Exam notes for CAB320

Graph theory

Vertex-vertex incidence matrix (also called adjacency matrix)

- **In-degree** is number of incoming arcs of a vertex
- **Out-degree** is number of outgoing arcs to a vertex

Clique is a subset of vertices of an undirected graph such that every two distinct vertices in a clique are adjacent. A **maximal clique** is a clique that cannot be extended by including one more adjacent vertex.

Dijkstra's algorithm

```
Dijkstra(Graph, source):
    1. Initialize:
       a. For each node v in the graph:
          - dist[v] := infinity (a very large value)
          - previous[v] := undefined (to store the path)
       b. dist[source] := 0 (distance from source to itself is 0)
    2. Create a priority queue Q with each node v, where the priority is dist[v]
       Q := the set of all nodes in the graph
    3. While Q is not empty:
       a. u := node in Q with the smallest dist value
       b. Remove u from Q
       c. For each neighbor v of u:
          - alt := dist[u] + weight(u, v)
          - If alt < dist[v]:
             i. dist[v] := alt
             ii. previous[v] := u
             iii. Update the priority of v in Q (decrease dist[v])
    4. Return dist and previous
```

Search algorithms

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes	Yes, if $\epsilon > 0$	No	Yes, if $l \geq d$	Yes
Time	b^d	$b^{\lceil C^*/\epsilon \rceil}$	b^m	b^l	b^d
Space	b^d	$b^{\lceil C^*/\epsilon \rceil}$	bm	bl	bd
Optimal?	Yes*	Yes	No	No	Yes*

*if all step costs are identical

Notes:

1. Uniform cost: expand the cheapest unexpanded nodes. frontier = priority queue ordered by path cost g(n). If step costs are same then UCS = BFS
2. DFS is complete if avoid repeated states & state-space is finite.
3. Iterative deepening recalculates the shallow nodes (but no big effect)
4. DLS has a returns of cutoff occurred / solution / failure

A *algorithm = UCS $g(n)$ (path cost, history) + greedy algorithm $h(n)$ (goal proximity)*
Terminates when dequeue a goal (for optimality)
***Admissibility** $0 \leq h(n) \leq h^(n)$* (optimal in tree search)
Consistency $h(A) - h(C) \leq \text{cost(A to C)}$ (implies admissibility, optimal in graph search)
Optimality $f(n_1) \leq f(n_2) \leq \cdots \leq f(n_k)$

Intelligent agent

Environment -> (percepts) -> sensors -> decision-making -> actuators -> actions -> env
Environment types:

- **Deterministicness** (deterministic or stochastic): An environment is deterministic if the next state is perfectly predictable given knowledge of the previous state and the agent's action.
- **Staticness** (static or dynamic): Static environments do not change while the agent deliberates.
- **Observability** (full or partial): A fully observable environments is one in which the agent has access to all information in the environment relevant to its task.
- **Agency** (single or multiple): If there is at least one other agent in the environment, it is a multi-agent environment. Other agents might be apathetic, cooperative, or competitive.
- **Knowledge** (known or unknown): An environment is considered to be "known" if the agent understands the laws that govern the environment's behavior. For example, in chess, the agent would know that when a piece is "taken" it is removed from the game. On a street, the agent might know that when it rains, the streets get slippery.
- **Episodicness** (episodic or sequential): Sequential environments require memory of past actions to determine the next best action. Episodic environments are a series of one-shot actions. An AI that looks at radiology images to determine if there is a sickness is an example of an episodic environment. One image has nothing to do with the next.
- **Discreteness** (discrete or continuous): A discrete environment has fixed locations or time intervals. A continuous environment could be measured quantitatively to any level of
  Agents types
- **Simple reflex agents** (line follow robot)
- **Reflex agents with state**: Choose action based on current percept (and maybe memory). Do not consider the future consequences of their actions
- **Goal-based agents** (Sokoban agent): Goal-driven agents
- **Utility-based agents** (Pac-man player): Adding utility component and "how happy I am in such state"
  PEAS: Performances, Environment, Actuator, Sensor

Problem class:

1. Initial state: the initial state of the problem `In(state)`

2. Actions: a set of actions for state s `Actions(In(s)) = {ActionA, ActionB, ActionC}`
3. Transitions: the result of the action, or successor state `Result(In(s), ActionA) = In(sA)`
4. Goal test: can be explicit set of states, or implicit property (such as `checkmate`) `{In(sucess_state)}`
5. Path cost: is the summation of step costs (sum of distances, number of actions executed, etc) `c(s, a, s')`
   State $\neq$ nodes: state is a physical configuration, node is a data structure constituting parts of search tree (state, parent, child, depth, path cost, while state does not!)

Machine learning

Collect data -> Prepare data -> Train data -> Validate model -> Test model (-> online inference)
Holdout validation is a simple technique that splits the dataset into a training set and a test set, usually according to a certain ratio (e.g., 70/30 or 80/20).
Cross-validation is a more complex and robust model evaluation technique that divides the dataset into multiple (k) subsets for multiple rounds of training and testing
Bias is inability for a machine learning method to capture the true relationship between the input and the output (x and y)
Variance is difference in fit (error between estimates and reality) between the datasets. Variability of results for a new dataset

	supervised	**Unsupervised**
discrete	classification / categorisation	clustering
Continuous	regression	dimensionality reduction

Overfitting - perform very well in training, not in test. Coeffs explosion
Sum of squares error function $E(w) = \frac{1}{2}\sum_{n=1}^{N}(y(x_n, w) - t_n)^2$
Regularisation $E(w) = \frac{1}{2}\sum_{n=1}^{N}(y(x_n, w) - t_n)^2 + \frac{\lambda}{2}||w||^2$
Bayes theorem $p(Y|X) = \frac{p(X|Y)p(Y)}{p(X)}$ and $p(X) = \sum_Y p(X|Y)p(Y)$ (where $p(X|Y)p(Y)$ is class model, $p(Y)$ is prior, $p(X) = \sum_Y p(X|Y)p(Y)$ is normaliser)
Assumption: attributes are conditionally independent
$v_{map} = argmax\ P(v_j|a_1, \dots a_n) = argmax\ P(v_j)\prod P(a_i|v_j)$
kNN: Low values of k may lead to noisy results + sensitive to outliers. Large values of k may lead to smoother results, but inappropriate for classes with limited samples, and computationally expensive. K is often be odd for deterministic result.
Evaluation
Accuracy = $\frac{TN+TP}{TN+TP+FN+FP}$ easy to see/ doesn't show types of error, sensitive to class imbalance
Precision $P = \frac{TP}{TP+FP}$, Recall $R = \frac{TP}{TP+FN}$ gives types, better for imbalanced class / declares all to minimised false alarms (precision) or missed detection (recall), one doesn't tell story
F1 $= \frac{2PR}{P+R} = \frac{2\times TP}{2\times TP+FP+FN} = 2 \times \frac{\text{sensitivity}\times\text{precision}}{\text{sensitivity}+\text{precision}}$ sensitive to class imbalanced
Sensitivity (True positive rate) = $\frac{TP}{TP+FN}$, Specificity (True negative rate) = $\frac{TN}{TN+FP}$
Balanced accuracy = $\frac{TPR+TNR}{2} = \frac{\text{sensitivity + specificity}}{2}$ doesn't affected by imbalanced class
MCC $= \frac{TP\times TN - FP\times FN}{\sqrt{(TP+FP)(TP+FN)(TF+FP)(TF+FN)}}$
Receiver Operating characteristic (ROC) Curve: True Positive Rate (TPR) vs False Positive Rate (FPR) (i.e., Sensitivity vs. (1-specificity)). Area under the curve (AUC) is a way to measure the performance of a classifier.

Other evaluation: reliability / robustness / computation cost / latency / efficiency / correctness / predictability / transparency

Ethics

1. **Explainability**: AI system that can explain its decisions. Explanations must be understandable by the user.
2. **Auditability**: when something goes wrong, we need to be able to work out what happened. Equivalent to the black box on planes?
3. **Robustness**: An AI system is robust if it is capable of dealing with perturbations to their inputs.
4. **Correctness**: assurances the syst will act 'correctly'. E.g. safe bounds that will never be passed.
5. 5. Fairness: are the results computed by the AI system "fair"?
6. **Respect for Privacy**: where does the data come from? Is that ok to train the ML algo. with it?
7. **Transparency**: Transparency can help engender trust. However, transparency itself does not necessarily engender trust. There are also pitfalls to being transparent.

DP

Greedy: Build up a solution incrementally, myopically optimising some local criterion.
Example: Colouring graph
Divide and conquer: Break up the problem into independent, subproblems, solve each subproblem, and combine the solution to subproblems to form a solution to original problem.
Example: Quicksort
Dynamic programming: Break up the problem into a series of overlapping subproblems, and build up solutions to larger and larger subproblems. Fancy name for caching away intermediate results in a table for later use. Example: Dijkstra's algorithm

Weighted interval (binary choice)
Case 1: OPT selects job j

- Collect profits v_j
- Do not include the incompatible jobs $\{p(j)+1, p(j)+2, \ldots, j-1\}$
- Must include the optimal solution to problem consisting of remaining compatible jobs $\{1, 2, \ldots, p(j)$ $\}$
  Case 2: OPT does not select job j
- Must include the optimal solution to problem consisting of remaining compatible jobs $\{$ $1, 2, \ldots, j-1\}$

```
# Memoisation
Input: n, s[1...n], f[1...n], v[1...n]
Sort jobs by finish time so that f[1] <= f[2] <= ... <= f[n]
Compute p[1], p[2], ... p[n]

# Initialisation
for j = 1 to n
        M[j] <- empty
M[0] = 0
```

```
# M-Compute-Opt(j)
if M[j] is empty
        M[j] <- max(v[j]+ Compute-Opt(p[j]), Compute-Opt(p[j-1]))
return M[j]

# Find-solution(j)
if j = 0
        return empty_set
else if (v[j] + M(p[j]) > M[j-1])
        return {j} union Find-solution(p[j])
else
        return Find-solution(j-1)

# Bottom-up(n, s, f, v)
M[0] <- 0
for j = 1 to n
        M[j] <- max{v_j + M[p(j)], M[j-1]}
```

Segmented least squares (multiway)

- $\mathrm{OPT}(j)$ be the minimum cost for points $p_1, p_2 \ldots, p_j$
- $e(i,\ j)$ be the minimum SSE for points $p_1, p_2 \ldots, p_j$
  $\mathrm{OPT}(j)$ is computed by
- last segment uses points $p_i, p_{i+1} \ldots, p_j$ for some j
- Cost = $e(i,\ j) + c + \mathrm{OPT(i\text{-}1)}$ (optimal substructure property, proved via exchange argument)

Knapsack

Case 1: OPT does not select item i
OPT selects best of $\{1, 2, \ldots, i-1\}$ using weight w
Case 2: OPT selects item i
New weight limit $w - w_i$. OPT selects best of $\{1, 2, \ldots, i-1\}$ using weight $w - w_i$

$$OPT(i,w) = \begin{cases} 0 & \text{if } \mathrm{i} = 0 \\ OPT(i-1,w) & \text{if } \mathrm{w_i} > \mathrm{w} \\ \max\{\ OPT(i-1,w),\ \ v_i + OPT(i-1, w-w_i)\} & \text{otherwise} \end{cases}$$

Levenshtein distance (adding a new variable)

Case 1: OPT matchs $x_i - y_j$
Pay mismatch for $x_i - y_j$ + min cost of aligning $\{x_1, x_2, \ldots, x_{i-1}\}$ and $\{y_1, y_2, \ldots, y_{j-1}\}$
Case 2-1: OPT leaves x_i unmatched
Pay gap for x_i + min cost aligning $\{x_1, x_2, \ldots, x_{i-1}\}$ and $\{y_1, y_2, \ldots, y_j\}$
Case 2-2: OPT leaves y_j unmatched

Pay gap for y_j + min cost aligning $\{x_1, x_2, \ldots, x_i\}$ and $\{y_1, y_2, \ldots, y_{j-1}\}$

$$OPT(i,j) = \begin{cases} j\delta & \text{if } i = 0 \\ \min \begin{cases} \alpha_{x_i y_j} + OPT(i-1, j-1) \\ \delta + OPT(i-1, j) \\ \delta + OPT(i, j-1) \end{cases} & \text{otherwise} \\ i\delta & \text{if } j = 0 \end{cases}$$

Reinforcement

Action-value method: $Q_t(a) = \frac{\text{sum of rewards when a taken prior to t}}{\text{number of times a taken prior to t}} = \frac{\sum_{i=1}^{t-1} R_i * 1_{A_i=a}}{\sum_{i=1}^{t-1} 1_{A_i=a}}$
Running average: $Q_{n+1} = Q_n + \frac{1}{n}[R_n - Q_n]$ or $(1-\alpha)^n Q_1 + \sum_{i=1}^{n} \alpha(1-\alpha)^{n-i} R_i$ if non-stationary
State value: $v_\pi(s) = E[G_t | S_t = s, A_t = a, A_{t+1:\infty} \sim \pi]$
Optimal state value $v_\pi(s) = \max_\pi v_\pi(s)$
Action-value function: $q_\pi(s,a) = E[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots \mid S_t = s, A_t = a, A_{t+1:\infty} \sim \pi]$
Optimality of action value pairs: $q_{\pi_*}(s,a) = \max_\pi q_\pi(s,a) = q_*(s,a)$
Optimal policy: $\pi_*(s) = \arg\max_a q_*(s,a)$ and strictly $\pi_*(a|s) > 0$ and $q_*(s,\ a) = max_b q_*(s,\ b)$
Greedification (policy dancing): $\pi'(s) = \arg\max_a q_p i(s,\ a)$ and $q_{\pi'}(s,\ a) \geq q_\pi(s,\ a)$
Continuing tasks (discounted return, $G_t = R_{t+1} + \gamma R_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$)
Episodic task (total reward when at terminal state, $G_t = R_{t+1} + R_{t+2} + \cdots + R_T$)
Bellman: $G_t = R_{t+1} + \gamma R_{t+2} + \cdots = R_{t+1} + \gamma G_{t+1}$ (γ is discounted rate)
so $v_\pi(s) = E[G_t | S_t = s] = E[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s]$
or $v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)[r + \gamma v_\pi(s')]$
Bellman optimality:

- for v, $v_*(s) = max_a q_{\pi_*}(s,a) = max_a \sum_{s',r} p(s',r|s,a)[r + \gamma v_\pi(s')]$
- for q, $q_*(s) = E[R_{t+1} + \gamma max_{a'} q_*(S_{t+1}, a') | S_t = s, A_t = a] = \sum_{s',r} p(s',r|s,a)[r + \gamma \max_{a'} q_* \pi(s',a')]$
  Policy evaluation backup: $v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s\,r} p(s',r|s,a)[r + \gamma v_k(s')]$
  TD

Tabular TD(0) for estimating v_π

Input: the policy π to be evaluated
Initialize $V(s)$ arbitrarily (e.g., $V(s) = 0$, for all $s \in \mathcal{S}^+$)
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    $A \leftarrow$ action given by π for S
    Take action A, observe R, S'
    $V(S) \leftarrow V(S) + \alpha\big[R + \gamma V(S') - V(S)\big]$
    $S \leftarrow S'$
  until S is terminal

SARSA

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
    Initialize S
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Repeat (for each step of episode):
        Take action A, observe R, S'
        Choose A' from S' using policy derived from Q (e.g., ε-greedy)
        $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma Q(S',A') - Q(S,A)]$
        $S \leftarrow S'$; $A \leftarrow A'$;
    until S is terminal

Q-Learning

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
    Initialize S
    Repeat (for each step of episode):
        Choose A from S using policy derived from Q (e.g., ε-greedy)
        Take action A, observe R, S'
        $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$
        $S \leftarrow S'$;
    until S is terminal

Neural network

Loss function $L = -\mathrm{y}_{\text{true}} + \log \sum_j e^{y_j}$ or $-\log(\frac{e^{\mathrm{y}_{\text{true}}}}{\sum_j e^{y_j}})$
$y = W^T x + b$, in backprop, $W' = W - s \times \frac{\partial L}{\partial W}$ and $b' = b - s \times \frac{\partial L}{\partial b}$ (s = learning step size)
softmax $\sigma_i(z) = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}$

Appendices

Tree-search

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

```
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(STATE[node], action, result)
       DEPTH[s] ← DEPTH[node] + 1
       add s to successors
   return successors
```

Depth-limited search

```
function DEPTH-LIMITED-SEARCH( problem, limit) returns soln/failure/cutoff
   RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)
function RECURSIVE-DLS(node, problem, limit) returns soln/failure/cutoff
   if GOAL-TEST(problem, STATE[node]) then return node
   else if limit = 0 then return cutoff
   else
       cutoff-occurred? ← false
       for each action in ACTIONS(STATE[node], problem) do
           child ← CHILD-NODE(problem, node, action)
           result ← RECURSIVE-DLS(child, problem, limit − 1)
           if result = cutoff then cutoff-occurred? ← true
           else if result ≠ failure then return result
       if cutoff-occurred? then return cutoff else return failure
```

Iterative-deepening search

```
function ITERATIVE-DEEPENING-SEARCH( problem) returns solution/failure

   for depth ← 0 to ∞ do
       result ← DEPTH-LIMITED-SEARCH( problem, depth)
       if result ≠ cutoff then return result
   end
```

Graph search

```python
def graph_search(problem, frontier):
    """
    Search through the successors of a problem to find a goal.
    The argument frontier should be an empty queue.
    If two paths reach a state, only use the first one. [Fig. 3.7]
    Return
        the node of the first goal state found
        or None is no goal state is found
    """
    assert isinstance(problem, Problem)
    frontier.append(Node(problem.initial))
    explored = set() # initial empty set of explored states
    while frontier:
        node = frontier.pop()
        if problem.goal_test(node.state):
            return node
        explored.add(node.state)
        # Python note: next line uses of a generator
        frontier.extend(child for child in node.expand(problem)
                        if child.state not in explored
                        and child not in frontier)
    return None

```

Best graph search

```python
def best_first_graph_search(problem, f):
    """
    Search the nodes with the lowest f scores first.
    You specify the function f(node) that you want to minimize; for example,
    if f is a heuristic estimate to the goal, then we have greedy best
    first search; if f is node.depth then we have breadth-first search.
    """
    node = Node(problem.initial)
    if problem.goal_test(node.state):
        return node
    frontier = PriorityQueue(f=f)
    frontier.append(node)
    explored = set() # set of states
    while frontier:
        node = frontier.pop()
        if problem.goal_test(node.state):
            return node
        explored.add(node.state)
        for child in node.expand(problem):
            if child.state not in explored and child not in frontier:
                frontier.append(child)
            elif child in frontier:
                # frontier[child] is the f value of the
                # incumbent node that shares the same state as
                # the node child.  Read implementation of PriorityQueue
                if f(child) < frontier[child]:
                    del frontier[child] # delete the incumbent node
                    frontier.append(child) #
    return None
```

A* graph search
is 'best first graph search'
with f=g+h