NATIONAL UNIVERSITY OF SINGAPORE

SCHOOL OF COMPUTING

MIDTERM ASSESSMENT FOR
Semester 2 AY2018/19

CS3243: INTRODUCTION TO ARTIFICIAL INTELLIGENCE

March 7, 2019 Time Allowed: 1 Hour 30 Minutes

INSTRUCTIONS TO CANDIDATES

1. This examination paper contains **THREE (3)** parts and comprises **13** printed pages, including this page.
2. Answer **ALL** questions as indicated. Unless explicitly said otherwise, you **must explain your answer**. Unexplained answers will be given a mark of 0.
3. Use the space provided to write down your solutions. If you need additional space to write your answers, we will provide you with draft paper. clearly write down your **student number** and **question number** on the draft paper, and attach them to your assessment paper. Make sure you clearly indicate on the **body of your paper as well** if you used draft paper to answer any question.
4. This is a **CLOSED BOOK** examination.
5. Please fill in your **Matriculation Number** below; **DO NOT WRITE YOUR NAME**.
6. For your convenience, we include an overview section of important definitions and algorithms from class at the end of this paper.

MATRICULATION NUMBER: _____

EXAMINER'S USE ONLY		
Part	Mark	Score
I	12	
II	18	
III	20	
TOTAL	50	

In Parts I, II, and III, you will find a series of short essay questions. For each short essay question, give your answer in the reserved space in the script.

Part I
Uninformed and Informed Search

(12 points) Short essay questions. Answer in the space provided on the script.

Consider the following game called **jumping Jack**: a player (called Jack) needs to jump from one platform to another, in order to reach a door. Jack may only jump to other platforms (or he will fall, which will make him lose!), and may do so under the following rules:

(a) Jack may only jump to platforms that are of distance 0 or 1 horizontally.

(b) Jack may only jump to platforms that are no more than one step higher than him; he can also jump to platforms on the same height, or that are arbitrarily lower than him.

See Figure I.1 for an illustration of legal jumps leading from Jack's initial position to a goal node. The cost of

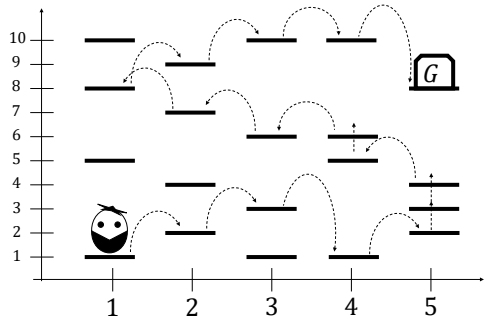


Figure I.1: An illustration of the Jumping Jack game. The goal door is marked with G. The dotted arrows mark one legal path from Jack's initial position to the goal; note that this is not the shortest path!

jumping from one platform to another is always 1.

1. (3 points) Formalize the Jumping Jack game as a search problem; what are the states? What are the actions? What is the cost function? You do not need to explain your answers.

   Solution: A state should be the platform that Jack is on; actions are valid jumps from one platform to another; cost is 1 always.

2. (9 points) Design two **non-trivial admissible** heuristics for the Jumping Jack game; by non-trivial we mean the heuristic you propose cannot be the all-zero heuristic, nor the optimal heuristic. **Your heuristics should be designed for a general instance of the game, not the specific illustration given in Figure I.1**.

   Do your heuristics dominate one another? If so, explain why, if not - provide a counterexample.

   Solution: We can take many options here, but one easy option is to play with horizontal distance, i.e. $h(v)$ is the horizontal distance from v to the goal. Note that vertical distance is **not admissible** since Jack can drop arbitrarily far, and it is never mentioned that the goal must be above the initial position. A variation here can be that $h(v)$ is the horizontal distance from v to the goal, plus 1 if they're not on the same height. A yet more refined variation can be that $h(v)$ is the horizontal distance from v to the goal, plus 1 if v is above the goal, and plus the vertical distance if it's below the goal. We can also have rule relaxation heuristics here: heuristics that result from relaxing the rules of the game.

Part II
Adversarial Search

(18 points) Short essay questions. Answer in the space provided on the script.

1. (10 points) Consider the minimax search tree shown in Figure II.1. The MAX player controls the black nodes (s, b_1 and b_2) and the MIN player controls the white nodes a_1 and a_2. Square nodes are terminal nodes with utilities specified with respect to the MAX player. Suppose that we use the α-β pruning algorithm (reproduced in Figure IV.7), in the direction from **right to left** to prune the search tree. Complete the values for A, B and

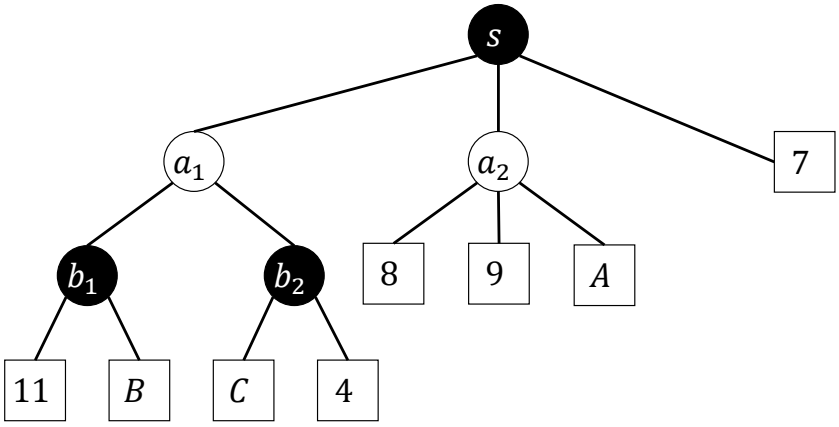


Figure II.1: Extensive form game with missing values.

C in the terminal nodes in order to make the statement below true. You may assume that A, B and C are **positive integers**.

(a) (8 points) In order to ensure that **no arcs are pruned**, it must be the case that:

> **Solution:** $A > 7, B < C, C > 8$. The MAX player can choose to be paid 7 by taking the rightmost arc. Therefore, in order to continue exploring a_2, it must be that $A > 7$ (else the MIN player can guarantee $\beta(a_2) \leq A \leq 7$ and the subtree rooted in a_2 is pruned). Next, we note that $\alpha(s) = 8$ after exploring the righthand side (assuming that $A > 7$ and that A is an integer, $A \geq 8$). Therefore, after exploring b_2, if we see $C \leq 8$ there is no point in exploring the subtree rooted in a_1 further. Finally, we note that after exploring the tree rooted in b_2 we have $\alpha(b_2) = C$, and therefore $\beta(a_1) = C$ as well; in order to check the terminal node of 11, we must have that $B < C$. If $B \geq C$ then $\alpha(b_1) \geq B \geq C$ and there is no point in checking the subtree rooted in b_1 further.

(b) (2 points) Under the values you found for A, B and C above, what is the payoff to player 1 in a sub-perfect Nash equilibrium for this game, assuming that $B \leq 11$? You do not need to explain your answer.

> **Solution:** As computed above, $A > 7, B < C, C > 8$. We have $\beta(a_1) = \min\{C, \max\{B, 11\}\}$: either the MIN player picks b_2 and pays C, or they pick b_1 and pay the maximum of B and 11. Since $B \leq 11$ this boils down to $\min\{C, 11\}$.

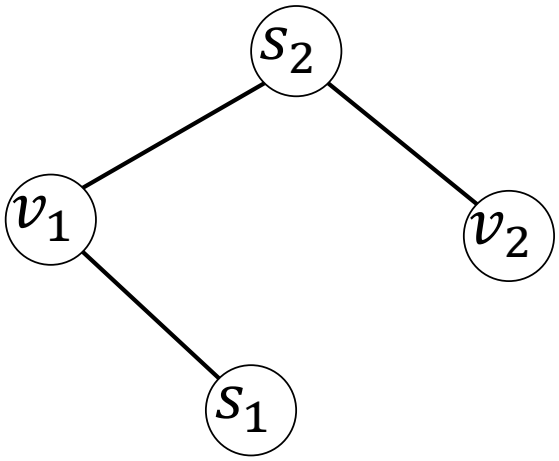
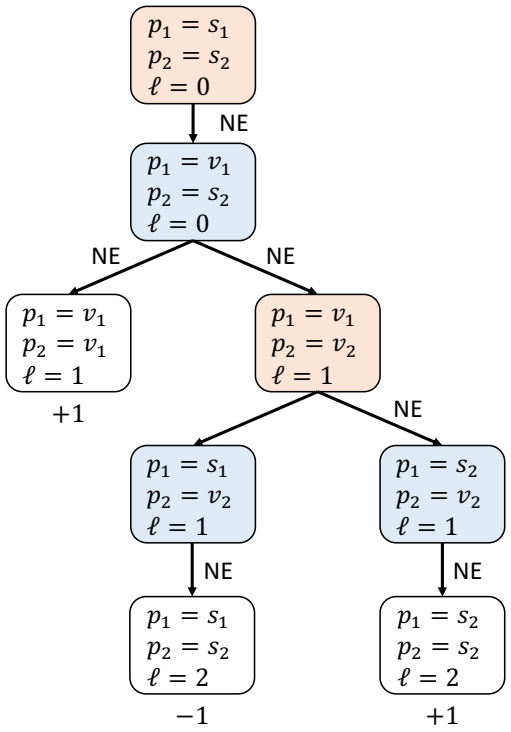


Figure II.2: Pursuer-evader graph; the pursuer starts in position s_1 and the evader starts in position s_2.

2. (8 points) Consider a two player game described as follows: we have a *pursuer* (player 1, the MAX player) and an *evader* (player 2, the MIN player). The evader wants to not be caught, and the pursuer tries to catch them. Both players start in different positions on an undirected graph $G = \langle V, E \rangle$, denoted s_1 (pursuer's start position) and s_2 (evader's start position). Both players take turns to make moves, with the pursuer making the first move. If a player is in node $v \in V$, they can move to any of the neighbors of v. The pursuer catches the evader if they both share a node, in which case the pursuer gets 1 point and the evader gets -1. If the evader manages to not get caught after making k steps[1], then the evader wins and gets 1 point (the pursuer gets -1 points). **if the evader gets caught on the k-th step, they lose**. Consider the graph in Figure II.2, where the pursuer starts in position s_1 and the evader starts in position s_2. Suppose that $k = 2$. Draw the extensive form game tree that results. In every node, write the position of the pursuer, the position of the evader, and the total number of steps that the evader has taken so far. Mark down the edges corresponding to actions in sub-perfect Nash equilibria on your game tree (by writing ``NE'' next to them).

Solution: Here the orange nodes are controlled by the pursuer, and the blue by the evader. White nodes are terminal nodes.

[1]For example, if $k = 3$ then the evader needs to move 3 times without being caught



Note that the game has *two SPNE*, since the evader is indifferent between getting caught in the beginning or getting caught later. The counter ℓ is the number of steps that the evader has taken so far.

Part III
Constraint Satisfaction Problems

(20 points) Short essay questions. Answer in the space provided on the script.

1. (10 points) In the *vertex cover* problem we are given a graph $G = \langle V, E \rangle$. We say that a vertex v *covers* an edge $e \in E$ if v is incident on the edge e. We are interested in finding a *vertex cover*; this is a set of vertices $V' \subseteq V$ such that every edge is covered by some vertex in V'. In what follows you may **only** use variables of the form x_v where $x_v = 1$ if v is part of the vertex cover, and is 0 otherwise. When writing the constraints you may **only use**

   - Standard mathematical operators ($+, -, \times, \div$) and
   - Standard logical and set operators ($\forall, \exists, \vee, \wedge$ and $x \in X, X \subseteq Y$)

   (a) (5 points) Write down constraints that ensure that the condition "every edge $e \in E$ is covered by at least one vertex $v \in V$" holds, and a constraint that ensures that the total size of the vertex cover is no more than k. You may assume that edges are represented as sets of two vertices, i.e. $e = \{u, v\}$ for some $u, v \in V$. You **need to justify** your answer here.

   Solution: $\forall \{u, v\} \in E : x_u + x_v \geq 1$ and $\sum_{v \in V} x_v \leq k$.

(b) (5 points) Consider the graph in Figure III.1. In what follows, If you have two constraints of the form

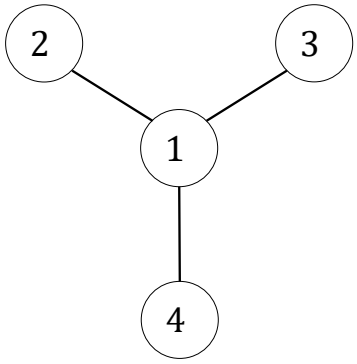


Figure III.1: Graph for Vertex Cover CSP part (b)

$A \leq B$ and $A \geq B$ you may merge them to a single constraint of the form $A = B$.

(i) (2 points) Write down the vertex cover CSP induced by Figure III.1 with a vertex cover size $k = 1$. Use binary constraints to describe the constraint "the vertex cover must be of size 1".

Solution: Since the size constraint is set to $k = 1$ and we use binary constraints, it means that $x_i + x_j \leq 1$ for all $i, j \in \{1, \ldots, 4\}$. We get

$$\begin{aligned} x_1 + x_2 &\geq 1 \\ x_1 + x_3 &\geq 1 \\ x_1 + x_4 &\geq 1 \\ x_1 + x_2 &\leq 1 \\ x_1 + x_3 &\leq 1 \\ x_1 + x_4 &\leq 1 \\ x_2 + x_3 &\leq 1 \\ x_2 + x_4 &\leq 1 \\ x_3 + x_4 &\leq 1 \end{aligned}$$

which narrows down to

$$\begin{aligned} x_1 + x_2 &= 1 \\ x_1 + x_3 &= 1 \\ x_1 + x_4 &= 1 \\ x_2 + x_3 &\leq 1 \\ x_2 + x_4 &\leq 1 \\ x_3 + x_4 &\leq 1 \end{aligned}$$

(ii) (3 points) Suppose that in our backtracking search we began by setting the value of $X_1 = 0$ (i.e. we say that the vertex 1 is not part of our vertex cover), and then run the `AC3` algorithm. What will the `AC3` algorithm do? Assume that edges are processed in increasing lexicographic order, i.e. the queue is initialized to

$$(x_1, x_2), (x_1, x_3), (x_1, x_4), (x_2, x_1), (x_2, x_3), (x_2, x_4), \ldots, (x_4, x_1), (x_4, x_2), (x_4, x_3).$$

You must explain your answer, but you do not have to provide a complete trace of the algorithm.

Solution: It is quite clear to us that the only solution here is to set $x_1 = 1$ and the rest to 0. What will `AC3` do if we assign $x_1 = 0$? It begins by processing the edges of the form (x_1, x_j). For each one of them, it does nothing, since the domains of x_2, x_3 and x_4 have values that make the constraint with x_1 consistent. When reaching the edge (x_2, x_1) it will have the constraint $x_1 + x_2 = 1$, which can only be satisfied by reducing x_2's domain to $\{1\}$. It will do nothing for $(x_2, x_3); (x_2, x_4)$. For (x_3, x_1) it will again reduce the domain of x_3 to $\{1\}$ for a similar reason (and add edges to the end of the queue).
However, when reaching (x_3, x_2) we will have a problem: no value in the reduced domain of x_3 will satisfy the constraint $x_2 + x_3 \leq 1$. Thus, the domain of x_3 is reduced to $D_3 = \emptyset$ after the REVISE(x_3, x_2) function is called, it will exit the REVISE(x_3, x_2) procedure; the subsequent check of D_3's size will result in `AC3` returning `false`.

2. (10 points) We have seen in class that if a CSP has a constraint graph that is a tree, it is possible to compute a satisfying assignment (or decide that one does not exist) in $\mathcal{O}(nd^2)$ time, where the number of variables is n, and the maximal size of variable domains is d. Consider a binary CSP whose constraint graph is *nearly a tree*: there exists one variable (without loss of generality assume that it is X_1) such that removing X_1 from the constraint graph will result in a tree-structured constraint graph; in other words, the constraint graph induced by the variables $X_2, \ldots, X_n$ is a tree, but the one induced by the variables $X_1, \ldots, X_n$ is not. Prove that there is a polynomial-time algorithm that can find a satisfying assignment (or decide that one does not exist) for nearly-tree structured CSPs.

Solution: The solution is simple: the CSP will have a satisfying assignment if and only if for some value $x \in X_1$ there is some satisfying assignment for the variables $X_2, \ldots, X_n$. So, fixing the value of X_1 to be some arbitrary x, we run the following procedure

1. For every $x \in D_1$:
   (a) Reduce the domains $D_2, \ldots, D_n$ so that they contain only values that are consistent with X_1, i.e. for every X_j, the constraint $C(x, y)$ is satisfied for every value $y \in D_j$. This can be done via forward checking in $\mathcal{O}(nd)$ time.
   (b) If none of the domains of $X_2, \ldots, X_n$ are empty, run the poly-time algorithm for tree structured CSPs on the reduced domains for $X_2, \ldots, X_n$. This takes $\mathcal{O}(nd^2)$ time.
   (c) If the algorithm returns a satisfying assignment $(y_2, \ldots, y_n)$, return $(x, y_2, \ldots, y_n)$
2. return `false`.

The algorithm's runtime is thus $\mathcal{O}(nd^3)$. If the algorithm returns an assignment, it must satisfy the CSP: all of the constraints of the form $C(X_1, X_j)$ are guaranteed to be satisfied, because we only use values for X_j that satisfy $C(x, X_j)$. The other constraints only involve constraints of the form $C(X_j, X_k)$ where $j, k \neq 1$, so we get a satisfying assignment by the tree algorithm. Suppose that there is some satisfying assignment to the CSP, say $(x_1, x_2, \ldots, x_n)$; our algorithm is guaranteed to find one: if it hasn't found any assignment up to the point that X_1 was set to x_1, it will find one on the iteration where X_1 is set to x_1: the tree algorithm will only consider variable assignments that agree with constraints involving $X_1 = x_1$ (one of which is $(x_2, \ldots, x_n)$) and since the tree algorithm is guaranteed to find a satisfying assignment when one exists, it will output either $(x_2, \ldots, x_n)$ or some other assignment that agrees with all constraints when $X_1 = x_1$.