Introduction to AI

Donglin Jia

July 22, 2020

1 Lecture 1 – Definition of AI

1.1 What is AI?

- Two ways to measure performance:
  - measure against human → how human think and act
  - measure against rationality → mathematics defined, developed analytic model
- Care about:
  - how they think
  - how they behave → can be observed internally
- Systems that think like humans
- Systems that act like humans
- Systems that think rationally
- Systems that act rationally

1.2 Human

1.2.1 Thinking Humanly

- The Cognitive Modeling Approach
  - Human as an intelligence – think like human
  - How human think?
    - Introspection
    - Psychological experiments
    - Brain imaging
  - Cognitive science – develop the theory of mind using AI modelling

1.2.2 Acting Humanly

- The Turing Test Approach
  - Assumption: the interrogator allows to ask any questions, if entity is intelligence, then it should handle all those including visual signal or/and object
  - Simple but powerful idea.
  - Need to be able to do NLP, knowledge representation, reason, learning, proceed to object and move and manipulate object
  - Drawback:
    - recognize but not realize intelligence

1.3 Rationality

- Rationality: an abstract ”ideal” of intelligence, rather than ”whatever humans do”
- Doing the right thing, given what it knows

1.3.1 Thinking Rationality

- The Laws of Thought Approach
  - Greek philosopher Aristotle defined syllogisms
    - Given right premise get right conclusion
  - the logicist tradition – using logic to express the knowledge
  - Two obstacles for using this approach in practice
    - Too precise to express knowledge in logic → have difficulty to translate knowledge into logic
    - Hard to resolve the problem even if all the knowledge has been encode in logic, e.g. bruce force search takes long time and does not provide the which order to use first

1.3.2 Acting Rationality

- The Rational Agent Approach:
  - Agent means todo
  - A rational agent acts to achieve the best outcome
  - Rational behavior:
    - Create and presume goal and agent should operate atomically that be able to proceed the environment and be able to adapt the changes and learn.

1.4 Caring about Behavior rather than Thoughts

- Acting rationally is more general idea than thinking rationally
  - no right/wrong answer, need to act quickly

1.5 Measure Success against Rationality rather than Human

- Human acting irrational more often
- Rationality is a well-defined concept mathematically $\rightarrow$ easy to develop theoretical model & perform experiments
- Understanding the principles behind certain object and using that to develop certain model

2 Lecture 2 Uniformed Search

2.1 Applications

- Travelling sales man problem
- Propositional Satisfiability: FCC spectrum auction
- 8 puzzle: rearrange numbers into normal order starts with 1 to 8 in a 3x3 grid.
- N-Queens Problem: Place n queens on an $n \times n$ board so that no pair of queens attacks(horizontal, vertical, cross) each other.

2.2 Formulating a Search Problem

- Component for a search problem can be represented in a graph.
- **Definition: Search Problem**
  - A set of states
  - A start states
  - Goal states or a goal test
    - A boolean function which tells us whether a given state is a goal state
  - A successor (neighbor) function
    - an action which takes us from one state to other states
  - A cost associated with each action

  For example – 8 puzzle Search Problem

  State: $x_{00}x_{01}x_{02}$, $x_{10}x_{11}x_{12}$,$x_{20}x_{21}x_{22}$ x_{ij} is the number in row i and column j, $i, j \in \{0, 1, 2\}, x_{ij} \in \{0, 1, \cdots, 8\}$ $x_{ij} = 0$ iff the square is empty.
  Initial state: 530, 876, 241
  Goal state: 123, 456, 780
  successor function: the state generated by moving the empty square left, right, up and down, whenever possible.
  cost function: each move take one cost

- The state definition can affect the amount of information needed to encode each state, and how difficult it is to implement the successor function.
- The successor function can affect the size and structure of the search graph

2.3 Solve a search Problem

- Generating search tree
  - consists start node, explored nodes, frontier and unexplored nodes.
  - A frontier consists of a set of path from starting node to all the neighbors of all the last explored nodes. It also contains all the leaf nodes that is available for expansion.

Algorithm 1 A Generic Search Algorithm

```
1: Search(Graph, Start node s, Goal test goal(n))
2: frontier = {(s)};
3: while frontier is not empty do
4:     select and remove path $(n_0, \cdots, n_k)$ from frontier
5:     if goal($n_k$) then
6:         return $(n_0, \cdots, n_k)$
7:     end if
8:     for every neighbour $n$ of $n_k$ do
9:         add $(n_0, n_1, \cdots, n_k, n)$ to frontier;
10:    end for
11: end while
12: return no solution
```

- Every time we go through WHILE loop, we need select one path from frontier. – which path we select from frontier defines our search strategy which is here.

2.4 Uniformed Search Algorithms

2.4.1 Depth-first Search

- Treats the frontier as a stack (LIFO)
- Expands the last/most recent node added to the frontier
- It starts one successor and explores all the paths first, then turns to another successor then explores all its paths, and repeat, until it reaches all successors for current node.
- Properties of DFS:
  b: branching factor (there are less than b successors for each node)
  m: maximum depth
  d: depth of shallowest goal node
  - Space complexity is $O(bm)$, we might find the deepest result (m level).
    - Need to back track if no goal node found
    - There are m level the path will be maximum m nodes in a path, each node has maximum b successors, thus it needs to memorize all its successors for back track, thus $m \times b$
  - Time complexity is $O(b^m)$ (visit all the states)
    - There are m levels, the first level has 1 node, second level has at most b nodes, and so on, then m^{th} level has b^{m-1} nodes.
    - Compute all together it is bounded by $O(b^m)$
  - It may not find a solution if there is an infinite path forever.

Algorithm 2 Search Algorithm with Cycling Pruning

```
1:  Search(Graph, Start node s, Goal test goal(n))
2:  frontier = {(s)};
3:  while frontier is not empty do
4:      select and remove path $(n_0, \cdots, n_k)$ from frontier
5:      if goal($n_k$) then
6:          return $(n_0, \cdots, n_k)$
7:      end if
8:      for every neighbour $n$ of $n_k$ do
9:          if $n \notin (n_0, \cdots, n_k)$ then
10:             add $(n_0, n_1, \cdots, n_k, n)$ to frontier;
11:         end if
12:     end for
13: end while
14: return no solution
```

- To check if the node has already been visited. If yes, then there is a cycle.

2.4.2 Breath-First Search

- Treats the frontier as a queue (FIFO)
- Expands the first/oldest node added to the frontier.
- Properties of BFS:
  b: branching factor (there are less than b successors for each node)
  m: maximum depth
  d: depth of shallowest goal node
  - Space complexity: $O(b^d)$ (size of the level containing the shallowest goal mode)
  - Time complexity: $O(b^d)$ (find the shallowest goal node)
  - Guarantee to find a solution if exists and guarantee to find the shallowest goal node.

Algorithm 3 Search with Multi-Path Pruning

```
1:  Search(Graph, Start node s, Goal test goal(n))
2:  frontier = {(s)};
3:  explored = {};
4:  while frontier is not empty do
5:      select and remove path $(n_0, \cdots, n_k)$ from frontier
6:      if $n_k \notin$ explored then
7:          add $n_k$ to explored
8:          if goal($n_k$) then
9:              return $(n_0, \cdots, n_k)$
10:         end if
11:         for every neighbour $n$ of $n_k$ do
12:             add $(n_0, n_1, \cdots, n_k, n)$ to frontier;
13:         end for
14:     end if
15: end while
16: return no solution
```

- Since we can sometimes encounter with multi-path that reach to the same nodes, then we will have duplicate sub-tree.
- However BFS intends to find the shortest path, therefore, we do not want to waste efforts on tracing the duplicated sub-tree especially for those with longer path.
- Therefore, we mark the node that we first met as explored so that we will not revisit it in the future.

Choose BFS or DFS

- When memory is limited → DFS
- All solutions are deep in the tree → DFS
  - Then d is approaching m, so with the similar time complexity, DFS has better performance in space complexity.
- The search graph contains a cycle → BFS
  - DFS will loop forever if there is a cycle.
- The branching factor is large/infinite → DFS
  - BFS needs too much space while DFS only needs linear space
- Find shallowest goal mode → BFS
  - BFS guarantee to find the shallowest solution if there is one.
- Some solutions are very shallow → BFS

2.4.3 Iterative Deepening Search

- For every depth limit, perform depth-first search until the depth limit is reached.
- Keep track depth limit and which node we have in the frontier.
- Combine BFS and DFS
  - BFS: search tree level by level
  - DFS: for each depth limit, we are doing DFS
- Properties IDS:
  - Space complexity: $O(bd)$, we will find the shallowest one with depth d
  - Time complexity:
    - Since we use depth limit to search level by level, therefore there are certain number of repeated searches on each level. For level one, it has been search d times since it always starts from first node.
    - Consider we have

$$b^d + 2 \times b^{d-1} + 3 \times b^{d-2} + \cdots + db + d + 1 \leq b^d (\frac{b}{b-1})^2$$

    - Similar to BFS but a bit worse because of re-computation
  - Guarantee to find a solution if one exist and find the shallowest one.

2.4.4 Lowest Cost First Search

- Need an algorithm to find the optimal solution – the solution with the lowest total cost
- The frontier is a priority queue ordered by path cost (cost(n)). Expand the path with the lowest total cost. (a.k.a Dijkstra's Shortest Path Algorithm)
- Complexity:
  - Space: exponential (worst case: track all the nodes on the bottom layers)
  - Time: exponential (need to visit all the nodes in the search tree)
  - Guarantee to find a solution with lowest cost.

3 Lecture 3: Heuristic Search

3.1 Difference between Uninformed search algorithm and Heuristic search algorithm

- An uninformed search Algorithm:
  - Consider every state to be the same
  - Does not know which state is closer to the goal
  - May not find the optimal solution
- An heuristic search algorithm
  - User heuristics function to estimate how close the current state is to the goal state
    - Using domain knowledge $\rightarrow$ come up with estimation
    - Domain knowledge not always be correct $\rightarrow$ might be wrong in some areas
  - Try to find the optimal solution (also find a solution faster)

3.2 Heuristic Function

- A search heuristic $h(n)$ is an estimate of the cost of the cheapest(shortest/best) path from node n to a goal node.
  - $h(n)$ is arbitrary, non-negative, and problem-specific.
    - Arbitrary: come from some useful domain knowledge which might be wrong
    - Non-negative: it is the estimate of the total cost of path. (no-negative cost is considered)
    - Problem-specific: Different intuition to different problem
  - If n is a goal node, $h(n) = 0$
    - When reaching the goal state, the cost to get it is just 0.
  - $h(n)$ must be easy to compute without search
    - The purpose of heuristic function is to simplify the difficulty of problems. With additional search, it makes problems more complex.
    - Search approaches will only be applied if the problem is difficult.

3.3 Greedy Best-First Search

Relay on heuristic function as only information.

- Frontier is a priority queue ordered by the heuristic $h(n)$
- Expand the node with the lowest $h(n)$

Circumstances of cannot find solution or optimal solution

- Heuristic function is wrong $\rightarrow$ guided the search to a loop
- Heuristic function's value is not accurate
- Ignored the cost

3.4 A^* Search

- The frontier is a priority queue ordered by $cost(n) + h(n) = f(n)$
  - $cost(m, n)$ is the actual cost that starts m^{th} state and arrives to n^{th} state.
  - $h(n)$ is just the estimate
- Expand the node with the lowest $f(n)$
- A mix of lowest-cost-first and greedy best-first search
- Select the node in the frontier with the lowest estimated distance from the start to a goal node constrained to go via that node

Properties

- If the heuristic $h(n)$ is admissible, then the solution found by A^* is optimal
- A heuristic $h(n)$ is admissible if it is never an overestimate of the cost of the cheapest path from node n to a goal node.
  - $0 \leq h(n) \leq h^*(n)$ for any state n.
  - $h^*(n)$ is the actual cost of the cheapest path from n to a goal node.
- Optimal Efficient
  - Among all optimal algorithms that start from the same start node and use the same heuristic, A^* expands the fewest nodes.

3.5 Design an Admissible Heuristic

3.5.1 Some Heuristic Functions for 8-puzzles

- Manhattan Distance Heuristic: The sum of the Manhattan distances of the tiles from their goal positions
- Misplaced Tile Heuristic: The number of tiles that are NOT in their goal positions.

3.5.2 Constructing an Admissible Heuristic

- Define a relaxed problem by simplifying or removing constraints on the original problem
- Solve the related problem without search
- The cost of the optimal solution to the relaxed problem is an admissible heuristic for the original problem

E.g. 8-puzzle: A tile can mover from square A to square B

- if A and B are adjacent, and
- B is empty

3.5.3 Choose a better Heuristic Functions

- Heuristic function must be admissible!
- Prefer a heuristic that is very different for different states. (state-specific)
- Wat a heuristic to have higher values (as close to h^* as possible)

3.5.4 Dominating Heuristic

Definition (dominating heuristic)
Given heuristic $h_1(n)$ and $h_2(n)$. $h_2(n)$ dominates $h_1(n)$ if

- $\forall\ n\ (h_2(n) \geq h_1(n)))$
- $\exists\ n\ (h_2(n) > h_1(n))$

Theorem
If $h_2(n)$ dominates $h_1(n)$, A^* using h_2 will never expand more nodes than A^* using h_1.

3.6 Pruning

3.6.1 Cycle Pruning

- A cycle cannot be part of a least-cost path
- Works well with depth-first search
- The complexity of cycle pruning is:
  - Constant time for depth-first search with adding one more bit to node (either explored or not)
  - Other methods needs to check multiple path at a time; when generating a neighbor, does this node already in the path (linear in the length of the path)

3.6.2 Multi-Path Pruning (more general)

- If we have already found a path to a node, we can prune other paths to the same node
- Cycle check is a special case for multi-path pruning
- Requires storing all nodes we have found paths to.
- For Lowest-cost-first search, it always explore the shorter path first so multi-path pruning will not prune the optimal answer.
- For A^* search, multi-path pruning may prune the optimal answer.

3.6.3 Find Optim. Solution with Multiple-Path Pruning

- Remove all the paths from the frontier the use the longer path
- Change the initial segment of the paths on the frontier to use the shorter path
- Make sure that the least-cost path to a node is found first

3.6.4 A^* search with mulitple-path pruning - optimal solution

- Ensure that we find the least-cost path to every node first
- Admissible heuristic guarantees the above for a goal node, but not for other nodes.
- We need the heuristic (to be consistent) to satisfy the monotone restriction (between any nodes):

  $$\text{for any acr(m,n)}, \quad h(m) - h(n) \leq cost(m, n)$$

  where $h(m) - h(n)$ is the heuristic estimate of the path cost from m to n.
  if n is a goal node, $h(m) \leq cost(m, n)$

If the heuristic satisfies the monotone restriction, A^* search with multiple-path pruning is optimal.
It is easy for us to come up a heuristic which is both admissible and consistent.

4 CSP

- Search algorithms are unaware of the internal structure of states.
  - Generate successors
  - test whether its a goal state
- However, knowing a state's internal structure can help
  - search algorithm so far: this is not a goal state; let's add more queens
  - smarter algorithm: this is a dead end, let's backtrack immediately
    - Can prune search tree & make search more efficient

Defining a CSP: Each state contains:

- A set X of variables: $\{X_1, X_2, \cdots, X_3\}$
- A set D of domains (contains all the possible value for a variable): D_i is the domain for variable X_i $\forall i$
- A set C of constraints specifying allowable value combinations.

A solution is an assignment of values to all variables that satisfy all the constraints.
e.g. 4-Queens Problem:
Variables: x_0, x_1, x_2, x_3, x_i is the row position of the queen in column i where $i \in \{0, 1, 2, 3\}$
(Assume that exactly one queen is in each column)
Domains: $D_i = \{0, 1, 2, 3\}$ for each x_i
Constraints: No two queens can be in the same row or in the same diagonal.

$$\forall i\ \forall j\ (i \neq j) \rightarrow ((x_i \neq x_j) \cap (|x_i - x_j|) \neq |i - j|))$$

4.1 Backtracking Search

Assumptions:

- place queens from left to right
- always ensure that constraints are satisfied
- State: one queen per column in the leftmost k columns with no pair of queens attacking each other
  - e.g.
  - _ _ _ _
  - 2 _ _ _
  - 2 0 _ _
  - 2 0 3 _
  - 2 0 3 1
- Initial state: no queens on the board
- Goal state: 4 queens on the board. No pair of queens are attacking each other
- Successor function: add a queen to the leftmost empty column such that it is not attacked by any other existing queen

4.2 Arc-Consistency

- Detect if one state is a dead-end before tracing its subtree.

4-Queens constraint Network
Start with $x_0 = 0$

- $x_2 = 1$ does not lead to a solution $\rightarrow$ no position for x_3
- $x_2 = 3$ does not lead to a solution $\rightarrow$ no position for x_1

Only necessary to include binary constraints:

- Unary constraints: some domains values are allowed and some are not allowed
  - Take each domain do some preprocessing (check if its value follows unary constraints), if no, then remove those value. $\rightarrow$ we are done
- Constraints with more variables: every constraints with more variables can be decomposed to binary constraints

A constraints graph has:

- Variables
- Constraints: Arc between variables

4.2.1 Definition

- An arc $(X, c(X,Y))$ is arc-consistent iff for every value v in D_x, there is a value w in D_Y such that (v,w) satisfies the constraint $c(X,Y)$

$$\forall\ v \in D_x\ \exists w \in D_y\ (v,w) \text{ satisfies } c(X,Y)$$

  e.g.
  $D_x = \{1,2\}\ D_y = \{1,2,3\}$, arc-consistent
  $D_x = \{1,2\}\ D_y = \{1,2\}$, not arc-consistent

  - If $(X, c(X,Y))$ is NOT arc-consistent, we can reduce the domain of X. This will not rule out any solution.
  - When consider $(Y, c(X,Y))$, the constraint is still the same! **Do not change it.**
  - Arc-consistency is not symmetric! When giving the counter-example, we cannot change the constraints.

4.3 AC-3 Algorithm

Algorithm 1 The AC-3 Algorithm

```
1: put every arc in the set $S$.
2: while $S$ is not empty do
3:     select and remove $\langle X, c(X,Y)\rangle$ from $S$
4:     remove every value in $D_X$ that doesn't have a value in $D_Y$
   that satisfies the constraint $c(X,Y)$
5:     if $D_X$ was reduced then
6:         if $D_X$ is empty then return false
7:         for every $Z \neq Y$, add $\langle Z, c'(Z,X)\rangle$ to $S$
   return true
```

- Order does not matter when select and remove arc from set S.
- Line 7: for every variable that is different from Y, we need to add back arcs to the set. The second variable involved in the constraint is the same variable that we just reduced from D_x.

Effect of Removing a Value on Arc Consistency

- After reducing the domain of the second variable, the arc may no longer be arc-consistent. We may need to re-visit the constraint.
- After reducing the domain of the first variable, the arc is still arc-consistent. No need to revisit the constraint.

Note:
Since we reduce x from D_x, therefore, it is no need to revisit the constraints where the first variable is x, but we do need to revisit the constraints that has a second variable is x.

Three possible outcomes of the arc consistency algorithm

- A domain is empty $\rightarrow$ No solution exists!
- Every domain has 1 value left. A unique solution!
- AC-3 does not give a definite answer: 1 value left and ≥ 1 domain has multiple values left

AC-3 algorithm guarantee to terminate

- Complexity of the AC-3 algorithm:
  - CSP has n variables, size of each domain $\leq d$, c binary constraints. Each arc can be added to the set d times (d values to delete). $A - C$ checked in $O(d^2)$ time.
  - We have number of c binary constraints and each binary constraint can be added in d times. Each time A-C algorithm needs to checked d^2 time for worst case (number of variables in X is d and number of variables in Y is d).
  - Put in together we have $O(c \cdot d \cdot d^2) = O(cd^3)$

5 Local Search

Reasons why Local Search

- Other search algos explore the space systematically and keep track of one or more paths
- The search space can be too big or infinite
- The path to a goal is irrelevant

Properties of local Search

- Explores only a portion of the search space
- Requires little memory (only a few states instead of a path)
- Advantages:
  - Can find solutions quickly on average
  - Works for CSPs and general optimization problems
- Disadvantages:
  - No guarantee that a solution will be found if one exists.
    - Because it does not search for the entire search space.
  - Cannot prove that no solution exists.
    - Only use if there exists or high likely exists a solution.

Steps:

- Start with a complete assignment of values to variables
  - Different from CSP which starts from an empty state and build through the algorithm, local search starts with a complete state with all the variables have been assigned different values.
- Take steps to improve the solution iteratively
  - Improve cost function or get closer to goal state

Formulation

- A state: a complete assignment to *all* of the variables.
- A neighbor relation: which states to explore next?
- A cost function: how good is each state? (minimize cost function)

e.g. 4-queen problem:
Variables: x_0, x_1, x_2, x_3 where x_i is the row position of the queen in column i. Assume that there is one queen per column $i \in \{0, 1, 2, 3\}$
Domains: $x_i \in \{0, 1, 2, 3\}\ \forall i$
Initial state: 4 queens on the board in random row positions.
Goal state: 4 queens on the board. No pair of queens are attacking each other.
Neighbor relation:

- Move a single queen to a different row in the same column
- Swap the row positions of two queens.

Cost function: the number of pairs of queens attacking each other, directly or indirectly.

5.1 Greedy Descent – minimize the cost function

- Start with a random state
- Move to a neighbour with the lowest cost, if it is better than the current state
  - Called greedy because it only checks the immediately neighbor
- Stop when no neighbour has a lower cost than current state
  - Current state is the best state in the neighborhood
- Only remembers the current node.
  - Need small memory

Properties of Greedy Descent

- Perform quite well in practice. Make rapid progress towards a solution.
- Give enough time, it will not find the global optimum. (only for the local optimal)

Global optimum vs Local optimum

- Local optimum: No neighbor has a lower cost
  - strict local optimum: strictly less than its neighbor cost
  - flat(shoulder) local optimum: less or equal than its neighbor cost
- Global optimum: A state that has the lowest cost among all the states

5.2 Escaping flat local optimum

- Sideway moves: allow the algorithm to move to a neighbour that has the same cost
- Tabu list: keep a small list of recently visited states ad forbid the algorithm to return to those states.

5.3 Greedy Descent with Randomization

Greedy descent can get stuck at a local optimum

- Random restarts: (global random move)
  - restart search in a different part of the search space.
  - e.g.: Greedy descent with random restart
- Random walks: (local random move)
  - Move to a state with a higher cost occasionally. (deal with a lot of local minimums)
  - e,g, simulated annealing

Steps:

- Performs multiple greedy descents from randomly generated initial states.
- It will find the global optimum with random restarts when giving enough time.
  - If giving enough time, it will generate the global optimum as the initial state randomly.

5.4 Simulated Annealing

- Start with a high temperature and reduce it slowly
- Choose a random neighbour
- if the neighbour is an improvement, move to it
- if the neighbour is not an improvement, move to the neighbour probabilistically depending on (or we do not move at all)
  - the current temperature T
  - how much worse is the neighbor compared to current state.

Probability to move a worse neighbour

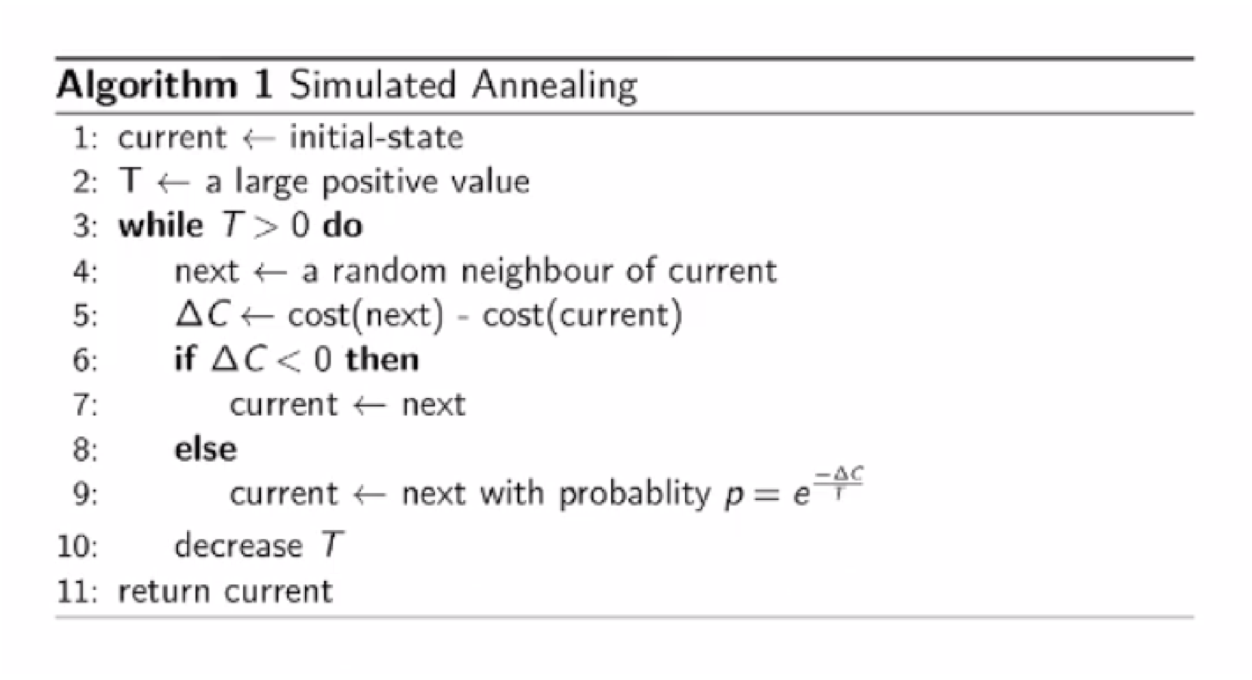
- A is the current state and A' is the worse neighbour.

- Let $\Delta C = cost(A') - cost(A)$. The current temperature is T We move to the neighbour A' with probability

  $$e^{-\dfrac{\Delta C}{T}}$$

  Gibbs distribution / Boltzmann distribution

Simulated Annealing Algorithm



Algorithm 1 Simulated Annealing

```
current ← initial-state
T ← a large positive value
while $T > 0$ do
    next ← a random neighbour of current
    $\Delta C$ ← cost(next) - cost(current)
    if $\Delta C < 0$ then
        current ← next
    else
        current ← next with probablity $p = e^{\frac{-\Delta C}{T}}$
    decrease $T$
return current
```

- We want to decrease the temperature every slowly
  - If the temperature decreases slowly enough, simulated annealing is guaranteed to find the global optimum with probability approaching 1
- In practice, a popular schedule is geometric cooling.

5.5 Genetic Algorithm

- Maintain a population of k states
- Randomly choose two states to reproduce. Probability of choosing a state for reproduction is proportional to the fitness of the state
- Two parent states crossover to produce a child state
- The child state mutates with a small probability
- Repeat the steps above to produce a new population
- Repeat until the stopping criteria is satisfied

5.5.1 Beam Search

Remember k states
Choose the k best states out of all of the neighbors
k controls space and parallelism

- When $k = 1$, we will update the state to its best neighbor.
- Beam search is choosing k neighbors as a whole while k random restarts in parallel chooses k random states independently.
- Communication between k states, which is not good for exploration.

5.5.2 Stochastic Beam Search

- Choose k states probabilistically
- Probability of choosing a neighbour is proportional to its fitness
- Maintains diversity in the population of states
  - More chances to explore the search space
- Mimics natural selection

5.5.3 Comparison between greedy descent and genetic algorithm

- Explore state spaces
  - Greedy descent: generates neighbors of the current state based on the neighbour relation

- Genetic algorithm: generating each state in the next population somewhat randomly (cross two parents states to produce a child state, the crossover can be random) – new state is the combination of two parents states with little possibility of mutation.
  - it also introduce randomness and potentially allow us to produce a state which is different from parent $\rightarrow$ increase diversity

- Optimize the quality of the population
  - Greedy descent: move to the best neighbor
  - Genetic algorithm: probabilistically choosing the relatively good states to reproduce
    - choose the parents based on its fitness
    - The higher the fitness and the lower the cost, the more likely these states are being chosen.
    - So good parents produce good/better children.

6 Machine Learning

- Learning is the ability of an agent to improve its performance on future tasks based on experience.
  - Do more → expand range of behaviour
  - Do things better → improve accuracy on tasks
  - Do things faster → improve speed
- Learning Architecture
  - Problem/Task
  - Experiences/Data
  - Background Knowledge/Bias
  - Measure of improvement
- Learning type:
  - Supervised Learning
    - Given input features, target features, and training examples. predict the value of the target features for new examples given their values on the input features
    - Classification: target features are discrete
      - e.g. sunny, cloudy or rainy
    - Regression: target features are continuous
      - e.g. temperatures, prices
  - Unsupervised Learning
    - Learning classifications when the examples do not have targets defined
    - provide inputs value and learn the pattern in the inputs value
  - Reinforcement Learning
    - Learning what to do based on rewards and punishments

6.1 Supervised Learning

Settings:

- Given training examples of the form $(x, f(x))$ where x is the input features, $f(x)$ is the target features
- Return a function h hypothesis that approximates the true function f

Learning as a search problem

- Given a hypothesis space, learning is a search problem
- Search space is prohibitively large for systematic search
- ML techniques are often some forms of local search

Generalization

- Goal of ML is to find a hypothesis that can predict unseen examples correctly
- Ochkam's razor:
  - It is an assumption
- Cross validation
- A trade-off between
  - complex hypotheses that fit the training data well
  - simpler hypotheses that may generalize better

6.1.1 Bias-Variance Trade-off

- Bias: how well can I fit the data with my learned hypothesis.
  - A hypotheses with high bias: make strong assumptions, too simplistic, has few degrees of freedom(few parameters for choices), does not fit the training data well.
  - Problems: Does not fit on untrained data well
- Variance: How much does the learned hypothesis vary given different training data
  - A hypotheses with high variance: has a lot of degrees of freedom. is very flexible. whenever the training data changes the hypothesis changes a lot. $\rightarrow$ fits the training data very well
  - Problems: We maybe fit the training data too well $\rightarrow$ does not fit on untrained data well (over-fitting)
- we want low bias and low variance

6.1.2 Cross Validation

- 4-fold cross validation.
  - Divide the dataset into equally four pieces
  - Using one pieces for validation but rest as training set
  - Once done four times, we have four values – test accuracy, test error
  - Take average of these errors $\rightarrow$ be the performance be the specific hypotheses
  - hypotheses have a set of parameters $\rightarrow$ need to choose value for those parameters
  - Take one set of parameters run 4-fold cross validation $\rightarrow$ choose the best performance (lowest average error)
- after cross validation
  - select one of the K trained hypotheses as the final hypotheses
  - train a new hypotheses on all of the data, using parameters selected by cross-validation

7 Decision Tree

- Simplest machine learning algorithm – supervised
- Structure:
  - Internal nodes: performing a boolean test on input features
  - Leaf nodes: labelled with value for the target features
- Build a decision tree:
  - determine the order of testing the input features
  - give an order of testing the input features, we can build a decision tree by splitting the examples.
  - When do we stop:
    * All the examples belong to the same class
    * There are no more features to test
    * There are no more examples
- which decision tree should be generated
  - which order of testing the input features should we use
    * the search space is too big for systematic search

        * solution: greedy (myopic) search
    - should we grow a full tree or not
        * A decision tree can represent any discrete function of input features
        * Need a bias, small tree? least depth? fewest node?

- Test feature multiple times:
    - With discrete features, if we are allowed to do multi-way splits, then we will test each feature at most once. If we are only allowed to do binary splits, then we might test a feature multiple times if it has multiple values.
    - A real-valued feature is like a discrete feature with many values. Therefore, if we are only allowed binary splits, it is often the case that we will test a feature multiple times for different split values.

7.1 No features to test

Why this happens?

- The data is noisy. Even if we know the values of all the input features, we still cannot make a deterministic decision. The target features may be influenced by an unobserved input feature.

Resolve this issue

- Predict a majority class or make a probabilistic decision

7.2 No example left

Why this happens?

- A combination of features is not present in the training set. If we never observe this pattern, we do not know how to predict it.

Resolve this issue

- Use the examples at the parent node.
- Making a majority decision or a probabilistic decision.

7.3 Which feature should we test at each step

- Create a shallow/small tree $\rightarrow$ need to minimize the number of tests
- Finding the optimal order of testing feature is difficult
    - Greedy search – make the best myopic choice at each step

  - i.e. at each step, test a feature that makes the biggest difference to the classification

- reduce uncertainty as much as possible
- Information content of a feature = uncertainty before testing the feature - uncertainty after testing the feature

Entropy – measure uncertainty
Given a distribution $P(c_1), \cdots, P(c_k)$ over k outcomes $c_1, \cdots, c_k$, the entropy of the distribution is

$$I(P(c_1), \cdots, P(c_k)) = -\sum_{i=1}^{k} P(c_i) \log_2(P(c_i))$$

Consider a distribution $(p, 1-p)$ where $0 \leq p \leq 1$.

- Maximum entropy of distribution: p = 0.5, I(0.5, 0.5) = 1
- Minimum entropy of distribution: p = 0 or p = 1, I(1, 0) = 0

7.4 Expected information gain

- The feature has k values $v_1, \cdots, v_k$
- Before testing the feature, we have p positive and n negative examples
- After testing the feature, for each v_i of the feature, we have p_i positive and n_i negative examples
- Entropy before testing the feature: $H_{before} = I(\frac{p}{p+n}, \frac{n}{p+n})$
- Expected entropy after testing the feature:

$$H_{after} = \sum_{i=1}^{k} \frac{p_i + n_i}{p+n} \times I(\frac{p_i}{p_i+n_i}, \frac{n_i}{p_i+n_i})$$

- Expected information gain (entropy reduction) is $InfoGain = H_{before} - H_{after}$
- We are looking for H_{after} as lowest as possible, so we can maximize InfoGain

7.5 Real-Valued Features

Handling a discrete feature

- Binary splits only
  - tree is simpler and more compact
  - tree tends to be deeper
- Allow multi-way splits
  - tree becomes more complex than if-then-else
  - tree tends to be shallower
  - info gain prefers a variable with a larger domain
- Discretize the feature
  - pro: easy to do
  - cons: lose valuable info; may make tree complex
- Always do binary tests. Dynamically choose a split point
  - multi-way splits problematic with unbounded domain.
  - Binary splits may test the feature many times. The gree may become much deeper.
  - Choose two sets with different outcomes (yes/no) and splitting points between these sets (even if one set contains yes and no and the other set only has yes, then it is still a possible splitting points since we can pick one side contains no as no and the other side as yes).
    Then calculate all the possible splitting points' infoGain. Get the maximum infoGain and start the split.

7.6 Over-fitting

- Problem: Growing a full tree is likely to lead to over-fitting

Strategies to prevent over-fitting

- Minimum examples at a node (training error reduce by more than a threshold)
- Information gain must be above a threshold
- Maximum depth (max number of nodes in the tree)
- Post-prune the tree

- Have multiple features working together might be formative, but one of them is not working very formative
- if post-prune the tree, we probably have tested the feature regarding both bits already. And compute the test error, if a lot of error then we will not prune the branch

8 Neural Network

8.1 Introduction

- Deal with complex inputs and outputs relationship

8.2 Neuron

- A linear classifier – it ”fires” when a linear combination of its inputs exceeds some threshold
- it computes the linear combination of its input and if the results exceed threshold, then it outputs the output signal.
- Neuron j computes a weighted sum of its input signals. $in_j = \sum_{i=0}^{n} w_{ij} a_i$
- Neuron j applies an activation function g to the weighted sum to derive the output. $a_j = g(in_j) = g(\sum_{i=0}^{n} w_{i,j} a_i)$
- The reason why we need to have Bias Weight as one of the input is that we want to represents a more general linear function, thus it is possible to have a constant term that is Bias Weight.

8.2.1 Activation Function

- be non-linear
  - combining linear functions (different neuron has different linear function) will not produce a non-linear function
  - Complex relationships are often non-linear
- mimic the behavior of real neuron
  - If the weighted sum of input signals is large enough, the neuron fires. Otherwise, it does not fire
- be differentiable almost everywhere
  - learn a neural network using gradient descent, which requires the activation function to be differentiable
- Common activation functions
  - step function: $g(x) = 1$ if $x > 0$. g(x) = 0 if $x \leq 0$
    - simple to use, not differentiable, not used in practice, but useful to explain concepts

- Sigmoid function: $g(x) = \frac{1}{1+e^{-kx}}$
  - can approximates the step function,
  - It is a clear and bounded prediction (0,1).
  - It is differentiable
  - disadvantage:
    - vanish gradient problem: the gradient of function changes very little when x is either really large or really small (does not learn when x is very large or very small)
    - computationally expensive
- Rectified linear unit (ReLu): $g(x) = max(0, x)$
  - computationally efficient
  - non-linear and differentiable
  - only fix the positive side vanish gradient problem
  - The dying ReLu problem: the negative side gradient is 0 $\rightarrow$ does not learn at negative side.
- Leaky ReLu: $g(x) = (0.1 * x, x)$
  - Enables learning for negative input values $\rightarrow$ solve the vanish gradient problem
  - + all the advantages from ReLu

8.3 Neuron Network

- Feed-forward network:
  - no loop in the network, one flow only
  - directed acyclic graph
  - The output is the function of its input
- Recurrent network:
  - has loop
  - has memory – better model of human brain
  - the output is the function of its input + the historical input

8.3.1 Perceptrons

- Single-layer feed-forward neural network
- The inputs are connected directly to the outputs

- Can represent logical functions, e.g. AND, OR, and NOT

Mathematical ways to find the weight
Note that: it only works for Perceptrons.
Firstly, we list all the data points, say x_0, x_1, x_2, and all the weights associated between different data points, say w_{01}, w_{11}, w_{21}. Then we list the actual output, say O_{actual} and the expected output, say $O_{expected}$.
Secondly, we start assign all weight to 0 first (randomly assigned) and give all the data points 1

- if the actual output is less than expected output, we increase the weight associated to the data points whose value is greater than 0
- if the actual output is greater than expected output, we decrease the weight associated to the data points whose value is greater than 0

8.4 Limitations of Perceptrons

- XOR cannot be represented using perceptrons
- since a perceptron is a linear classifier. XOR is not linearly separable

Hidden Layer

- layers that is not visible.

9 Gradient Descent

- A local search algorithm to find a minimum of a function
- It is a iterative method
  - function: error $\rightarrow$ expected outputs based on training data and the actual outputs based on training data and weighted network
- Steps of the algorithm:
  - Initialize weights randomly
  - Change each weight in proportion to the negative of the partial derivative of the error with respect to the weight.

$$w = w - \sum \eta \frac{d_{error}}{d_w}$$

  - η is the learning rate, we sum all of error from the training data that is how much we will decrease by

    - Terminate after some number of steps when the error is small or when the changes get small

- In terms of direction: Change x in the direction of the negative of the partial derivative
- In terms of step size: Take a step in proportion to the magnitude of the partial derivative

9.1 Update weights based on the data points

- Gradient descent updates the weights after sweeping through all the examples
- To speed up learning, update weights after each example
    - Incremental gradient descent (systematically choose example)
        * Only look at one example, calculate the changes for all the weights and update weights immediately $\rightarrow$ update weights after each example
    - Stochastic gradient descent (randomly choose example)
        * choose the example randomly and update weights after that
- Pros and Cons
    - Pros: Update more frequently, each step is cheaper and more accurate
    - cons: Update every step, the process cannot guarantee to converge
- Trade off learning speed and convergence
    - Batched gradient descent
        * Update the weight after a batch of examples
        * Start with small batches to learn quickly, then increase batch size so it converge
        * We can define the size of batch if batch = 1, then it is equal to incremental gradient descent

10 Back-propagation Algorithm

- Learn the weights in a neural network by using gradient descent
- For each training example (x_1, x_2, y_1, y_2), perform 2 passes.

  - Forward pass: compute o_1, o_2 given x_1, x_2 and the weights

$$h_j = g(\sum_{i=0}^{2} w1_{ij}x_i), \text{ for } j = 1, 2 \rightarrow \text{ calculate hidden nodes}$$
$$o_j = g(\sum_{i=0}^{2} w2_{ij}h_i), \text{ for } j = 1, 2 \rightarrow \text{ calculate output nodes}$$

    where g is the activation function(sigmoid function)
  - Backward pass: calculate the partial derivative of the error with respect to each weight.

- Update each weight by the sum of changes for all the training examples.

11 NN vs Decision Trees

When use NN

- High dimensional or real-valued inputs, noisy (sensor) data
- Form of target function is unknown (no model)
- Not important for humans to explain the learned function
  - Model tends not to be interpretable

When not to use NN

- Difficult to determine the network structure (number of layers, number of neurons)
- Difficult to interpret weights, especially in multi-layered networks
- Tendency to over-fit in practice (poor prediction outside of the range of values it was trained on)

Decision tree vs NN

- Data type
  - NN: high-dimensional data (e.g. images, audio and even text)
  - DT: tabular(simpler) data (e.g. jeeze data set)
- Size of data set
  - NN: needs a lot of data set (since it is easily to be over-fit, so we need great amount of data to train it well)

  - DT: works well with little data

- Form of target function
  - NN: powerful enough to model any arbitrary function
  - DT: nested if-then-else statement

- The architecture (the hyper-parameters in the model)
  - NN: a lot of parameters: number of layers, number of neurons per layer, which activation function do we use, what learning rate for gradient descent
  - DT: grow full tree $\rightarrow$ no parameters; prevent over-fitting: we add parameters such as maximum depth or the maximum number of nodes

- Interpret the model that learned
  - NN: not interpretable $\rightarrow$ kind of a black box
  - DT: interpretable (e.g. apply credit card, decision making for loan, government etc)

- Time available for training and classification
  - NN: hard to train a model $\rightarrow$ usually takes long time; also takes long time for classification since there are a lot of computation in the model
  - DT: really fast on training and classification

Note:

- NN cannot solve all the problems; not always the best choice.
- Try the simpler model first; might do as well as complicated NN model

12 Intro to Uncertainty

- Why need to handle uncertainty?
  - An agent may not observe everything in the world
  - An action may not have its intended consequences
- An agent needs to
  - Reason about their uncertainty
  - Make a decision based on their uncertainty

12.1 Probability

- Probability is the formal measure of uncertainty
- Two campus: Frequentists and Bayesians
- Frequentists' view of probability
  - Frequentists view probability as something objective
  - Compute probabilities by counting the frequencies of the events
  - Disadvantages: if observe nothing, then no probability will come up
- Bayesians' view of probability
  - Bayesians view probability as something subjective
  - Probabilities are degree of belief
  - Start with prior belief and update belief based on new evidence

12.1.1 Random Variable

- A random variable
  - Has a domain of possible values
  - associated probability distribution, which is a function from the domain of the random variable to [0,1]

12.1.2 Shortern notation

- P(A) denotes P(A = true)
- P($!A$) denotes P(A = false)

12.1.3 Axioms of Probability

- $0 \leq P(A) \leq 1$
- P(true) = 1, P(false) = 0
- $P(A \cup B) = P(A) + P(B) - P(A \cap B)$

12.1.4 Joint Probability Distribution

- A probabilistic model contains a set of r.v.
- An atomic event assigns a value to every random variable in the model
- A joint probability distribution assigns a probability to every atomic event

12.1.5 Prior and Posterior Probablities

P(X)

- prior or unconditional probability
- Likelihood of X in the absence of any other information
- Based on the background information

P(X|Y)

- posterior or conditional probability
- Likelihood of X given Y
- Based on Y as evidence

12.2 The Sum and Product Rules

12.2.1 The Sum Rule

- Compute the probability over a subset of the variables
  - Using sum rule: sum out every variable that we do not care about
    - Given a joint distribution over A, B and C, we can calculate $P(A \cap B)$ by summing out C

$$P(A \cap B) = P(A \cap B \cap C) + P(A \cap B \cap !C)$$

12.2.2 The Product Rules

- How to compute P(A|B)

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)}$$

$$P(A \mid B) + P(!A \mid B) = 1$$

$$P(A \mid B)P(B) = P(A \cap B)$$

- we only care about B is true, thus we are ruling out all the possibilities where B is false

12.3 Chain Rule and the Bayes' Rule

12.3.1 Chain Rule

- For Two variables: $P(A \cap B) = P(A|B) * P(B) = P(B|A) * P(A)$
- For Three variables: $P(A \cap B \cap C) = P(A|B \cap C) * P(B|C) * P(C)$
- For any number of variables:

$$\begin{aligned} & P(X_n \cap X_{n-1} \cap X_{n-2} \cap \cdots \cap X_2 \cap X_1) \\ =& \Pi_{i=1}^{n} P(X_i|X_{i-1} \cap \cdots \cap X_1) \\ =& P(X_n|X_{n-1} \cap \cdots \cap X_2 \cap X_1) * \cdots * P(X_2|X_1) * P(X_1) \end{aligned}$$

12.3.2 Bayes' rule

- $P(X|Y) = \frac{P(Y|X)*P(X)}{P(Y)}$
- $P(X|Y) = \frac{P(Y|X)P(X)}{P(Y|X)P(X)+P(Y|!X)P(!X)}$
- $P(!X|Y) = \frac{P(Y|!X)P(!X)}{P(Y|X)P(X)+P(Y|!X)P(!X)}$
- calculate $P(Y|X)P(X)$ and $P(Y|!X)P(!X)$
- normalize the two values so that they sum to 1
- so we only need to know $P(Y|X)$, $P(X)$ and $P(Y|!X)$, then we can calculate $P(X|Y)$ and $P(!X|Y)$

13 Definition of Independence and Conditional Independence

13.1 Unconditionally Independent

- X and Y are independent iff
  - $P(X|Y) = P(X)$
  - $P(Y|X) = P(Y)$
  - $P(X \cap Y) = P(X)P(Y)$
- Learning Y does NOT influence belief about X

13.2 Conditional Independent

- X and Y are conditionally independent given Z if
  - $P(X|Y \cap Z) = P(X|Z)$
  - $P(Y|X \cap Z) = P(Y|Z)$
  - $P(Y \cap X|Z) = P(Y|Z)P(X|Z)$
- Learning Y does NOT influence the belief about X if Z is known

13.3 Deriving a Compact Representation of a Joint Distribution

- n random variables $\rightarrow 2^n - 1$ probabilities
  - since we can use 1 - those probabilities to get the last one

14 Bayesian Network

Applications

- Medical diagnosis of diabetes
- Nuclear system/ Fire alarm

Motivation

- We can compute any probability using the joint distribution
  - Quickly become intractable as the number of variables grows
  - Unnatural and tedious to specify all the probabilities

Definition
A Bayesian Network

- is a compact version of the joint distribution, and
- takes advantage of the unconditional and conditional independence among the variable

Components

- Node: represents a random variable
- X is a parent of Y if there is an arrow from node X to node Y
- Each node X_i has a conditional probability distribution $P(X_i|Parents(X_i))$ that quantifies the effect of the parents on the node

14.1 The Semantics of Bayesian Network

- Two ways for understanding
  - A representation of the joint probability distribution
  - An encoding of the conditional independence assumption
- Representing the joint distribution

$$P(X_n \cap \cdots X_1) = \Pi_{i=1}^{n} P(X_i|Parents(X_i))$$

15 Bayesian Network Cont.

15.1 Two types of independence Relationships

15.2 Independence Relationships in Three Key Structures

- A $\rightarrow$ B $\rightarrow$ C
  - B is not observed, A and C dependent
  - B observed, A and C conditionally independent
- One (A) and Two (B and C) out
  - A is not observed, B and C dependent
  - A is observed, B and C conditionally independent
- Two (B and C) in One (A)
  - A is not observed, B and C unconditional independent

  - A is observed, B and C dependent

- A node is conditionally independent of its non-descendants given its parents
- A node is conditionally independent of all other nodes given its Markov blanket
  - The Markov blanket of a node consists of its parents, its children and its children's parent

15.3 D-Separation

- A set of variables E d-separates X and Y, if E blocks every undirected path between X and Y in the network
- If E d-separates X and Y, then X and Y are conditionally independent given E
- Block
  - $X \to \cdots \to E \to \cdots \to Y$
    * if E is observed, then it blocks the path between X and Y
  - $X \leftarrow \cdots \leftarrow E \to \cdots \to Y$
    * if E is observed, then it blocks the path between X and Y
  - $X \to \cdots \to E \leftarrow \cdots \leftarrow Y$
    * If E and $E's$ descendants are not observed, then they block the path between X and Y
    * If we observe at least one of $E's$ descendants or E, then they does not block the path between X and Y
- Normally X and Y are connected by several paths. We need to check every path and to see if X and Y have been blocked. They are conditionally independent if and only if all the paths between them have been blocked.

15.4 Construct a Bayesian Network

- For a joint probability distribution, there are multiple correct Bayesian networks
- A Bayesian network is correct if every independence relationship the network represents is correct
- We prefer one Bayesian network over another one if the former requires fewer probabilities
  - Small Bayesian Network is more compacted and easy to work with
- Steps:

1. Determine the set of variable for the domain
2. Order the variables $\{X_1, \cdots, x_n\}$
3. For each variable X_i in the ordering
   - Choose the node's parents:
   - Choose the smallest set of parents from $\{X_1, \cdots, X_{i-1}\}$ such that given Parents(X_i),X_i is independent of all nodes in $\{X_1, \cdots, x_{i-1}\} - Parent(X_i)$. Formally, $P(X_i|Parents(X_i)) = P(X_i|X_{i-1} \cap \cdots \cap X_1)$
   - Create a link from each parent of X_i to the node X_i
   - Write down the conditional probability table $P(X_i|Parents(X_i))$

- NOT all the link has a cause relationship, but has a correlation
- If causes precede effects, we get a more compact Bayesian Network
  - always try to add the causes first to the network before adding corresponding effect

16 Variable Elimination

Notation – $P(B|w \cap g)$

- Query variables: B
- Evidence variables: W and G
- Hidden variables: A, E and R

Shorthand notation – $P(B|w \cap g)$:

- B can be true and false – uppercase letter
- b represents B is true – lowercase letter
- $!b$ represents B is false – lowercase letter

Intro

- Performing probabilistic inference is challenging
  - Calculating hte posterior distribution of one or more query variables given some evidence is NP
  - No general efficient implementation available
- Exact and approximate inferences

- The variable elimination algorithm uses dynamic programming and exploits the conditional independence
  - VEA = factor + operations on factors (restrict, sum out, multiply, normalize)

Factors

- A function from some random variables to a number
- A factor can represent a joint or a conditional distribution
- Define a factor for every conditional probability distribution in the Bayes network

Restrict a factor

- Restrict a factor by assigning a value to the variable in the factor
  - For each observed variable, restrict the factor to the observed value
  - Restricting $f(X_1, X_2, \cdots, X_j)$ to $X_1 = v_1$, produces a new factor $f(X_1 = v_1, X_2, \cdots, X_j)$ on $X_2, \cdots, X_j$
  - $f(X_1 = v_1, X_2 = v_2, \cdots, X_j = v_j)$

Sum out a variable

- Sum out a variable
- Summing out X_1 with domain $\{v_1, \cdots, v_k\}$ from factor $f(X_1, \cdots, X_j)$, produces a factor on $X_2, \cdots, X_j$ defined by:

$$(\sum_{X_1} f)(X_2, \cdots, X_j) = f(X_1 = v_1, \cdots, X_j) + \cdots + f(X_1 = v_j, \cdots, X_j)$$

Multiplying factors

- Multiply two factors together
- The product of factors $f_1(X, Y)$ and $f_2(Y, Z)$, where Y are the variables in common, is the factor $(f_1 \times f_2)(X, Y, Z)$ defined by:

$$(f_1 \times f_2)(X, Y, Z) = f_1(X, Y) * f_2(Y, Z)$$

Normalize a factor

- Convert it to a probability distribution
- Divide each value by the sum of all the values

17 Markov Chain

17.1 Transition Model

- In general, every state may depend on all the previous states
- Problem: As t increases, the conditional probability distribution can be unboundedly large
- Solution: Let the current state depend on a fixed number of previous states
- K-order Markov Chain
  - First order Markov process: $P(X_t|X_{t-1} \cap X_{t-2} \cap \cdots \cap X_1) = P(X_t|X_{t-1})$
  - Second order Markov process: $P(X_t|X_{t-1} \cap X_{t-2} \cap \cdots \cap X_1) = P(X_t|X_{t-1} \cap X_{t-2})$
- The Markov assumption: The future is independent of the past given present

17.2 Stationary Process

- The dynamics does not change over time
- The conditional probability distribution for each time step remains the same
- Advantage:
  - easy, only need to specify one for the entire model
  - Natural choice, the dynamic usually not change
  - use a finite number of parameters to generate an infinite networks

17.3 Sensor Model

- Sensor Markov assumption: Each state is sufficient to generate its observations

  $$P(E_t|X_t \cap X_{t-1} \cap \cdots \cap X_1 \cap E_{t-1} \cap E_{t-2} \cap \cdots \cap E_1) = P(E_t|X_t)$$

  For E_t to be observable variable at time t, X_t to be unobservable variable at time t
- Hidden Markov Model
  - Contains a Markov chain: which only depends on the previous state, future state and past state is independent given current state, each state is unobservable
  - Each state generates a observable noisy signal

17.4 Inference Tasks

- Filtering: which state am I right now?
  - $P(R_t|U_{1:t})$
- Prediction: which state will I be in tmw?
  - $P(R_k|U_{1:t}) \quad k > t$
- Smoothing: which state was I in yesterday?
  - $P(R_k|U_{1:t}) \quad 1 \leq k < t$
- Most likely explanation: Which sequence of states is most likely to have generated the observations
- Algorithms:
  - The forward-backward algorithm: filtering and smoothing
  - The Viterbi algorithm: most likely explanation

17.5 Smoothing

- The purpose of doing smoothing is that once we made new observations, the new observations can help us to get a more accurate estimate in the past

18 Decision Theory

- Decision theory = probability theory(what should the agent believe based on the evidence) + utility theory(what does agent want)
- The principle of maximum expected utility:
  - A rational agent should choose the action that maximizes the agent's expected utility
- Decision networks = Bayesian network + actions + utilities
  - Chance nodes: represent random variables (no control over)
  - Decision nodes: represent actions (decision variables)
  - Utility nodes: represents agent's utility function on states (happens in each state) – needs to add utility function(maps to a state of the world to a number)
- Choose an action:
  - Set evidence variables for current state
  - For each possible value of decision node
    - set decision node to that value
    - calculate posterior probability or parent nodes of the utility node
    - calculate expected utility for the action
  - Return action with highest expected utility
- VEA for a single-stage Decision Network
  - Prune all the nodes that are not ancestors of the utility node
  - Sum out all chance node
  - For the single remaining factor, return the maximum value and the assignment that gives the maximum value
- Policy
  - A policy specifies what the agent should do under all contingencies
  - For each decision variable, a policy specifies a value for the decision variable for each assignment of values to its parents
- Solving decision problem:
  - Compute the expected utility of each policy, and choose the policy that maximize the expected Utility (even we have a small network, we can have large number of policies that we cannot handle properly)

  - Use the variable elimination algorithm

- VEA for DN:
  - Remove all variables that are not ancestors of the utility node.
  - Create factors
  - While there are decision nodes remaining
    - Sum out each random variable that is not a parent of decision node.
    - Find the optimal policy for the last decision
  - Return the optimal policies

19 Markov Decision Process

- Finite-stage vs ongoing problems
  - infinite horizon: the process may go on forever
  - indefinite horizon: the agent will eventually stop, but it does not know when it will stop
- Utility at the end vs a sequence of rewards
  - may not make sense to consider only the utility at the end, because the agent may never get to the end
  - the rewards incorporate the cost of actions and any rewards/punishments
- A Markov Decision Process
  - A set of states
  - A set of actions
  - transition probabilities: $P(s'|s,a) \rightarrow$ the probability of going to next state, s', given current state s and action a
    - it remains the same for each time step
  - reward function $R(s,a,s')$

19.1 Choose reward function

- Rewards: $R(s)$: the reward of entering state s
  - Total reward: $R(s_0)+R(s_1)+\cdots$
    - if time stamp is infinite, then we have infinite total reward; no way to tel which reward is better when it is time step approach to infinite

- Average reward:
$$\lim_{n\to+\infty}\frac{1}{n}(R(s_0)+R(s_1)+\cdots)$$
    - the average reward is zero, when total reward is finite, but time step is infinite
- Discounted reward: $R(s_0)+\gamma R(s_1)+\gamma^2 R(s_2)+\cdots$ where $0\le\gamma\le 1$ is the discount factor
    - widely used in game theory
    - we prefer to receive reward sooner rather than later
    - the tomorrow might not be coming
    - the total discounted reward is finite

19.2 Variations of MDP

- Full-observable MDP
    - each state is fully observable, agent knows what state it is in
- Partial observed MDP (POMDP) – combines a MDP and a hidden Markov model
    - The agent does not know what state it is in, but it can get some noisy signal of the state

19.3 Policies

- A policy specifies what the agent should do as a function of the current state
- has to specify for every possible state what should agent do, in a particular state, what action should agent take
- A policy is
    - non-stationary if it is a function of the state and the time
    - stationary if it is a function of the state

19.4 The expected Utility of a Policy

- $R(s)$: one-time reward of entering state s
- $V^\pi(s)$: expected utility of entering state s and following the policy π thereafter
- $V^*(s)$: expected utility of entering state s and following the optimal policy π^* thereafter
- expected utility: the expected value of total discounted reward

19.5 Calculate the optimal Policy Given $V^*(s)$

- $Q^*(s,a) = \sum_{s'} P(s'|s,a)V^*(s')$
  - that is the expected utility in state s and take action a
- $\pi(s) = arg\, max_a Q^*(s,a)$

19.6 Bellman Equation

- V and Q are defined recursively in terms of each other
  - $V^*(s) = R(s) + \gamma max_a Q^*(s,a)$
  - $Q^*(s,a) = \sum_{s'} P(s'|s,a)V^*(s')$
- Bellman Equation:
  - $V^*(s)$ are the unique solution to the Bellman equations

$$V^*(s) = R(s) + \gamma max_a \sum_{s'} P(s'|s,a)V^*(s')$$

19.7 The Value Iteration Algorithm

Let $V_i(s)$ be the values for the ith iteration

- Start with arbitrary initial values for $V_0(s)$
- At the i-th iteration, compute $V_{i+1}(s)$ as follows.

$$V_{i+1}(s) \leftarrow R(s) + \gamma max_a \sum_{s'} P(s'|s,a)V_i s'$$

  where, $V_i s'$ is always the old value.

- Terminate when $max_s|V_i s - V_{i+1}(s)$ is small enough (converge)

If we apply the Bellman update infinitely often, the $V_i's$ are guaranteed to converge to the optimal values.

Observation from Value Iteration

- Each state accumulates negative rewards until the algorithm finds a path to the +1 goal states.
- synchronous: store and use $V_i(s)$ to calculate $V_{i+1}(s)$
- asychronous: stores $V_i(s)$ and update the values one at a time, in any order

  - suggests that if there are certain states that are more promising, maybe update the value more frequently so that they will converge faster
  - make sure to update the estimate for each state a sufficient number of times so that all of the values will converge

20 Policy Iteration

- Deriving the optimal policy does not require accurate estimates of the utility function $V^*(s)$
- Policy iteration alternates between two steps
  - Policy evaluation: Given a policy π_i, calculate $V^{\pi_i}(s)$, which is the utility of each state if π_i were to be executed
  - Policy improvement: Calculate a new policy π_{i+1} using V^{π_i}
- Terminates when there is no change in the policy

Compare with value iteration

- value iteration: $V_1(s) \to V_2(s) \to \cdots \to V^*(s) \to \pi^*(s)$
- policy iteration: $\pi_1(S) \to V^{\pi_1}(s) \to \pi_2(s) \to V^{\pi_2}(s) \to \cdots \to \pi^*(s) \to V^*(s) \to \pi^*(s)$

20.1 Computation

- Policy improvement: $V^{\pi_i}(s) \to \pi_{i+1}(s)$

$$\pi_{i+1}(s) = argmax_a \sum_{s'} P(s'|s,a)V^{\pi_i}(s')$$

- Policy evaluation: $\pi_i(s) \to V^{\pi_{i+1}}(s)$

$$V_i(s) = R(s) + \gamma \sum_{s'} P(s'|s,\pi_i(s))V_i(s')$$

  - it is much easier to solve policy evaluation since we just need to solve the linear equation
  - Solve the system of linear equations exactly using standard linear algebra techniques – for n states this will take $O(n^3)$ time – more accurate
  - Solve the system of linear equations approximately by performing a number of simplified value iteration steps — approximately

20.2 Passive Reinforcement Learning

- between supervised learning and unsupervised learning
- have some feedback, but not all the examples
- want to learn how to perform well in the environment to maximum our total expected reward
- no reward function $R(s)$ nor transition probabilities $P(s'|s, a)$

20.2.1 Direct Utility Estimating

Estimating $V^*(s)$

- $V^\pi(s)$ is the expected total discounted reward from state s onward
- Each trial provides a sample of $V^\pi(s)$ for each state visited on the trial
- $V^\pi(s)$ is the average reward in all samples for s

Pros and cons

- pros: simple. reduced Reinforcement learning to supervised learning
- cons: missed the fact that the V^π values for different states are not independent. They are related by the Bellman equations; they converge really slow.

20.2.2 Adaptive Dynamic Programming

- Learn $V^\pi(s)$ via Bellman Equation

$$V_i(s) = R(s) + \gamma \sum_{s'} P(s'|s, \pi_i(s))V_i(s')$$

- reward function $\rightarrow$ observed reward
- transition probabilities
  - supervised learning;
  - input: state-action pair, output: resulting state

20.2.3 Temporal Difference

- use observed transition to adjust the utilities so that utility values agree with the bellman equations
- calculate the difference between the current estimate on the left and the target value on the right, then use the difference to decide the update rule
- not always increase or decrease – depends on how fast we want to adjust these utility value
- $V^{\pi}(s) = R(s) + \gamma V^{\pi}(s')$
- LHS = current estimate, RHS = target estimate
- Update $V^{\pi}(s)$ as follows

$$V^{\pi}(s) \leftarrow V^{\pi}(s) + \alpha(R(s) + \gamma V^{\pi}(s') - V^{\pi}(s))$$

- α is the learning rate
  - depends on how much the progress in the algorithm
  - learning rate should decreases as the number of times visiting the state increasing $\rightarrow$ help to converge to some value for utility estimates
- Determine both magnitude and direction of update
- need reward $R(s)$ but not transition probability, since we use the observed transition directly rather than estimate all the transition probabilities

20.3 Active Reinforcement Learning

- An active agent must decide on what policy it should follow
  - learn the best policy and the best expected utilities

20.3.1 Passive ADP

- Learn the reward function $R(s)$ through the observed rewards
- Learn the transition probabilities $P(s'|s, a)$ for the policy π
- Solve for $V^{\pi}(s)$ using the simplified Bellman equations

$$V^{\pi}(s) = R(s) + \gamma \sum_{s'} P(s'|s, \pi(s))V^{\pi}(s')$$

- Learn the transition probabilities $P(s'|s,a)$ for all (s,a)
- Learn the values of $V^*(s)$ the expected utilities of the optimal policy for all the states
- Agent should not follow the learned model
  - if followed, agent does not learn accurate utility values
  - if followed, agent does not learn the tru optimal policy
  - since learned model is different from true model, sometimes the optimal policy of the learned model is much worse than the optimal model policy of the true model

20.3.2 Exploration v.s. Exploitation

- Exploitation: Follow the optimal policy of the current learned model
- Exploration: Take actions to improve the current learned model
- reason to explore:
  - take an action will help to improve the model and learn the true model
  - improve model to help agent to get a better reward in the future
- Pure exploration:
  - always see other choice without applying our knowledge
- Pure exploitation:
  - might stuck in a bad policy due to an inaccurate learned model

20.3.3 Bandit Problems

- The trade of between Exploration and Exploitation

GLIE Scheme

- must try each action in each state an unbounded number of times
- the agent will eventually learn the true model and must eventually act in a greedy ways
- The update rule for value iteration:

$$V^+(s) \leftarrow R(s) + \gamma max_a f\big(\sum_{s'} P(s'|s,a)V^+(s), N(s,a)\big)$$

  - $V^+(s)$ is the optimistic estimate of the utility of the state s

        - $V^*(s)$ is the actual accurate estimate of utilities value if we follow by the optimal policy
    - $N(s,a)$ is number of times action a has been tried in state s
    - $f(u,n)$ is the exploration function, trading off preference state for high values of u(utility value) and preference for low values of n (the state that we have not visited mush)
        - That is prefer the state that agent has not tried very often
        - prefer actions that are of high utility
        - f is increasing as u increasing and n decreasing
        - $f(u,n) = \begin{cases} R^+ \text{ if } n < N_e \\ u \text{ otherwise} \end{cases}$ The agent will try each state-action pair at least N_e times.
        - u is worse than R^+

The Passive TD Agent
When a transition occurs from state s to s', we update V^π as follows.

$$V^\pi(s) \leftarrow V^\pi(s) + \alpha(R(s) + \gamma V^\pi(s') - V^\pi(s))$$

- α is the learning rate. α should decrease as the number of times a state has been visited increases
- $R(s) + \gamma V^\pi(s')$ is the target value of $V^\pi(s)$ based on this transition

The Active TD Agent

- Learn the utility values $V(s)$ by using the TD update rule

$$V^\pi(s) \leftarrow V^\pi(s) + \alpha(R(s) + \gamma V^\pi(s') - V^\pi(s))$$

- Learn the transition probabilities for all state-action pairs
- Determine the optimal policy using the utility values and the transition probabilities

$$\pi^*(s) = arg\ max_a \sum_{s'} P(s'|s,a)V^*(s')$$

Q-learning
Agent learns an action-utility representation instead of learning the utilities directly

- An action-utility representation $Q'(s,a)$
    - $Q'(s,a)$ obtains $R(s)$ and the expected total discounted reward starting from the next state

  - $Q(s, a)$ that is the expected total discounted reward starting from the next state

- The utilities of the states $V(s)$
- the optimal policy given $Q'(s, a)$ is $\pi^*(s) = arg\ max_a Q'(s, a)$
- Q-learning does not require model and know transition probability