COMP3702/7702 2020 - Mock Exam

MODULE 1: SEARCH

Question 1.1

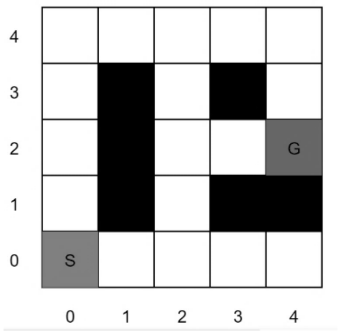
Regarding blind search algorithms:

a) What **data structure** is best used to implement breadth-first search?
A (first-in-first-out) queue
b) What **blind search** algorithm or algorithms can be implemented using priority queue?
Uniform cost search (not A* search)
c) Explain in your own words the **benefits of using iterative-deepening depth-first search over depth-first search** in a search problem.
IDDFS combines the optimality and completeness of BFS with the space (memory) efficiency of DFS. Whenever the goal state is shallower than the depth of the search tree, IDDFS takes a little more time than BFS, but saves a lot of memory compared to BFS (like DFS).

UCB looks at the lowest cost path first

Question 1.2

The details of this problem are:



- The 5x5 grid has obstacles indicated by black tiles in three contiguous blocks, at coordinates (1,1), (1,2), (1,3); at (3,1) and (4,1); and at (3,3).
- The initial state is S, at coordinates (0,0), and the goal state is G, at (4,2).

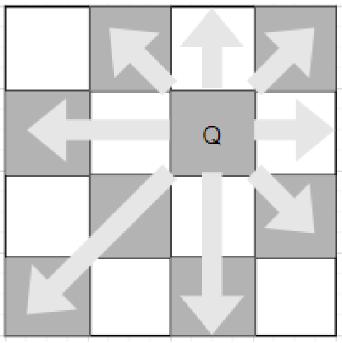
For the questions that ask for a path, please give your answers in the form 'S–(x,y)–…–(x,y)–G'. Break any priority ties using the move order *up, right, down, left*.

a) What path does iterative-deepening depth-first search with depth parameter k = 3 return for this search problem?
S-(1,0)-(2,0)-(2,1)-(2,2)-(3,2)-(G)
b) Now assume a constant cost of 1 per move. What path does uniform cost search return for this search problem?
S-(1,0)-(2,0)-(2,1)-(2,2)-(3,2)-(G)
c) Consider a heuristic for this search problem given by *h* = *4-x* (*x* is the x-coordinate).
   i. Is the heuristic *h* consistent? Yes
   ii. Is the heuristic *h* admissible? Yes

A consistent heuristic is also admissible, i.e. it never overestimates the cost of reaching the goal (the converse, however, is not always true).
Consistent: calc the diff of estimated cost of each neighbour then compare with actual cost of each step, if diff > actual – not consistent
Admissible: calc the actual shortest dist from goal, compare with estimated, if estimated > actual – not admissible
A heuristic is deemed **admissible** if it never overestimates the cost to the nearest goal.
A heuristic is **consistent** if it never overestimates the actual step cost between neighboring nodes.

MODULE 2: REASONING WITH CERTAINTY

Question 2.1



In chess, a queen can move in straight lines any number of squares left or right, forward or backward, or diagonally. No two queens threaten each other.

A simpler version of the problem is the 4 queens puzzle, played on a 4x4 board, shown below: The possible moves of a single queen are shown in this figure.

The typical way to model this problem is to assign each of the 4 queens its own column, A, B, C or D, and then choose a row 1, 2, 3 or 4 in such a way that they cannot attack each other. Starting from this variable definition, answer the following questions on the 4 queens puzzle.

a) What is the domain of each queen?
Row numbers, i.e. $Q_i \in \{1,2,3,4\}$
b) List all binary constraints between variables for this CSP. How many constraints are there?
Row-different(Q_i,Q_j) for rows *i,j* in *{A,B,C,D}*, with *i* not equal to *j*

- I.e. $Q_i \neq Q_j$
- How many? From the left, there are three Row-diff for Q-A, two more for Q-B, and one more for Q-C; Q-D's row-diff constraint are included in the rest.
- Six row-diff constraints.

Diag-different(Q_i,Q_j) for *i,j* in *{A,B,C,D}*, *i* not equal to *j*

- They cannot be the same number of columns apart as they are rows apart, i.e. $|Q_i - Q_j| \neq |i-j|$
- How many? Same as Row-diff; for Q-A from the left, there are 3 Diag-diff for Q-A, two more for Q-B, and one more for Q-C; Q-D's row-diff constraint are included in the rest.
- Six row-diff constraints.

12 binary constraints in total.

c) Express the problem as one of logical satisfiability in *conjunctive normal form*.

$(Q_A \neq Q_B) \wedge (Q_A \neq Q_C) \wedge (Q_A \neq Q_D) \wedge (Q_B \neq Q_C) \wedge (Q_B \neq Q_D) \wedge (Q_C \neq Q_D) \wedge$
$(|Q_A - Q_B| \neq 1) \wedge (|Q_A - Q_C| \neq 2) \wedge (|Q_A - Q_D| \neq 3) \wedge (|Q_B - Q_C| \neq 1) \wedge (|Q_B - Q_D| \neq 2) \wedge (|Q_C - Q_D| \neq 1)$

d) Assume a partial assignment is given, where Q-A is placed in row 3. Apply *backtracking search* starting from this partially assigned CSP. Use the variable ordering (Q-B, Q-C, Q-D) and the variable domain order 1,2,3,4 to expand nodes in the search graph. List all variable assignment and removal operations, and any backtracking operations.

1. Check Q-B = 1 – **Assign Q-B to row 1**
2. Check Q-C = 1 – remove b/c conflict with Q-B row and Q-A diag
3. Check Q-C = 2 – remove b/c conflict with Q-B diag
4. Check Q-C = 3 – remove b/c conflict with Q-A row
5. Check Q-C = 4 – **Assign Q-C to row 4**
6. Check Q-D = 1 – remove b/c conflict with Q-B row
7. Check Q-D = 2 – **Assign Q-D to row 2**

e) Give a solution to this CSP.

Q-A=3   Q-B=1   Q-C=4   Q-D=2

Question 2.2

a) Construct a truth table to show that ¬(p ∨ q) is logically equivalent to (¬p ∧ ¬q).

Literals		Sentence	
p	q	¬(p ∨ q)	(¬p ∧ ¬q)
1	1	0	0
1	0	0	0
0	1	0	0
0	0	1	1

b) Given the premises (p ⇒ q) and (r ⇒ s), use the Propositional Resolution rule to prove the conclusion (p ∨ r ⇒ q ∨ s).

Drive out arrows from the premises:

(p ⇒ q) ⇔ (¬p ∨ q)   (1)

(r ⇒ s) ⇔ (¬r ∨ s)   (2)

Next, we want the goal statements in CNF. First, drive out arrows from the goal statement:

(p ∨ r ⇒ q ∨ s) ⇔ ¬ (p ∨ r) ∨ (q ∨ s)

Then apply de Morgan's laws (twice) to drive in ANDs:

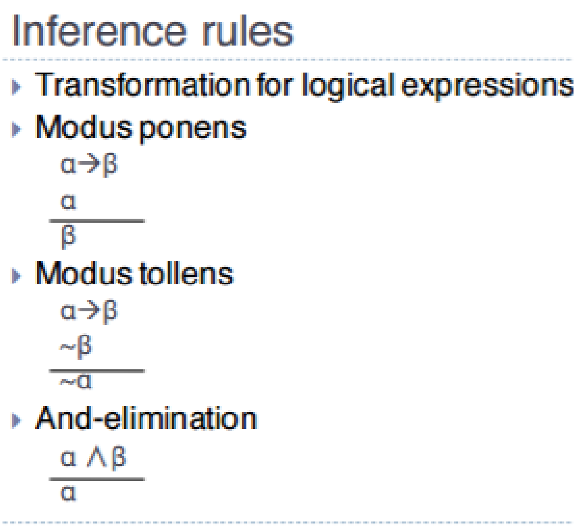
¬ (p ∨ r) ∨ (q ∨ s) ⇔ ¬ ((p ∨ r) ∧ ¬q ∧ ¬s))

Negate the goal (we are using proof by contradiction) and list the goal conjuncts separately:

(p ∨ r)   (3)
¬q   (4)
¬s   (5)

Now apply resolution to the premises and goal statements:

Resolution of (1) and (4):



Inference rules

- Transformation for logical expressions
- Modus ponens

$$\frac{\alpha \rightarrow \beta \quad \alpha}{\beta}$$

- Modus tollens

$$\frac{\alpha \rightarrow \beta \quad {\sim}\beta}{{\sim}\alpha}$$

- And-elimination

$$\frac{\alpha \wedge \beta}{\alpha}$$

$$\begin{array}{l} (\neg p \lor q) \\ \underline{\neg q} \\ \neg p \end{array} \qquad (6)$$

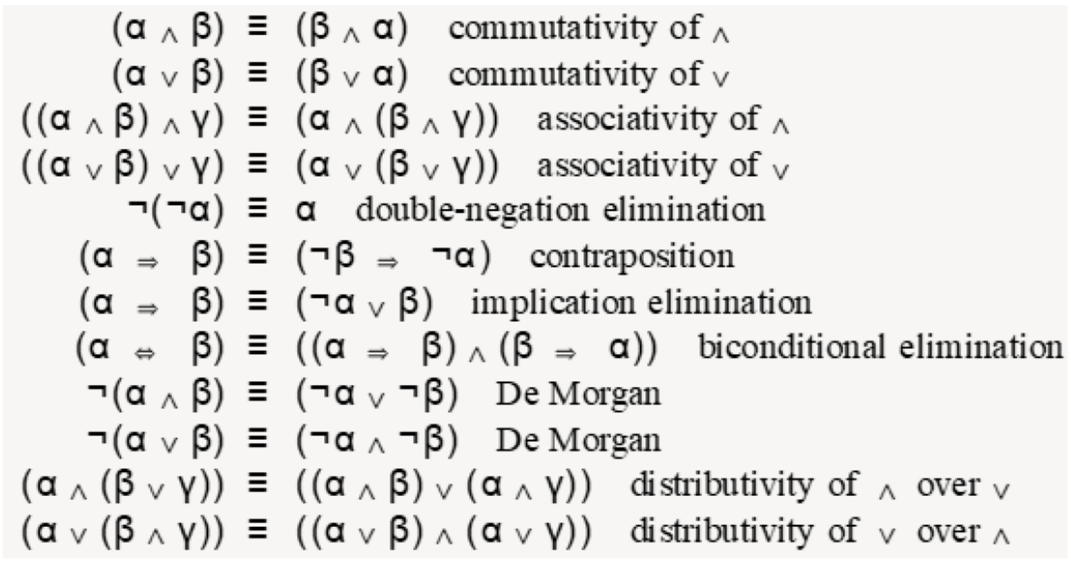
Resolution of (2) and (3):

$$\begin{array}{l} (\neg r \lor s) \\ \underline{\neg s} \\ \neg r \end{array} \qquad (7)$$

Resolution of (3) and (6):

$$\begin{array}{l} (p \lor r) \\ \underline{\neg p} \\ r \end{array} \qquad (8)$$

From (7) and (8) we have $r \land \neg r$, a contradiction. Thus, we prove the original conclusion.



$(\alpha \land \beta)$	$\equiv$	$(\beta \land \alpha)$	commutativity of $\land$
$(\alpha \lor \beta)$	$\equiv$	$(\beta \lor \alpha)$	commutativity of $\lor$
$((\alpha \land \beta) \land \gamma)$	$\equiv$	$(\alpha \land (\beta \land \gamma))$	associativity of $\land$
$((\alpha \lor \beta) \lor \gamma)$	$\equiv$	$(\alpha \lor (\beta \lor \gamma))$	associativity of $\lor$
$\neg(\neg\alpha)$	$\equiv$	α	double-negation elimination
$(\alpha \Rightarrow \beta)$	$\equiv$	$(\neg\beta \Rightarrow \neg\alpha)$	contraposition
$(\alpha \Rightarrow \beta)$	$\equiv$	$(\neg\alpha \lor \beta)$	implication elimination
$(\alpha \Leftrightarrow \beta)$	$\equiv$	$((\alpha \Rightarrow \beta) \land (\beta \Rightarrow \alpha))$	biconditional elimination
$\neg(\alpha \land \beta)$	$\equiv$	$(\neg\alpha \lor \neg\beta)$	De Morgan
$\neg(\alpha \lor \beta)$	$\equiv$	$(\neg\alpha \land \neg\beta)$	De Morgan
$(\alpha \land (\beta \lor \gamma))$	$\equiv$	$((\alpha \land \beta) \lor (\alpha \land \gamma))$	distributivity of $\land$ over $\lor$
$(\alpha \lor (\beta \land \gamma))$	$\equiv$	$((\alpha \lor \beta) \land (\alpha \lor \gamma))$	distributivity of $\lor$ over $\land$

MODULE 3: DECISION MAKING UNDER UNCERTAINTY

Question 3.1

The *decomposability* axiom of the rational preferences utility model asserts that “there is no fun in gambling.” Explain how an agent whose preferences violate the decomposability axiom can be manipulated using the concept of a *money pump*.

Using a violation of the decomposability axiom, you can set up an intransitive preference loop and apply the money pump.

Your typical money pump is constructed for an intransitive loop of preferences: $A < B < C < A$. At each step of the money pump, a small payment is taken from the agent to swap to the next-preferred option.

Let's now assume we don't have C, but instead we have a lottery, $D = pA + (1-p) B$.

If an agent values the lottery element enough, that is, if it gets enough pleasure from gambling, we can imagine a situation where: $A < B < D$, and D realises one of A or B.

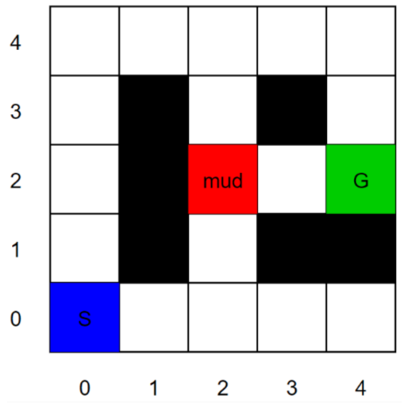
Let's cycle through these preferences to pump money out of the agent:

1. Assume the agent starts with A, and that you start with a stash of A and B. Repeat the following:
2. Trade B for A plus a small payment.
3. Offer the lottery D for B plus a small payment.
4. Realise D and give it to the agent.
5. If D realised A for the agent, go to 1; if D realised B for the agent, go to 2.

Question 3.2

Answer the following questions about the gridworld Markov decision process shown below:

The details are:



- The 5x5 grid has obstacles indicated by black tiles in three contiguous blocks, at coordinates (1,1), (1,2), (1,3); at (3,1) and (4,1); and at (3,3).
- The agent starts at the state labelled S, at coordinates (0,0).
- Its goal state is labelled G, at (4,2). **Assume the goal state is absorbing.**
- Entering the goal state earns the agent 10 points.
- Actions that cause collisions with the boundary or obstacles keep the agent at its current state (with no cost collision).
- The red square at (2,2) is mud, which can slow the agent down. The agent successfully moves out of the red square with probability 0.2 and remains stuck in the mud with probability 0.8.

- Assume $\gamma = 0.8$.

a) What is the transition function for each action starting in the mud? That is, write out *T(s,a,s')* for each *s'* with non-zero transition probability, starting at s=(2,2).

T((2,2),up,(2,2)) = 0.8
T((2,2),up,(2,3)) = 0.2

T((2,2),down,(2,2)) = 0.8
T((2,2),down,(2,1)) = 0.2

T((2,2),right,(2,2)) = 0.8
T((2,2),right,(3,2)) = 0.2

T((2,2),left,(2,2)) = 1

b) Initialise all values to 0, and compute three iterations of (synchronous) *value iteration* for the gridworld above. What is the value function iterates (i.e. approximate values) for each state after?

i. the first iteration,
Note V(4,2) = 0
V(3,2) = 10
V(4,3) = 10

ii. the second iteration, and
V(3,2) = 10 (same as above)
V(4,3) = 10 (same as above)
V(2,2) = 0.2*0.8*V(3,2) = 0.2*0.8*10= 1.6
V(4,4) = 0.8*V(4,3) = 0.8*10 = 8

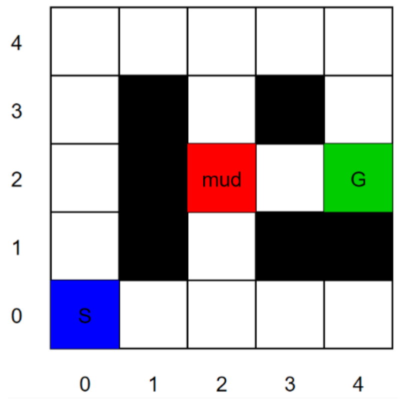
iii. the third iteration.
V(3,2) = 10 (same as above)
V(4,3) = 10 (same as above)
V(4,4) = 8 (same as above)
V(2,2) = 0.2*0.8*V(3,2) + 0.8*0.8*V(2,2) = 0.2*0.8*10 + 0.8*0.8*1.6 = 1.6 + 1.024 =2.624
V(2,1) = 0.8*V(2,2) = 0.8*1.6 = 1.28
V(2,3) = 0.8*V(2,2) = 0.8*1.6 = 1.28
V(3,4) = 0.8*0.8*10 = 6.4

All other iterates are equal to 0.

MODULE 4: LEARNING TO ACT

Question 4.1

Using the same grid world as above, assume now you do not know anything about the transitions or rewards. The gridworld is shown again below:



You choose to use SARSA to solve this problem and employ an epsilon-greedy exploration strategy with exploit probability $\epsilon = 0.8$ and exploration using uniform random sampling of any action otherwise.

Suppose you obtain the following observations:

s_t	a_t	s_{t+1}	reward	a_{t+1}
(2,1)	Up	(2,2)	0	Up
(2,2)	Up	(2,2)	0	Right
(2,2)	Right	(3,2)	0	Right
(3,2)	Right	(4,2)	10	Restart

a) What values does the Q-function attain if we initialise the Q-values to 0 and replay the experience in the table exactly two times? Use a learning rate $\alpha = 0.6$, and discount factor $\gamma = 0.8$. List only the non-zero approximate Q-values; all unstated Q-values are assumed to be equal to 0.

First experience replay:

- The first three actions result in no rewards and no non-zero values being propagated.
- The fourth observation, (s,a,s’) = ((3,2),Right, (4,2)) returns r =10, and is followed by a’ = restart.
- Based on this, the SARSA update for Q((3,2), Right) = 0+ 0.6(10 + 0.8*0 - 0) = 6.

Second experience replay:

- The first two actions result in no rewards and no non-zero values being propagated.
- The third observation, (s,a,s’) = ((2,2),Right, (3,2)) returns r =0 and is followed by a’ =Right. The next state (3,2), has nonzero Q-value Q((3,2), Right) = 6.
- Based on this, the SARSA update for Q((2,2), Right) = 0+ 0.6(0 + 0.8*6 - 0) = 2.88.
- The fourth observation, (s,a,s’) = ((3,2),Right, (4,2)) returns r =10, and is followed by a’ = restart. The sampled state-action pair ((3,2),Right) has nonzero Q-value Q((3,2), Right) = 6.
- Based on this, the SARSA update for Q((3,2), Right) = 6 + 0.6(10 + 0.8*0 - 6) = 8.4.

The final non-zero Q values are:
Q((2,2), Right) = 2.88
Q((3,2), Right) = 8.4

$$\hat{Q}^{\pi}(s,a) \leftarrow \hat{Q}^{\pi}(s,a) + \alpha\left(r + \gamma \hat{Q}^{\pi}(s',a') - \hat{Q}^{\pi}(s,a)\right)$$

b) Using these Q-values and the epsilon-greedy exploration strategy describe above, what are the probabilities of taking each action next time the SARSA agent gets stuck in the mud?

By these Q-values, the greedy action is to move *Right* from the mud state, s = (2,2), and the exploit probability is 0.8.

Random sampling the 4 directions with uniform probability gives every action 0.05.

Taken together, we have the following action probabilities:

Right = 0.8 + 0.05 = 0.05
Up = 0.05
Left = 0.05
Down = 0.05.

MODULE 5: REASONING ABOUT OTHER AGENTS

Question 5.1

In the game of *Morra*, each player shows either one or two fingers and announces a number between 2 and 4. If a player's number is equal to the sum of the number of fingers shown, then her opponent must pay her that many dollars. The payoff is the net transfer, so that both players earn zero if both or neither guess the correct number of fingers shown.

In this game, each player has 6 strategies: she may show one finger and guess 2; she may show one finger and guess 3; she may show one finger and guess 4; or she may show two fingers and guess one of the three numbers.

a) There are two *weakly dominated strategies* in Morra. What are they?

It never pays to put out one finger and guess that the total number of fingers will be 4, because the other player cannot put out more than two fingers.

Likewise, it never pays to put out two fingers and guess that the sum will be 2, because the other player must put down at least one finger.

Remaining payoffs to Player A (row player)

	one 2	one 3	two 3	two 4
one 2	0	2	-3	0
one 3	-2	0	0	3
two 3	3	0	0	-4
two 4	0	-3	4	0

b) Imagine that player A can read player B’s mind and guess how he plays before he makes his move. What *pure strategy* should player B use?

Use minimax reasoning. Minimise the greatest gain his opponent gains, which is equivalent to maximising the minimum it is guaranteed to receive.
Look the table above: Answer is (one 3)

c) Player B consults a textbook and decides to use randomisation to improve his performance in Morra. Ideally, if he can find a best mixed strategy to play, what would be his expected payoff?

Trick question - it depends what his opponent does!

Will his opponent adapt? What happens if Player B know that Player A favours one strategy over another? What happens if Player B know that Player A randomises uniformly over all strategies? …

Let's take the worst case and assume that Player A can adapt to whatever lottery Player B chooses and can itself randomise (i.e pick a mixed strategy). If Player A can still read his mind to see his mixed strategy, player B should choose to make Player A indifferent between pure strategies. Given this:

1. Because the game is symmetric, there is a symmetric mixed strategy Nash equilibrium.

2. Because the game is zero-sum, the symmetric mixed strategy Nash equilibrium must have payoffs to both players of 0.

d) One possible mixed strategy is to play show one finger and call "3" with probability 0.6, and to show two fingers and call "3" with probability 0.4 (and play the other strategies with probability 0). Is this a Nash equilibrium strategy? Assume that Player B is risk neutral with respect to the game payoffs.

We know that in the symmetric Nash equilibrium of a zero-sum game, the payoff is zero.

We also know that both players are indifferent to the pure strategies in the support set of their equilibrium mixed strategy. If they are indifferent, and they equal zero in total, they should be 0 individually too. If one is greater than the others, then Player A would not mix, they would exploit this.

To check, see if A's expected rewards for each pure action are zero. If they are, then Player A will have been induced to randomise by B's mixed strategy– this would be consistent with a Nash equilibrium.

Is this the case?

sigma_B = {P_B(one 3) : 0.6, P_B(two 3) : 0.4}

R_A(one 2, sigma_B) = 0.6*2 + 0.4*(-3) = 1.2 - 1.2 = 0.0
R_A(one 3, sigma_B) = 0.6*0 + 0.4*0 = 0
R_A(two 3, sigma_B) = 0.6*0 + 0.4*0 = 0
R_A(two 4, sigma_B) = 0.6*(-3) + 0.4*4 = -1.8 + 1.6 = -0.2

So, sigma_B is not consistent with a Nash equilibrium.

a **Nash equilibrium** is a strategy profile such that no agent can do better by unilaterally deviating from that profile.
Key results:

1. One of the great results of game theory, is that ***every finite game has at least one Nash equilibrium***.
2. A second great result (in the same paper) regards **symmetric games**. *These is a game in which all players have the same actions and symmetric payoffs given each individual's action*. Every ***finite symmetric game*** has at a symmetric Nash equilibrium, in which actions are played with the same probability by all players.
3. A **zero sum game** is *a game in which the payoffs for all players in each outcome sum to zero*. Another useful result of game theory is that in every ***finite two-player zero-sum game***, every Nash equilibrium is equivalent to a mixed-strategy minimax outcome.
4. Putting 2 and 3 together, in the equilibrium of a two-player, symmetric, zero-sum game, each player must receive a payoff of 0, and two-player, symmetric, zero sum games always have equilibria in symmetric strategies.

(NB: Any constant-sum game can be normalized to make it equivalent to a zero-sum game.)

MODULE 1: SEARCH

Completeness is the characteristic of an algorithm capable of finding a solution if one exists.

Optimality is the characteristic of an algorithm capable of always returning the path of least cost if one exists.

Complexity informs us of the required time and memory needed to solve a search problem generally based on worst case scenario and written using Big O notation. Above three are the measure of completeness.

Admissibility is the quality of a heuristic that never overestimates the cost of reaching the goal. Although related to optimality, admissibility is not a measure of performance.

If a solution exists, which of the following conditions must be true so that the **A* search algorithm** always **finds an optimal solution**:

(1) The search branching factor b must be finite because all successor nodes are expanded when a node is explored. **(2) The edge costs must be bounded above zero** because otherwise the algorithm may fall into an endless loop, expanding the same nodes repeatedly. While a h(n) that calculates the exact cost of the shortest path from n to a goal node will always find an optimal solution, it is not necessary to guarantee that the A* search algorithm always finds an optimal solution and is often too costly to calculate. If **(3) h(n) > 0 is an underestimate of the cost of the shortest path from n to a goal node**, then the A* search algorithm will never miss expanding the optimal path to the goal node.

DFS of binary tree: A **pre-order** traversal (Root, Left, Right)

An **in-order** traversal (Left, Root, Right)

A **post-order** traversal (Left, Right, Root)

Uniform Cost Search chooses the node on the frontier to explore by considering the path p with a minimum cost from the starting node g(p).

Greedy Best First Search chooses the node based on the estimated minimum cost (the heuristic) from the end of the path to the goal node h(p). A* Search combines these by choosing the path that minimizes g(p) + h(p).

MODULE 2: REASONING WITH CERTAINTY

For a formula to be in **conjunctive normal form (CNF)** it needs to be the conjunction of one or more clauses, such that each clause is made with disjunctions. $(\neg p \vee q) \wedge (r \vee \neg t \vee \neg p)$

MODULE 3: DECISION MAKING UNDER UNCERTAINTY

Axioms of rational preferences: preferences of a rational agent must obey certain rules.

Rational preferences imply behavior describable as maximization of expected utility

Completeness (for some reason R&N call this *Orderability*): $(o_1 \succ o_2) \vee (o_2 \prec o_1) \vee (o_1 \sim o_2)$

Transitivity: $(o_1 \succ o_2) \wedge (o_2 \succ C) \Rightarrow (o_1 \succ C)$

Monotonicity: $o_1 \succ o_2 \Rightarrow (p \geq q \Leftrightarrow [p : o_1, 1-p : o_2] \succeq [q : o_1, 1-q : o_2])$

Continuity: $o_1 \succ o_2 \succ C \Rightarrow \exists p \in [0,1][p : o_1, 1-p : C] \sim o_2$

Substitutability: $o_1 \sim o_2 \Rightarrow [p : o_1, 1-p : C] \sim [p : o_2, 1-p : C]$

Decomposability $[p : o_1, 1-p : [q : o_2, 1-q : o_3]] \sim [p : o_1, (1-p)q : o_2, (1-p)(1-q)o_3]$

Maximum Expected Utility (MEU)

Utility: a number that assigns the desirability of a state, MEU is the commonly used definition of "best" decision

Idea:

- Assigns utility function to each outcome (state) to represent the agent's preference
- "Best" decision maximizes the expected utility of the outcomes

In **value iteration**:

- Every iteration update both the values and (implicitly) the policy
- We don't track the policy, but taking the max over actions implicitly recomputes it
- **VI** converges to optimal value function through bellman optimality equation/backup;

In **policy iteration**:

- We do several passes that update utilities with fixed policy (each pass is fast because we consider only one action, not all of them)
- After the policy is evaluated, a new policy is chosen (slow like a value iteration pass)
- The new policy will be better (or we're done)
- **PI** converges to its optimal policy by acting greedily with respect to the action-value function; PI converges through the repeated application of policy improvement theorem(1. first evaluate the policy 2. Update the policy greedily with respect to the action value function)

Both are dynamic programs for solving MDPs and not online update algorithms(i.e. they reason their actions offline)

And-Or trees have **state nodes** and **action nodes**. These nodes can be either **closed** or **solved**.

State nodes require at least one child to be solved as the state node represents a choice and the agent can choose the solved action to take; **State nodes** are only closed if all their children are closed. This is since state nodes represent choices and therefore only need one solved state to continue; **State nodes** are closed once there is no possible solved state for the agent to select and therefore require all children to be closed; **State nodes** are solved when all its children are solved but not only. It requires only one child to be solved to be considered solved.

By the definition of the **And-Or tree**, a leaf state is solved if it is a goal-state.

Action nodes require all their children to be solved as these nodes represent the uncertainty in a system and so this ensures the success of the search; An **action node** with all its child solved would be a solved action node instead; **Action nodes** require all children to be solved in order to be solved. This is due to the uncertainty of the system and the algorithm finding a definite solution.

The solution to an AND-OR tree is a sub-tree which:

1. Which has a goal node at every leaf of the branch of the sub tree
2. Specifies one action at the OR node
3. Includes an outcome of every branch of an AND node

MODULE 4: LEARNING TO ACT

Epsilon-Greedy is a simple method to balance exploration and exploitation by choosing between exploration and exploitation randomly. The epsilon-greedy, where epsilon refers to the probability of choosing to explore, exploits most of the time with a small chance of exploring.

Exploration allows an agent to improve its current knowledge about each action, hopefully leading to long-term benefits. Improving the accuracy of the estimated action-values, enables an agent to make more informed decisions in the future.

Exploitation, on the other hand, chooses the greedy action to get the most reward by exploiting the agent's current action-value estimates. But by being greedy with respect to action-value estimates, may not actually get the most reward and lead to sub-optimal behavior.

When applying **reinforcement learning algorithm**, we should ensure that all state-actions pairs are explored infinitely many times as the time we train our model goes towards infinity.

Q-Learning estimates the reward for future actions and appends a value to the new state without following any greedy policy.

The **SARSA** algorithm, in general, is more conservative than Q-learning i.e. Solutions generated by SARSA tends to be safer or less risky than solutions generated by Q-learning

LEARNING:

- Q-learning: off-policy learning - Learns the values of an optimal policy no matter what it does
- SARSA: on-policy learning - on policy: Learns the value of the policy being followed. (e.g. act greedily most of the time and randomly other times)

CHOOSE NEXT STATE:

- Q-learning: based on current action and state
- SARSA: based on the reward of the previous action and state

WHEN TO USE:

- Q-learning: don't care about performance and want to learn the greedy policy that will eventually use
- SARSA: where care about the agent's performance during the process of learning

Intuition regarding **Upper Confidence Bound Exploration Strategy**:

1. Higher estimated reward v-hat_i is better (exploit)
2. Expect \true value" to be in some confidence interval around v_i
3. Confidence interval is large when number of trials n_i is small, shrinks in proportion to sqrt(n_i)
4. High uncertainty about move, larger exploration term
5. Sample more if number trials is much less than total number of trials

Can use this to sample nodes to expand in \selection" phase of MCTS, leads to UCT algorithm.

The **UCB1** formula balances exploitation and exploration

$$UCB1_i = \hat{v}_i + C\sqrt{\frac{\ln(N)}{n_i}}$$

v stands for exploitation, the higher the value of v, the more likely it is to be visited. C is a tunable bias parameter, which can change exploration; N is the total number of arms pulls, and n_i is the number of times arm i has been pulled.