldots, x_n)$ just by looking at the conditional probabilities stored in the network
    * There are multiple ways to expand this joint probability, but there will always be one order

that works
    * The order that works is determined by the topological sorting of the graph
  - To compute the joint probability over all the events in the network, compute the product of each event conditioned on its immediate parents
    * Always guaranteed to work due to the existence of a topological sort as above
- Using Bayesian networks, we can compute the probability of 2^n events using only $n \cdot 2^q$ entries

Lecture 8, Mar 5, 2024

Independence in Bayesian Networks

- Given a Bayesian network, how can we tell if there exists a dependence between variables, given some other variable?
- Example: in the DAG from the previous lecture, is $P(E|A,G,C) = P(E|G,C)$? (i.e. given G, C, is E independent of A?)
  - We only know that given all direct parents, a node is independent of its non-descendants
  - Use the law of total probability to bring in the direct parents, and then use Bayes rule to "move" variables from the left of the conditional to the right of the conditional
  - Assume that all variables have binary values, e.g. A can take values of a or $-a$
  - $$\begin{aligned} P(E|A,G,C) &= \sum_B P(E,B|A,G,C) \\ &= \sum_B P(E|B,A,G,C)P(B|G,A,C) \\ &= \sum_B P(E|B)P(B|G,A,C) \\ &= \sum_B P(E|B) \sum_{D,F} P(B,D,F|G,A,C) \\ &= \sum_B P(E|B) \sum_{D,F} P(B|D,F,G,A,C)P(D,F|G,A,C) \\ &= \sum_B P(E|B) \sum_{D,F} P(B|D,F)P(D,F|G,C) \\ &= \sum_B P(E|B)P(B|G,C) \\ &= P(E|G,C) \end{aligned}$$
- General rule: if all paths from one node to another have to pass through at least one of the given nodes, then these two nodes are independent, given the other nodes
  - i.e. remove all given nodes from the graph, if there no longer exists a path between two nodes, then they are independent
  - This only holds if the given nodes are descendants of only one of the nodes given
  - e.g. $P(B|E,D,FA) \neq P(B|E,D,F)$, because E is a descendant of B and A
- More formally, a set of variables $\mathcal{E}$ d-separates X and Y if it blocks every undirected path in the network between X and Y; given this, X and Y are independent
  - Every path from X to Y must pass through a variable in the set
  - Let $\mathcal{P}$ be any undirected path from X to Y in the network; $\mathcal{E}$ blocks $\mathcal{P}$ iff there is some node $Z \in \mathcal{P}$ such that one of the following is true:
    1. $Z \in \mathcal{E}$ and one arc on $\mathcal{P}$ enters Z and another leaves Z
    2. $Z \in \mathcal{E}$ and both arcs on $\mathcal{P}$ leave Z
    3. Both arcs on $\mathcal{P}$ enter Z and neither Z, nor any of its descendants, are in $\mathcal{E}$

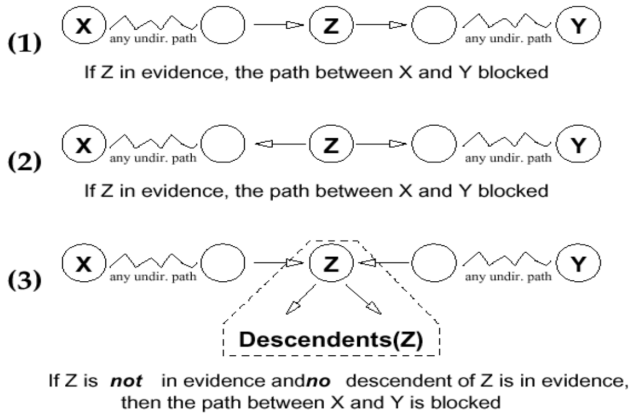

Figure 18: The 3 cases of d-separation.

Variable Elimination

- To compute the joint probability of all the variables, we can keep applying Bayes rule to express it as a sum of probabilities in the tables
- The conditional probability can be obtained using Bayes rule, by dividing the joint probability of all the variables by the joint probability of the variables being conditioned on
- If we don't want all variables, use the law of total probability over all the variables that the expression does not involve
  - However, to sum over all the different combinations of the variables not involved, we would need an exponential number of terms
  - Expand out the joint probability using the product rule, and then break apart the sum as much as possible
  - Start from the innermost sum, and if we're able to complete it without depending on any of the outer sum variables, we can factor it out
  - If the inner sum depends on a variable from an outer sum, the value of the sum will be a function
    * Compute the value of the inner sum for all possible values of that variable
    * The result is another conditional probability table, like the ones we started with
- *Factors* are these probability tables, from either the original conditional probabilities in the expression, or the functions produced as a result of the above
  - Each factor is a function over some variables, e.g. $f(A, C)$ denotes a factor over A and C, which could be $P(C|A)$
    * However, the factors aren't necessarily probabilities; they don't always sum to 1
    * Factors don't always have a probabilistic interpretation
  - Factors have the following operations:
    * Product: if $f(X, Y)$ and $g(Y, Z)$ with Y in common, then the product $h = f * g$ is a factor $h(X, Y, Z) = f(X, Y) \cdot g(Y, Z)$
      - We look at the column that's the same between the two tables and multiply corresponding entries
      - Like a database join
    * Summing out: if $f(X, Y)$, summing out X from f produces the new factor $h = \sum_X f$, where $h(Y) = \sum_X f(X, Y)$
    * Restricting: if $f(X, Y)$, restricting f to $X = a$ produces $h = f_{X=a}$ where $h(Y) = f(a, Y)$
  - The summing and restricting operations may produce a factor over no variables, which is a constant that we can take out
- The variable elimination algorithm: given query variable(s) Q, evidence (i.e. given) variables $\mathcal{E}$, remaining variables $\mathcal{Z}$, form the original set of terms $\mathcal{F}$

1. Replace all factors $f \in \mathcal{F}$ that mention variables in $\mathcal{E}$ with its restriction (i.e. set the known values)
2. For each $Z_j \in \mathcal{Z}$, eliminate it by:
    1. Find all factors $f_i \in \mathcal{F}$ that include Z_j
    2. Compute the new factor $g_j = \sum_{Z_j} f_1 * f_2 * \cdots * f_k$ (i.e. take their product and sum out Z_j)
    3. Remove all f_i from $\mathcal{F}$ and add g_j to $\mathcal{F}$
3. Take the product of the remaining factors (which should refer only to Q) and normalize to produce $P(Q|\mathcal{E})$
    - The resulting factor is unnormalized, i.e. $f(Q) + f(\bar{Q}) \neq 1$, so we need to compute the normalization factor so they sum to 1

Lecture 9, Mar 12, 2024

Variable Elimination Complexity

- As we perform variable elimination, we may end up with factors that had more variables than they began with, due to multiplication of factors
    - If each variable is binary, then a factor with k variables takes 2^k space and time to compute/store
    - Can we put a bound on this?
- A *hypergraph* is a set of vertices like an ordinary graph, but instead of edges connecting two vertices, it has *hyperedges* connecting multiple vertices
    - Each hyperedge is a set of vertices

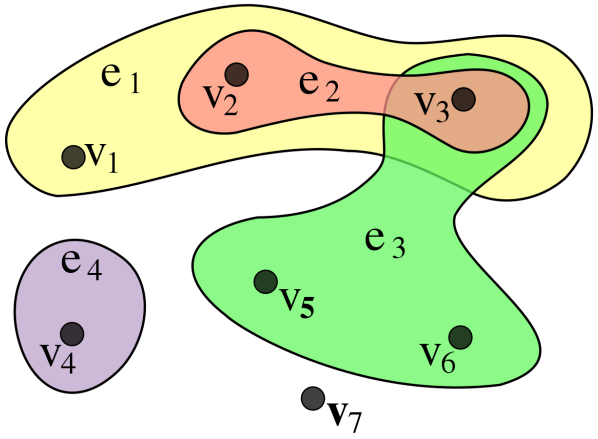

Figure 19: An example hypergraph.

- For a Bayesian network we start initially with a hypergraph where the vertices correspond to each variable and the hyperedges are the factors
- When we try to eliminate a variable C, we remove all the factors where the variable appears and add a new factor
    - We remove the hyperedges that C appears in and add a new hyperedge, containing all the variables that C was once connected to
    - The size of the hyperedges can grow or reduce
- Given an ordering of the variables and an instal hypergraph $\mathcal{H}$, eliminating the variable yields a sequence of hypergraphs $\mathcal{H}_0, \ldots, \mathcal{H}_n$
    - The *elimination width* k of π is the maximum size of any hyperedge in any of the hypergraphs
        * The elimination width of $\mathcal{H}$ is the minimum elimination width of any of the $n!$ different orderings of the variables
    - The complexity is $O(2^k)$ in both time and space (since a table with 2^k entries needs to be computed and stored)

  - In the worst case k can be equal to the number of variables
- We can try to find the best order of elimination that gives the smallest k, but this is an NP-hard problem
  - Heuristics can be used to find orderings with low elimination widths
  - In practice, we don't often encounter graphs that force a very high elimination width
- A *polytree* is a singly connected Bayesian network, i.e. there is only a single path between any pair of nodes
  - Eliminating a singly connected node (i.e. node connected to only one other node) will not increase the size of the hypergraph
  - Having a polytree ensures that at every step in elimination, there is always at least one singly connected node
  - Therefore the elimination width is simply the size of the largest input conditional probability table
- On a polytree, variable elimination can run in linear time in the size of the network (not necessarily linearly in the number of variables)
  - There always exists a good variable elimination order, but not every order is good

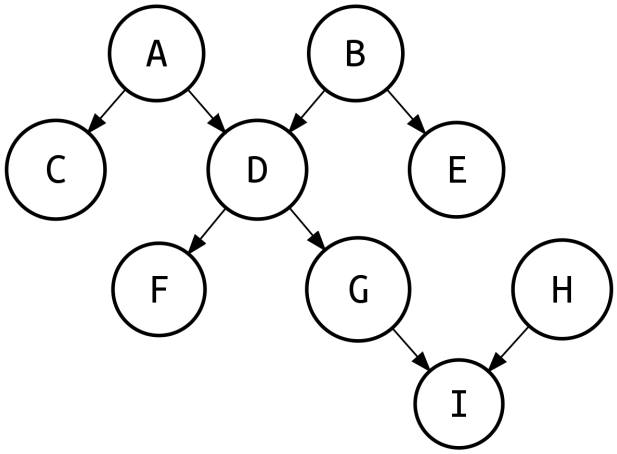

Figure 20: An example polytree.

- One effective heuristic for VE is to always eliminate the variable that creates the smallest sized factor
  - This is the *min-fill heuristic*
  - For polytrees, this guarantees linear time

Bayesian Model Selection

- Based on the data we have, we can come up with a number of different models; how do we select which one is the best?
- We can randomly leave out a part of the data as the validation set, and make the model on the training set
- The model to keep is the one that makes the validation data more likely
  - Use Bayesian hypothesis testing/likelihood ratio test
  - $\frac{P(\mathcal{E}|M_1)P(M_1)}{P(\mathcal{E}|M_2)P(M_2)} \underset{M_2}{\overset{M_1}{\gtrless}} 1$ where $\mathcal{E}$ is a set of evidence in the validation set
- We can improve models with local search, or start with a random model and build it using local search
  - Define a neighbourhood around the model, e.g. by removing or adding some edges
  - Now check everything around the neighbourhood of a model and compare it with the model using the likelihood ratio
  - Use the new model if it's better and repeat with the local search, or use simulated annealing

Lecture 10, Mar 19, 2024

Knowledge Representation and Reasoning (KR&R)

- How do we represent the information that we know, and reason intelligently using this information?
- This is the problem of *knowledge representation and reasoning* (KR&R)
  - *Representation*: symbolic encoding of propositions believed by some agent
    - Assigning symbols, and relating them
  - *Reasoning*: manipulation of said symbolic encoding to produce new propositions believed by the agent, but were not initially explicitly stated
    - Making rules for manipulating the symbols
- *Boolean algebra* or *propositional logic* is a set of rules for how to manipulate these propositions
  - In propositional logic, propositions are restricted to true statements

Propositional Logic

- For propositional logic, the set of operators are: $\vee$ (OR), $\wedge$ (AND), $\neg$ (NOT) and the brackets (and)
  - $\circ$ denotes any one of the binary operators
- Define PROP as the set of all propositions, OP as the set of all operators, and WDF (aka WFF) as the set of all well-defined (aka well-formed) formulas
  - Valid propositions follow a syntax; all propositions in the WDF is considered well-defined
  - If PROP is a finite set, then WDF is a countably infinite set
  - We construct WDF recursively as follows:
    - $\text{FORM}_0 = \text{PROP}$
    - $\text{FORM}_{i+1} = \text{FORM}_i \cup \{ (\alpha \circ \beta) \mid \alpha, \beta \in \text{FORM}_i \} \cup \{ (\neg\alpha) \mid \alpha \in \text{FORM}_i \}$
    - $\text{FORM} = \bigcup_{i=0}^{\infty} \text{FORM}_i$
  - Note that this excludes things like $(\alpha \circ \beta \circ \gamma)$
- Every formula is either:
  1. Atomic formula: a member of PROP
  2. Composite formula: made of a *primary connective* and set of *subformulas*
     - e.g. in $(\neg\alpha)$, $\neg$ is the primary connective and $\{ \alpha \}$ is the subformula; in $(\alpha \circ \beta)$, $\circ$ is the primary connective, and $\{ \alpha, \beta \}$ are the subformulas

Theorem

Unique Readability Theorem: Every (well-formed) formula is either atomic, or has a unique primary connective and unique set of well-defined subformulas.

- A *truth assignment* is a mapping $\tau : \text{PROP} \mapsto \{ 0, 1 \}$
  - This basically assigns a value to all the propositions, but not all well-defined formulas
- The *extension* of τ for a formula φ is $\bar{\tau}(\varphi) = \begin{cases} \tau(p) & \varphi = p \in \text{PROP} \\ \neg(\bar{\tau}(\alpha)) & \varphi = (\neg\alpha) \\ \circ(\bar{\tau}(\alpha), \bar{\tau}(\beta)) & \varphi = (\alpha \circ \beta) \end{cases}$
  - Sometimes we denote this as $\varphi(\tau)$
  - This assigns a value to all WDF
- We say that τ *models* φ, or $\tau \models \varphi$ iff $\bar{\tau}(\varphi) = 1$
- Let $\text{AP}(\varphi) = \begin{cases} \{ p \} & \varphi = p \in \text{PROP} \\ \text{AP}(\alpha) & \varphi = (\neg\alpha) \\ \text{AP}(\alpha) \cup \text{AP}(\beta) & \varphi = (\alpha \circ \beta) \end{cases}$
  - This defines the set of atomic propositions that appear in a formula

Theorem

Relevance Lemma: If for all propositions p that appear in φ, both τ_1 and τ_2 assign it the same value, then τ_1 models φ iff τ_2 models φ, i.e.

$$\forall p \in \text{AP}(\varphi), \tau_1(p) = \tau_2(p) \quad \implies \quad \tau_1 \models \varphi \iff \tau_2 \models \varphi \text{ and } \varphi(\tau_1) = \varphi(\tau_2)$$

- Define $\varphi \models\!\!=\!\!\models \psi$ iff $\forall \alpha \in 2^{\text{PROP}}, \varphi(\tau) = \psi(\tau)$, i.e. they have the same value for all assignments
  - e.g. $((p \vee q) \vee p) \models\!\!=\!\!\models (p \vee q)$
- $\text{models}(\varphi) = \{ \varphi \mid \tau \models \varphi \text{ or } \tau(\varphi) = 1 \text{ or } \bar{\tau}(\varphi) = 1 \}$
- *Valid* formulas, denoted $\models \varphi$, are formulas such that $\forall \tau, \tau \models \varphi$ – they are always true
  - If $\forall \tau, \tau \not\models \varphi$, then φ is *unsatisfiable* – they are always false
  - If $\exists \tau, \tau \models \varphi$, then φ is *satisfiable* – they are sometimes true
- Propositional logic is limited to only boolean variables, which makes cross-references between individuals in statements impossible
  - e.g. we can't model statements like "if x likes y and y plays golf then x watches golf", because we don't have variables or statements that are only sometimes true
- We also have no quantifiers; to state a property for all or some members of the domain, we have to explicitly list them

Lecture 11, Mar 26, 2024

First-Order Logic

- We generalize propositional logic to have the notion of variables
- *First-order logic* consists of the following components:
  - A set of *variables*, V
    - These can take values from a domain D
  - A set of *predicate/relation symbols* $P^k \colon D^k \mapsto \{0, 1\}$ where k is the number of arguments
    - These take a set of arguments (variables) and can be true or false, depending on the value of the variables
    - P^0 is the set of predicates that don't take any arguments, which is the set of propositions
    - These define relations among variables
  - A set of *function symbols* $f^k \colon D^k \mapsto D$ where k is the number of arguments
    - These define functions based on the variables, returning another variable
    - A special case of the relations
  - The *quantifiers* $\forall$ and $\exists$
- Define the set of all terms:
  - $\text{TERM}_{i+1} = \text{TERM}_i \cup \{ f_n^k(t_1, \ldots, t_k) \mid t_1, \ldots, t_k \in \text{TERM}_i, \forall n, k \}$
  - $\text{TERM}_0 = V$
- Define the set of all well-formed formulas:
  - $$\begin{aligned}\text{FORM}_{i+1} = \quad & \text{FORM}_i \\ & \cup \{ (\alpha \circ \beta) \mid \alpha, \beta \in \text{FORM}_i \} \\ & \cup \{ (\neg\alpha) \mid \alpha \in \text{FORM}_i \} \\ & \cup \{ \forall x \varphi \mid x \in V, \varphi \in \text{FORM}_i \} \\ & \cup \{ \exists x \varphi \mid x \in V, \varphi \in \text{FORM}_i \}\end{aligned}$$
    - We augment our definition from propositional logic with the new quantifiers $\forall$ and $\exists$
  - $\text{FORM}_0 = \{ P_n^k(t_1, \ldots, t_k) \mid t_1, \ldots, t_k \in \text{TERM}, \forall n, k \}$
    - This is the set of all predicates over all terms
- Consider the expression $\forall x \exists y (x + y > 3)$
  - Formally we express this as $\forall x \exists y (> (+(x, y), 3))$
  - $+$ is a function and $>$ is a predicate
  - We need to define the domain of all variables, and define the meaning of $+$ and $>$

- A *context* or *structure* consists of $\mathcal{A} = (D, f_i^{k,\mathcal{A}}, \dots, P_i^{k,\mathcal{A}}, \dots)$, which is a domain and definition of all the functions and predicates
  - $f_i^{k,\mathcal{A}}$ defines the meaning of function f_i^k in the context of $\mathcal{A}$
  - $P_i^{k,\mathcal{A}}$ defines the meaning of predicate P_i^k in the context $\mathcal{A}$
  - The definitions assign $D^k \mapsto \{0, 1\}$ for every combination of the values of the variables
- A k-ary function can be converted into a predicate by adding an extra argument; so functions are syntactic sugar that's not needed to define first-order logic
- An *assignment* is $\sigma\colon V \mapsto D$ which gives a value to all variables
  - We need to assign values to variables before evaluating some expressions, e.g. $\forall x(x + y > 3)$
  - $\sigma(x \mapsto m)$ or equivalently $\sigma(x/m)$ denotes the value m being assigned to x
- Similar to propositional logic $\mathcal{A}, \sigma \models \varphi$ if φ is true under the structure $\mathcal{A}$ and assignment σ
  - Note that in ϕ we might have variables appearing in quantifiers that have been assigned a value by σ; in this case we don't care about the assignment in σ
  - $\exists x\psi$ iff there exists $m \in D$ such that $\mathcal{A}, \sigma(x \mapsto m) \models \psi$
  - $\forall x\psi$ iff for all $m \in D$ we have $\mathcal{A}, \sigma(x \mapsto m) \models \psi$
- The *extension* of σ is $\bar{\sigma}\colon \text{TERM} \mapsto D$ which assigns a value to all terms
  - This can be defined recursively, since a term is either a variable or a function of terms
  - $\bar{\sigma}(t) = f_i^k(\bar{\sigma}(t_1), \dots, \bar{\sigma}(t_k))$ for $t \in \text{TERM}$
  - Base case is $\bar{\sigma}(t) = \sigma(t)$ for $t \in V$
- We are now interested in the analog of the relevance lemma from propositional logic
  - Not all variables in formulas are *free variables*, e.g. for $\exists x(x + y > 3)$, x is not a free variable because of the $\exists$, and it doesn't matter what value σ assigns to it
- $\text{FreeVars}\colon \text{FORM} \mapsto 2^V$, a mapping from formulas to sets of variables
  - $\text{FreeVars}(\forall x\varphi) = \text{FreeVars}(\varphi) \setminus \{x\}$
  - $\text{FreeVars}(\exists x\varphi) = \text{FreeVars}(\varphi) \setminus \{x\}$
  - $\text{FreeVars}(\neg\varphi) = \text{FreeVars}(\varphi)$
  - $\text{FreeVars}(\varphi \circ \psi) = \text{FreeVars}(\varphi) \cup \text{FreeVars}(\psi)$
  - $\text{FreeVars}(P_i^k(t_1, \dots, t_k)) = \text{FreeVars}(t_1) \cup \dots \cup \text{FreeVars}(t_k)$ for $t_n \in \text{TERM}$
  - $\text{FreeVars}(f_i^k(t_1, \dots, t_k)) = \text{FreeVars}(t_1) \cup \dots \cup \text{FreeVars}(t_k)$ for $t_n \in \text{TERM}$
  - $\text{FreeVars}(x) = \{x\}$ for $x \in V$
- *Relevance lemma*: if $\forall x \in \text{FreeVars}(x)$, $\sigma_1(x) = \sigma_2(x)$, then $\mathcal{A}, \sigma_1 \models \varphi$ iff $\mathcal{A}, \sigma_2 \models \varphi$
  - This has the same interpretation as the relevance lemma for propositional logic
- Define $\mathcal{A} \models \varphi$ iff $\forall\sigma(\mathcal{A}, \sigma \models \varphi)$, i.e. φ is always satisfied in structure $\mathcal{A}$ (analog of valid formulas)
  - Likewise $\mathcal{A} \not\models \varphi$ iff $\forall\sigma(\mathcal{A}, \sigma \not\models \varphi)$ (analog of unsatisfiable formulas)
  - We only need to care about the free variables, since the non-free ones don't affect whether φ is modelled
- If $\text{FreeVars}(\varphi) = \varnothing$, then φ is a *sentence*
  - We can use sentences to store our knowledge in a knowledge base
- Define $\varphi \models \psi$ if the set of all assignments that model φ is a subset of all assignments that model ψ (so if φ is modelled by an assignment, ψ will also be)
  - $\text{models}(\varphi) \subseteq \text{models}(\psi)$
  - Alternatively $\text{models}(\varphi) \cap \overline{\text{models}(\psi)} = \varnothing$
  - Equivalently $\text{models}(\varphi \wedge \neg\psi) = \varnothing$
  - If φ is a knowledge base of sentences, we use this to check if a formula is true
- $p \wedge (\neg p)$ is an *empty clause*, denoted $()$, which is a contradiction
- $(\alpha \vee p) \wedge (\neg p \vee \beta)$ gives $(\alpha \vee \beta)$
  - This is known as *resolution*
  - This is the transitivity of implication, since $\neg x \vee y$ means $x \to y$
- Given some logical statement we can keep applying resolution, and eventually if we end up with an empty clause, we know the original statement was false because it leads to a contradiction
- Therefore if we want to check if our knowledge base models some formula α, we can check if $KB \wedge (\neg\alpha)$ leads to an empty clause

Lecture 12, Apr 2, 2024

Proof Procedures

- Given a knowledge base KB, we want to know whether $KB \models \alpha$ where α is some formula (whether the knowledge base *entails* α)
  - We can show that $KB \wedge \neg\alpha$ is unsatisfiable
- If we can manipulate a formula into the empty formula, denoted by $\square$, then we know it is unsatisfiable
  - We know that if $\varphi = \alpha \vee \beta$, then $\text{models}(\varphi) = \text{models}(\alpha) \cup \text{models}(\beta)$
  - If the formula is empty, we have nothing to union, so nothing models it
  - Therefore the empty formula is unsatisfiable
- We are interested in formulas in their *clausical form*, $(C_1 \wedge C_2 \wedge C_3 \wedge \cdots \wedge C_m)$
  - Each clause $C_i = (l'_1 \vee l'_2 \vee \cdots \vee l'_k)$
  - Each l'_i is a *literal*, which is a proposition p or $\neg p$
  - Any formula can be written in clausal form
  - Using the Tseytin transformation, any formula can be converted into an equisatisfiable formula that is linear in size
    - This is as opposed to the naive method of just expanding it out, which leads to exponential size formulas
- In propositional logic, each step of a *proof* is derived from *resolution* $\dfrac{(\alpha \vee p) \wedge (\neg p \vee \beta)}{(\alpha \vee \beta)}$ and $\dfrac{p \wedge \neg p}{\square}$
- Suppose we want to prove by resolution that formula φ is false, i.e. a *resolution refutation*; resolution refutation is a sequence of clauses $C_1, \ldots, C_t$ where $C_t = \square$, and all $C_i \in \varphi$ or $\dfrac{C_{i_1} C_{i_2}}{C_k}$ where $i_1, i_2 < k$
  - All the clauses are either the original formula, or implied by previous formulas, leading to an empty formula that is unsatisfiable
- In first-order logic we do this over predicates instead, $\dfrac{(\alpha(x) \vee \neg P(y))(P(y) \vee \beta(z))}{\alpha(x) \vee \beta(x)}$, but quantifiers may be involved
  - If the quantifies are the same, we can do this
  - But if quantifiers are different, this isn't true anymore
  - $\exists x, y, z(\varphi(x, y, z))$ is equivalent to $\neg\forall x, y, z = \neg\varphi(x, y, z)$
- *Skolemization*: we can get rid of one set of quantifiers, e.g. replacing all $\exists$ with $\forall$ or vice versa
- Given any formula we want it first in a form $\forall x, y, z(C_1 \wedge C_2 \wedge \cdots \wedge C_m)$
  - First we get all the quantifiers out of the formula, and then apply skolemization so we are only left with $\forall$
    - Note with $\exists y P(y) \wedge \exists y R(y)$ we cannot simply take out $\exists y$ because the y in P and R are not guaranteed to be the same
    - To avoid this, first we make sure all variables in quantifiers have unique names
    - Once we have unique names for all of them, we can pull them out
    - Note the order of quantifiers matters – we cannot swap them
  - Once we take it out, we can apply resolution just like in propositional logic