CS2109S

INTRO (AI)

Agent

- Perceives environment through **sensors**
- Acts through **actuators**
- Completely specified by **agent function** mapping percept sequences to actions

Percept Info perceived through sensors

Percept sequence Complete history of perception

Rational agent

- Choose action that is expected to maximize performance measure
- Uses evidence from **percept sequence** and **built-in knowledge**

Omniscience A rational agent does the best it can given the knowledge it has. It may not be omniscient.

Autonomous If its behaviour is determined by its own experience

Performance measure

Best for who | What are we optimizing | What info is available | Unintended effects | Costs

Defining the problem

Examples given in the context of autonomous driving

1. Performance measure (safety, speed, legal, comfort)
2. Environment (roads, weather/visbility, other vehicles, pedestrians/obstacles)
3. Actuators (steering wheel, accelerator, brake, signal, horn)
4. Sensors (camera, LIDAR, speedometer, GPS, engine sensors)

Characterizing the enviroment

Fully observable (vs. partially observable) Sensors give access to the complete state of the environment at each point in time.

Deterministic (vs. stochastic)

- Next state of the environment is completely determined by current state + action executed.
- If the environment is deterministic except for the actions of other agents, then it is **strategic**

Episodic (vs. sequential)

- Agent's experience is divided into atomic episodes (perceiving + action)
- Choice of action in each episode depends only on that episode

Static (vs. dynamic)

- Environment does not change while agent is thinking
- Semi-dynamic if environment does not change but agent's performance score does

Discrete (vs. continuous) A limited number of distinct, clearly defined percepts and actions

Single agent (vs. multi-agent) Operates by itself in an environment

Explotation vs Exploration

An agent chooses between

- Maximizing expected utility according to current knowledge
- Trying to learn more about the environment (by doing something not optimal)

SEARCHING

Problem types

Deterministic, fully observable

- Single state problem
- Agent knows what state it will be in
- Solution is a sequence

Non-observable

- Sensorless (conformant) problem
- Agent may have no idea where it is
- Solution is a sequence of actions

Non-deterministic and/or partially observable

- Contingency problem
- Percepts provide new info about current state
- Often interleave search, execution

Unknown state space

- Exploration problem

Single state problem formulation

1. Initial state (e.g. at Arad)
2. Actions or successor function (set of action-state pairs)
3. Goal test, which can be
   - explicit, e.g. x = "at Bucharest"
   - implicit, e.g. e.g. Checkmate(x)
4. Path cost (additive)
   - e.g. sum of distances, # of actions
   - Step cost $c(x, a, y)$ assumed to be ≥ 0

Evaluating search strategies

- A search strategy is defined by choosing order of node expansion
- Evaluated using: Completeness | Time complexity | Space complexity | Optimality

Uninformed search

BFS

- Expand shallowest unexpanded node
- Special case of uniform-cost search

Uniform-cost

- Expand least-cost unexpanded node
- Equivalent to BFS if step costs all equal
- Equivalent to A^* with $h(n) = 0$

DFS

- Expand deepest unexpanded node
- Equivalent to best first search, with $f(n) = -\text{depth}$

Depth-limited DFS but terminate at a depth limit

Iterative deepening Run depth-limited from 0 to ∞

b: max branching factor of search tree • d: depth of least-cost solution
m: max depth of state space (may be ∞) • C^*: cost of optimal solution • G: goal state
Red text in this table is filled in by myself

Search	Complete	Time	Space	Optimal
BFS	Yes if b finite	$1 + b + \cdots + b^d = O(b^{d+1})$	$O(b^{d+1})$	Yes if step cost is 1
Uniform-cost	Yes if step cost $\geq \varepsilon$	# of nodes with $g \leq C^* = O\left(b\left\lceil \frac{C^*}{\varepsilon} \right\rceil\right)$	$O\left(b\left\lceil \frac{C^*}{\varepsilon} \right\rceil\right)$	Yes, nodes expanded in increasing order of $g(n)$
DFS	No: fails in ∞-depth spaces, or loops	$O(b^m)$, bad when $m \gg d$, but if solution is deep, can be faster than BFS	$O(bm)$	No
Depth-limited	No	$O(b^l)$	$O(bl)$	No, unless goal happens to be within range
Iterative deepening	Yes	$(d+1)b^0 + db^1 + \cdots + b^d = O(b^d)$	$O(bd)$	Yes if step cost is 1
Greedy best-first	No, can get stuck in loops	$O(b^m)$, but good heuristic may prune paths	$O(b^m)$	No
A^* search	Yes, unless infinitely many nodes with $f \leq f(G)$	$O((\text{effective branching factor})^d)$	$O(b^d)$	Yes
RBFS	Yes	??	$O(bm)$	Yes
Minimax	Yes, if tree finite	$O(b^m)$	$O(bm)$	Yes, against optimal opponent

Tree search

frontier

```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
    if fringe is empty then return failure
    node ← REMOVE-FRONT(fringe)
    if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
    fringe ← INSERTALL(EXPAND(node, problem), fringe)

function EXPAND( node, problem) returns a set of nodes
  successors ← the empty set
  for each action, result in SUCCESSOR-FN[problem](STATE[node]) do
    s ← a new NODE
    PARENT-NODE[s] ← node; ACTION[s] ← action; STATE[s] ← result
    PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
    DEPTH[s] ← DEPTH[node] + 1
    add s to successors
  return successors
```

- May revisit nodes
- Checks for goal state **after** traversing to the node

Graph search

```
function GRAPH-SEARCH( problem, fringe) returns a solution, or failure
  closed ← an empty set
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
    if fringe is empty then return failure
    node ← REMOVE-FRONT(fringe)
    if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
    if STATE[node] is not in closed then
      add STATE[node] to closed
      fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

- Will not revisit nodes
- Checks for goal state **after** traversing to the node

Bidirectional search

- Simultaneously search from initial state and backward state, and terminate when the two searches meet
- $2 \times O(b^{\frac{d}{2}}) < O(b^d)$

Implementation issues

- Succ(n) must be reversible/bidirectional
- Multiple goal states → Construct goal state containing superset of all goal states
- How to check if a node appears in the other search tree
- Can use different search strategies for each half

INFORMED SEARCH

Best-first search

- Use an evaluation function $f(n)$ for each node, estimates desirability
- Expand most desirable unexpanded node
- e.g. Greedy best-first search, A^* search

Greedy best-first search

- Expands node that appears closest to goal
- Can get stuck in loops
- Same as A^* with $g(n) = 0$

A^* search

- Avoid expanding expensive paths
- Evaluation function

$$f(n) = g(n) + h(n)$$

$g(n)$ = cost so far to reach n

$h(n)$ = estimated cost from n to goal

$f(n)$ = estimated total cost of path, through n, to goal

- Can be thought of as a variant for uniform-cost search, where cost is $f = g + h$

Admissible heuristics

- A heuristic h is admissible if it never overestimates the cost to reach the goal
- If $h(n)$ is admissible, then A^* using tree search is optimal

Dominance For admissible heuristic functions h_1 and h_2,

$\forall n\ h_2(n) \geq h_1(n) \Rightarrow h_2$ dominates h_1

and h_2 is typically better for search.

Inventing admissible heuristics

- A problem with fewer restrictions on the actions is called a relaxed problem
- Cost of an optimal solution to relaxed problem is an admissible heuristic for the original problem
- Non-admissible heuristics may be helpful for pruning, but lose optimality

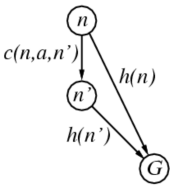
Sample heuristics

- Manhattan distance (can only move horizontally/vertically)
- Number of pieces on the board

Consistent heuristics

- A heuristic h is consistent if for every node n and successor n' generated by action a,

$$h(n) \leq c(n, a, n') + h(n')$$



- Consistent $h \Rightarrow f(n)$ is non-decreasing along any path
- Consistent $h \Rightarrow h$ is admissible
- Consistent $h \Rightarrow A^*$ using graph search is optimal

Memory-bounded heuristic search

IDA* Use $f = g + h$ instead of depth for IDS cutoff

Recursive best-first search

- Remember the 2nd best f-cost for all ancestors of node
- Expand from the best f-cost
- Once the f-cost in the current search exceeds any 2nd best f-cost among ancestors, continue from that instead

Criticism A^* and RBFS use linear space, which underutilizes available memory

Simplified MA*

- Run A^* until memory full
- When memory full
  - Drop node with worst f-value
  - Record on sub-tree
  - If all better paths explored, regenerate this path

Evaluating a heuristic

- Effective branching factor. With N nodes generated, and solution depth d, compute b^*:

$$N + 1 = 1 + b^* + (b^*)^2 + \cdots + (b^*)^d$$

As b^* gets closer to 1, the better the heuristic.

- Effective depth $d - k$, in $O(b^{d-k})$ vs $O(b^d)$ As k increases, the better the heuristic.

Local search

- Used when path to goal state is irrelevant; goal state is solution
- e.g. finding a configuration satisfying some contraints

Hill climbing

- Define objective function for each node
- Define actions/successor functions
- Given a starting node, recurse into highest-valued successor
- If the node has a higher value than successors, then return it. It is a local maximum, but may not be a global maximum

```
function HILL-CLIMBING( problem) returns a state that is a local maximum
  inputs: problem, a problem
  local variables: current, a node
                   neighbor, a node
  current ← MAKE-NODE(INITIAL-STATE[problem])
  loop do
    neighbor ← a highest-valued successor of current
    if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
    current ← neighbor
```

Simulated annealing

- Randomly pick successor/action
- If successor has higher value, recurse. Otherwise recurse with a probability that exponentially decreases.
- Escapes local maxima by allowing bad moves occasionally
- If T decreases slowly enough, then it finds a global optimum with probability approaching 1

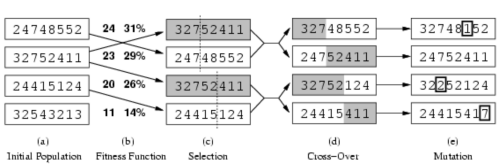
```
function SIMULATED-ANNEALING( problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to "temperature"
  local variables: current, a node
                   next, a node
                   T, a "temperature" controlling prob. of downward steps
  current ← MAKE-NODE(INITIAL-STATE[problem])
  for t ← 1 to ∞ do
    T ← schedule[t]
    if T = 0 then return current
    next ← a randomly selected successor of current
    ΔE ← VALUE[next] – VALUE[current]
    if ΔE > 0 then current ← next
    else current ← next only with probability e^(Δ E/T)
```

Beam search

- Perform k hill-climbing searches in parallel
- Like BFS, but only take best k nodes in each level
- Local beam search: the k threads share info
- Stochastic beam search: k independent threads

Genetic algorithms

- Start with k randomly generated states
- States represented as a string over finite alphabet (often as a string of 0s and 1s)
- Evaluation (fitness) function has higher values for better states
- Generate successor state by combining two parent states via selection, crossover, mutation



ADVERSARIAL SEARCH

Minimax

- Perfect play for deterministic games
- Choose move to position with highest minimax value
- Recursive algorithm (like DFS)
- Minimax algorithm is not optimal against non-optimal MIN player. MAX could have made high risk high reward moves if MAX knows that MIN was non-optimal, but MAX assumed that MIN was optimal and took a safer move, which is not optimal.

```
function MINIMAX-DECISION(state) returns an action
  v ← MAX-VALUE(state)
  return the action in SUCCESSORS(state) with value v

function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for a, s in SUCCESSORS(state) do
    v ← MAX(v, MIN-VALUE(s))
  return v

function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for a, s in SUCCESSORS(state) do
    v ← MIN(v, MAX-VALUE(s))
  return v
```

α-β pruning

- Skip (prune) less-optimal paths that will not be used
- At MIN node, stop if sibling has guaranteed higher value
- At MAX node, stop if sibling has guaranteed lower value

```
function ALPHA-BETA-SEARCH(state) returns an action
  inputs: state, current state in game
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in SUCCESSORS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  inputs: state, current state in game
          α, the value of the best alternative for MAX along the path to state
          β, the value of the best alternative for MIN along the path to state
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for a, s in SUCCESSORS(state) do
    v ← MAX(v, MIN-VALUE(s, α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v
```

Properties

- Pruning should not affect final result
- Best case time complexity is $O(b^{m/2})$

Resource limits

Search space in games is typically large

- Set depth limit to terminate early
- Use evaluation function to estimate desirability of position
- Transpositions: cache previously seen states, as different permuations of moves can lead to the same state
- Pre-compute opening/closing moves

Minimax Cutoff

- *Terminal?* replaced by *Cutoff?*
- *Utility* replaced by *Eval*
- Sets a limit to the search depth
- X-ply: can recurse X levels down
- In chess:
  - 4-ply ≈ human novice
  - 8-ply ≈ typical PC, human master
  - 12-ply ≈ Deep Blue, Kasparov

Evaluation functions

- Estimate the desirability of a certain state using the features
- e.g. Linear weighted sum in chess
  $Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s)$
  - Linear weighted sum assumes features are independent
  - Might need to use non-linear/higher-order terms for dependent features

INTRO (ML)

Definition A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E.

Problem solving

- Know exact formula → Direct Coding
- No formula, but have solution to related problem → Search
- Expert system
- Machine learning

Data collection

- Costly
- Inconsistent data could mean:
  Hidden variables | Noise | Error | All
- Visualize to understand data
- More features/dimensions → exponentially more data needed

ML pipeline

1. Data collection
2. Data extraction (Feature engineering)
3. Data understanding (with Visualization)
4. Data pre-processing
5. Model choice/design
6. Model training
7. Model validation (Evaluation)
8. Model understanding (Visualization/Explainability)
9. Model deployment

Usefulness

- Not possible to program all human knowledge, e.g. facial recognition
- Less to program if machine can learn
- Can discover new knowledge

Types of feedback

Supervised

- Each example has correct answer
- e.g. image of "A" and unicode value 0041
- **Regression**: Predict results within a continuous output, map input variables to some continuous function
- **Classification**: Predict results in a discrete output, map input variables into discrete categories

Unsupervised

- No answers given
- e.g. are there patterns in the data?

Weakly supervised

- Correct answer given, but not precise
- e.g. image contains face (but exact location is not specified)

Reinforcement

- Occasional rewards/penalties given
- e.g. robot navigating a maze

Decision trees

- Expresses a function of the input attributes, that maps to a decision (e.g true/false)
- Internal nodes are questions/tests, while leaf nodes represent the decision given the state
- Many possible orderings, want a compact tree ⇒ choose attribute with largest information gain

Lecture Training

- Can be used for multi-label classification, by having each label as a leaf in the tree
- Can be used for regression, as it splits the data into smaller subsets. These subsets then can be assigned the most common results in the case of a regression.
- Two decision trees may be logically equivalent, but have different internal structure (i.e. they are different)
- Can use binary or discrete or binned/discretized continuous valued attributes, but not continuous valued attributes

```
function DTL(examples, attributes, default) returns a decision tree
  if examples is empty then return default
  else if all examples have the same classification then return the classification
  else if attributes is empty then return MODE(examples)
  else
    best ← CHOOSE-ATTRIBUTE(attributes, examples)
    tree ← a new decision tree with root test best
    for each value v_i of best do
      examples_i ← {elements of examples with best = v_i}
      subtree ← DTL(examples_i, attributes − best, MODE(examples))
      add a branch to tree with label v_i and subtree subtree
    return tree
```

Entropy

- Measure "impurity" or "randomness" in data
- Defined as
  $$I(P(v_1), \cdots, P(v_n)) = -\sum_{i=1}^{n} P(v_i) \log_2 P(v_i)$$
- For training set with p positive examples and n negative examples,
  $$I\left(\frac{p}{p+n}, \frac{n}{p+n}\right) = -\left(\frac{p}{p+n}\log_2\frac{p}{p+n} + \frac{n}{p+n}\log_2\frac{n}{p+n}\right)$$

Information gain

- Entropy of this node minus entropy of children nodes
- p and n refer to the examples at the node, p_i and n_i refer to the examples remaining at a specific child node
  $$\text{remainder}(A) = \sum_{i=1}^{v} \frac{p_i + n_i}{p+n} I\left(\frac{p_i}{p_i+n_i}, \frac{n_i}{p_i+n_i}\right)$$
  $$IG(A) = I\left(\frac{p}{p+n}, \frac{n}{p+n}\right) - \text{remainder}(A)$$

LINEAR REGRESSION

m is the number of training samples

Error function

- Using an error function for loss implies some meaning to the distance between predicted and actual values
- In classification, this distance is between class labels, so it may not make sense to use error functions for loss

Mean squared error (MSE) Used for convenience, because it is differentiable everywhere

$$J(w_0, w_1) = \frac{1}{2m}\sum_{i=1}^{m}(h_w(x^{(i)}) - y^{(i)})^2$$

Mean absolute error (MAE)

- Not differentiable everywhere
- Emphasizes the error less, may be better if there are relatively many outliers in dataset

Gradient descent

- Gives a local minimum
- In implementation, learning rate α often decreases with more iterations
  - Start big to reach optimal values faster
  - Slow down to allow for gradual convergence
  - Idea behind a learning rate scheduler

Single-variable gradient descent

- Hypothesis h takes in a vector, returning a scalar
$$h_w(x) = w_0 + w_1 x$$
- Cost function
$$J(w_0, w_1) = \frac{1}{2m}\sum_{i=1}^{m}(h_w(x^{(i)}) - y^{(i)})^2$$
- Goal: Pick w_0, w_1 to minimize $J(w_0, w_1)$

Updating weights

- Update weights simultaneously using positive learning rate α

$$temp0 := w_0 - \alpha\frac{\partial J(w_0, w_1)}{\partial w_0}$$
$$temp1 := w_1 - \alpha\frac{\partial J(w_0, w_1)}{\partial w_1}$$
$$w_0 := temp0$$
$$w_1 := temp1$$

- Partial derivatives:
$$\frac{\partial J(w_0, w_1)}{w_0} = \frac{1}{m}\sum_{i=1}^{m}(w_0 + w_1 x^{(i)} - y^{(i)})^2$$
$$\frac{\partial J(w_0, w_1)}{w_1} = \frac{1}{m}\sum_{i=1}^{m}(w_0 + w_1 x^{(i)} - y^{(i)})^2 \cdot x^{(i)}$$

Variants

Batch gradient descent Consider all training examples at each iteration

Stochastic gradient descent Consider one data point at time

- Faster
- More randomness - might escape local minima

Multivariable gradient descent

- Hypothesis h takes in a vector, returning a scalar
$$h_w(x) = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n$$
or
$$h_w(x) = w^T x$$
where $w = \begin{pmatrix} w_0 \\ w_1 \\ \vdots \\ w_n \end{pmatrix}$ and $x = \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} 1 \\ x_1 \\ \vdots \\ x_n \end{pmatrix}$. The 1 introduced for the bias may be treated as another variable x_0.
- Cost function
$$J(w) = \frac{1}{2m}\sum_{i=1}^{m}(h_w(x^{(i)}) - y^{(i)})^2$$
- Goal: Pick w to minimize $J(w)$

Updating weights

$$w_j := w_j - \alpha\frac{\partial J(w)}{\partial w_j} \quad \text{simultaneously for all } w_j$$
$$= w_j - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_w(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$$

Feature scaling

- Gradient descent may not work well if features have signficantly different scales
- Perform mean normalization
$$x_i := \frac{x_i - \mu_i}{\sigma_i}$$
- Useful in polynomial regression, where features likely have significantly different scales

Polynomial regression

- Generate new features which are higher-order terms of the original ones
- With a dataset of N points, the max degree polynomial one should use is $N - 1$. This is the least degree required to fit to all N points; any higher degree polynomial would be overfitting.

Normal equation

Define design matrix X and output vector Y as follows:

$$X = \begin{pmatrix} 1 & (x^{(1)})^T \\ 1 & (x^{(2)})^T \\ \vdots & \vdots \\ 1 & (x^{(n)})^T \end{pmatrix}, \quad Y = \begin{pmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{pmatrix}$$

where each row corresponds to a sample. Then

$$w = (X^T X)^{-1} X^T Y$$

Properties

- No need to choose α
- No iterations
- $X^T X$ needs to be invertible
- Slow if n is large - $O(n^3)$

What if $X^T X$ is not invertible

- Infinite solutions to the normal equation
- Possible reasons:
  - If $m < n$, i.e. # training data < number of features, $X^T X$ is always not invertible. More data is required to get a unique solution.
  - Not all columns are linearly independent. Consider removing one feature that might be dependent on a linear combination of others

Derivation

$$J(w) = \frac{1}{2m}(Xw - Y)^T(Xw - Y)$$
$$2mJ(w) = ((Xw)^T - Y^T)(Xw - Y)$$
$$= (Xw)^T(Xw) - (Xw)^T Y - Y^T(Xw) - Y^T Y$$
$$= w^T(X^T X)w - 2(Xw)^T Y + Y^T Y$$

Differentiate and set to 0:

$$\frac{\partial J}{\partial w} = 2(X^T X)w - 2X^T Y = 0$$
$$(X^T X)w = X^T Y$$
$$w = (X^T X)^{-1} X^T Y$$