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 assessment 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 indicate on the **body of your paper as well** if you used draft paper to answer any question.
4. This is a **CLOSED BOOK** assessment.
5. Please fill in your **Student 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.

STUDENT 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.

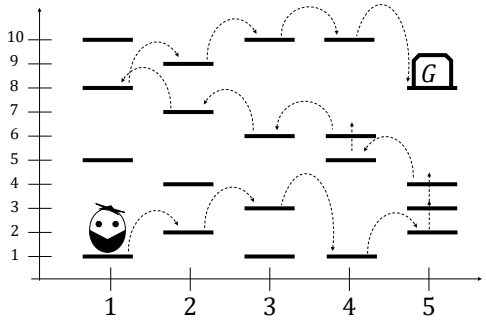

Figure I.1: An illustration of the Jumping Jack game. The goal door is marked with G. The dotted arrows mark one legal path from Jack's initial position to the goal; note that this is not the shortest path!

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 (this is the goal). Jack may only jump to other platforms (or he will fall, which will make him lose!), and may only jump to platforms that are

(a) of distance 0 or 1 horizontally.

(b) no more than one level 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 door. The cost of 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. Given a platform in x horizontal position and y vertical position, you must refer to it as $p(x, y)$.

Solution:
States:

Actions:

Cost function:

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:
First admissible heuristic:

Why is it admissible:

Second admissible heuristic:

Why is it admissible:

Do the heuristics dominate one another? Why?

Part II
Adversarial Search

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

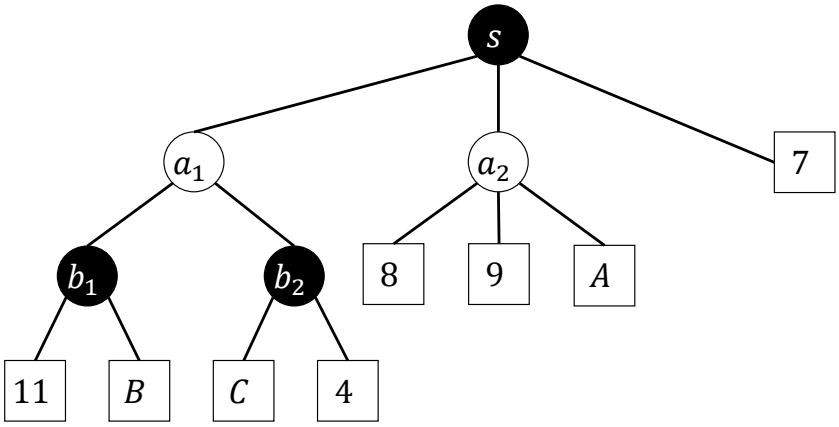

Figure II.1: Extensive form game with missing values.

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.

   (a) (8 points) Complete the values for A, B and C in the terminal nodes in order to make the statement below true. You may assume that A, B and C are **positive integers**. In order to ensure that **no arcs are pruned**, it must be the case that:

   Solution: $A >$ _____
   $B <$ _____
   $C >$ _____

   Explanation:

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

   Solution:

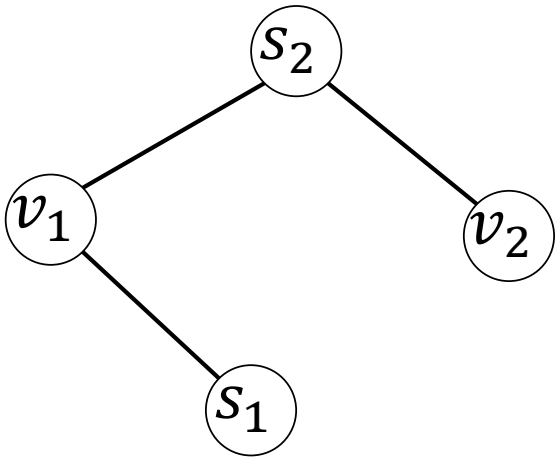

Figure II.2: Pursuer-evader graph; the pursuer starts in position s_1 and the evader starts in position s_2.

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 (but they **cannot stay in the same place**). 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 makes k steps without getting caught[1], then the evader wins and gets 1 point (the pursuer gets -1 points); **if the evader gets caught on the k-th step, it loses and gets -1 points**. 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 subgame-perfect Nash equilibria on your game tree (by writing `''NE''` next to them). You do not need to explain your answer to this question.

Solution:

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

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**

   - Numbers in $\mathbb{R}$, variables for constants (e.g. $a, b, c, d, \dots$)
   - Standard mathematical operators ($+, -, \times, \div, \sum, \prod, \dots$) and
   - Standard logical and set operators ($\forall, \exists, \vee, \wedge$ and $x \in X, X \subseteq Y$)

   (a) (4 points) Write down constraints that ensure that the condition "every edge $e \in E$ is covered by at least one vertex 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:
   Every edge in E is covered by at least one vertex:

   The size of the vertex cover is no more than k:

   Explanation:

(b) (6 points) Consider the graph in Figure III.1.

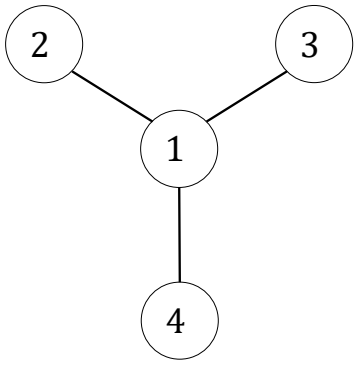

Figure III.1: Graph for Vertex Cover CSP part (b)

(i) (2 points) Use only binary constraints to describe the vertex cover CSP induced by Figure III.1 with a vertex cover size $k = 1$. You **must use binary constraints** to describe the condition "the vertex cover must be of size 1". If you have two constraints of the form $A \leq B$ and $A \geq B$ you may merge them to a single constraint of the form $A = B$.

Solution:

(ii) (4 points) Suppose that in our backtracking search we began by setting X_1 to be 0 (i.e. vertex 1 is not part of the vertex cover), and then run the `AC3` algorithm. What will the `AC3` algorithm do? Assume that arcs 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:

2. (10 points) We have seen in class that if the constraint graph of a CSP 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 with variables $X_1, \ldots, X_n$ whose domains are $D_1, \ldots, D_n$ and 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 (in the number of variables n and the max. domain size d) that can find a satisfying assignment (or decide that one does not exist) for nearly-tree structured CSPs.

Solution:

Part IV
Summary of Course Material

The following summary contains key parts from the course lecture notes. It **does not** offer a complete coverage of course materials. You may use any of the claims shown here without proving them, unless specified explicitly in the question.

1 Search Problems

1.1 Uninformed Search

We are interested in finding a solution to a fully observable, deterministic, discrete problem. A search problem is given by a set of *states*, where a *transition function* $T(s, a, s')$ states that taking action a in state s will result in transitioning to state s'. There is a cost to this defined as $c(s, a, s')$, assumed to be non-negative. We are also given an initial state (assumed to be in our frontier upon initialization). The *frontier* is the set of nodes in a queue that have not yet been explored, but will be explored given the order of the queue. We also have a *goal test*, which for a given state s outputs "yes" if s is a goal state (there can be more than one). We have discussed two search

```
function TREE-SEARCH(problem) returns a solution, or failure
    initialize the frontier using the initial state of problem
    loop do
        if the frontier is empty then return failure
        choose a leaf node and remove it from the frontier
        if the node contains a goal state then return the corresponding solution
        expand the chosen node, adding the resulting nodes to the frontier
```

Figure IV.1: The tree search algorithm

```
function GRAPH-SEARCH(problem) returns a solution, or failure
    initialize the frontier using the initial state of problem
    initialize the explored set to be empty
    loop do
        if the frontier is empty then return failure
        choose a leaf node and remove it from the frontier
        if the node contains a goal state then return the corresponding solution
        add the node to the explored set
        expand the chosen node, adding the resulting nodes to the frontier
            only if not in the frontier or explored set
```

Figure IV.2: The graph search algorithm

variants in class, tree search (Figure IV.1) and graph search (Figure IV.2). They differ in the fact that under graph search we do not explore nodes that we have seen before.
The main thing that differentiates search algorithms is the order in which we explore the frontier. In breadth-first search we explore the shallowest nodes first; in depth-first search we explore the deepest nodes first; in uniform-cost search (Figure IV.3) we explore nodes in order of cost.

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
    node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    frontier ← a priority queue ordered by PATH-COST, with node as the only element
    explored ← an empty set
    loop do
        if EMPTY?(frontier) then return failure
        node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
        if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
        add node.STATE to explored
        for each action in problem.ACTIONS(node.STATE) do
            child ← CHILD-NODE(problem, node, action)
            if child.STATE is not in explored or frontier then
                frontier ← INSERT(child, frontier)
            else if child.STATE is in frontier with higher PATH-COST then
                replace that frontier node with child
```

Figure IV.3: The uniform cost search algorithm

Property	BFS	UCS	DFS	DLS	IDS
Complete	Yes	Yes	No	No	Yes
Optimal	No	Yes	No	No	No
Time	$\mathcal{O}(b^d)$	$\mathcal{O}(b^{1+\lfloor \frac{C^*}{\epsilon} \rfloor})$	$\mathcal{O}(b^m)$	$\mathcal{O}(b^\ell)$	$\mathcal{O}(b^d)$
Space	$\mathcal{O}(b^d)$	$\mathcal{O}(b^{1+\lfloor \frac{C^*}{\epsilon} \rfloor})$	$\mathcal{O}(bm)$	$\mathcal{O}(b\ell)$	$\mathcal{O}(bd)$

Table 1: Summary of search algorithms' properties

We have also studied variants where we only run DFS to a certain depth (Figure IV.4) and where we iteratively deepen our search depth (Figure IV.5). Table 1 summarizes the various search algorithms' properties.

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
      cutoff_occurred? ← false
      for each action in problem.ACTIONS(node.STATE) 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
```

Figure IV.4: The depth-limited search algorithm

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
    for depth = 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH(problem, depth)
        if result ≠ cutoff then return result
```

Figure IV.5: The iterative deepening search algorithm

We also noted that no deterministic search algorithm can do well in general; in the worst case, they will all search through the entire search space.

1.2 Informed Search

Informed search uses additional information about the underlying search problem in order to narrow down the search scope. We have mostly discussed the A^* algorithm, where the priority queue holding the frontier is ordered by $g(v) + h(v)$, where $g(v)$ is the distance of a node from the source, and $h(v)$ is a *heuristic estimate* of the distance of the node from a goal node. There are two key types of heuristic functions

Definition 1.1. A heuristic h is *admissible* if it never overestimates the distance of a node from the nearest goal; i.e.

$$\forall v : h(v) \leq h^*(v),$$

where $h^*(v)$ is the *optimal heuristic*, i.e. the true distance of a node from the nearest goal.

Definition 1.2. A heuristic h is *consistent* if it satisfies the triangle inequality; i.e.

$$\forall v, v' : h(v) \leq h(v') + c(v, v')$$

where $c(v, v')$ is the cost of transitioning from v to v'.

We have shown that A^* with tree search is optimal when the heuristic is admissible, and is optimal with graph search when the heuristic is consistent. We also showed that consistency implies admissibility but not the other way around, and that running A^* with an admissible inconsistent heuristic with graph search may lead to a sub-optimal goal.

2 Adversarial Search

An extensive form game is defined by V a set of nodes and E a set of directed edges, defining a tree. The root of the tree is the node r. Let $V_{\max}$ be the set of nodes controlled by the MAX player and $V_{\min}$ be the set of nodes controlled by the MIN player. We often refer to the MAX player as player 1, and to the MIN player as player 2. A *strategy* for the MAX player is a mapping $s_1 : V_{\max} \to V$; similarly, a strategy for the MIN player is a mapping $s_2 : V_{\min} \to V$. In both cases, $s_i(v) \in chld(v)$ is the choice of child node that will be taken at node v. We let $\mathcal{S}_1, \mathcal{S}_2$ be the set of strategies for the MAX and MIN player, respectively.
The leaves of the minimax tree are *payoff nodes*. There is a payoff $a(v) \in \mathbb{R}$ associated with each payoff node v. More formally, the utility of the MAX player from v is $u_{\max}(v) = a(v)$ and the utility of the MIN player is $u_{\min}(v) = -a(v)$. The utility of a player from a pair of strategies $s_1 \in \mathcal{S}_1, s_2 \in \mathcal{S}_2$ is simply the utility they receive by the leaf node reached when the strategy pair (s_1, s_2) is played.

Definition 2.1 (Nash Equilibrium)**.** A pair of strategies $s_1^* \in \mathcal{S}_1, s_2^* \in \mathcal{S}_2$ for the MAX player and the MIN player, respectively, is a *Nash equilibrium* if no player can get a strictly higher utility by switching their strategy. In other words:

$$\forall s \in \mathcal{S}_1 : u_1(s_1^*, s_2^*) \geq u_1(s, s_2^*);$$
$$\forall s' \in \mathcal{S}_2 : u_2(s_1^*, s_2^*) \geq u_2(s_1^*, s')$$

Definition 2.2 (Subgame)**.** Given an extensive form game $\langle V, E, r, V_{\max}, V_{\min}, \vec{a} \rangle$, a subgame is a subtree of the original game, defined by some arbitrary node v set to be the root node r, and all of its descendants (i.e. its children, its children's children etc.), denoted by $desc(v)$. Terminal node payoffs the same as in the original extensive form game, and players still control the same nodes as in the original game.

Definition 2.3 (Subgame-Perfect Nash Equilibrium (SPNE))**.** A pair of strategies $s_1^* \in \mathcal{S}_1, s_2^* \in \mathcal{S}_2$ is a subgame-perfect Nash equilibrium if it is a Nash equilibrium for any subtree of the original game tree.

```
function MINIMAX-DECISION(state) returns an action
  return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))

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

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

Figure IV.6: The minimax algorithm (note a typo from the AIMA book - s should be $state$).

Figure IV.6 describes the minimax algorithm, which computes SPNE strategies for the MIN and MAX players, as we have shown in class. We discussed the α-β pruning algorithm as a method of removing subtrees that need not be explored (Figure IV.7).

3 Constraint Satisfaction Problems

A constraint satisfaction problem (CSP) is given by a set of variables $X_1, \ldots, X_n$, each with a corresponding domain $D_1, \ldots, D_n$ (it is often assumed that domain sizes are constrained, i.e. $|D_i| \leq d$ for all $i \in [n]$). Constraints specify relations between sets of variables; we are given $C_1, \ldots, C_m$ constraints. The constraint C_j depends on a subset of variables and takes on a value of "true" if and only if the values assigned to these variables satisfy C_j. Our objective is to find an assignment $(y_1, \ldots, y_n) \in D_1 \times \cdots \times D_n$ of values to the variables such that $C_1, \ldots, C_m$ are all satisfied. A binary CSP is one where all constraints involve at most two variables. In this case, we can write the relations between variables as a *constraint graph*, where there is an (undirected) edge between X_i and X_j if there is some constraint of the form $C(X_i, X_j)$.

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

function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Figure IV.7: The α-β pruning algorithm.

In class we discussed *backtracking search* (Figure IV.8), which is essentially a depth-first search assigning variable values in some order. Within backtracking search, we can employ several heuristics to speed up our search.

```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return BACKTRACK({ }, csp)

function BACKTRACK(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
      if value is consistent with assignment then
          add {var = value} to assignment
          inferences ← INFERENCE(csp, var, value)
          if inferences ≠ failure then
              add inferences to assignment
              result ← BACKTRACK(assignment, csp)
              if result ≠ failure then
                  return result
      remove {var = value} and inferences from assignment
  return failure
```

Figure IV.8: Backtracking search

1. When selecting the next variable to check, it makes sense to choose:
   (a) the most constrained variable (the one with the least number of legal assignable values).
   (b) the most constraining variable (the one that shares constraints with the most unassigned variables)
2. When selecting what value to assign, it makes sense to choose a value that is least constraining for other variables.

It also makes sense to keep track of what values are still legal for other variables, as we run backtracking search.

Forward Checking: as we assign values, keep track of what variable values are allowed for the unassigned variables. If some variable has no more legal values left, we can terminate this branch of our search. In more detail: whenever a variable X is assigned a value, the forward-checking process establishes arc consistency for it: for each unassigned variable Y that is connected to X by a constraint, delete from Y's domain any value that is inconsistent with the value chosen for X.

Arc Consistency: Uses a more general form of arc consistency; can be used when we assign a variable value (like forward checking) or as a preprocessing step.

Definition 3.1. Given two variables X_i, X_j, X_i is consistent with respect to X_j (equivalently, the arc (X_i, X_j) is consistent) if for any value $x \in D_i$ there exists some value $y \in D_j$ such that the binary constraint on X_i and X_j is satisfied with x, y assigned, i.e. $C_{i,j}(x, y)$ is satisfied (here, $C_{i,j}$ is simply a constraint involving X_i and X_j).

The AC3 algorithm (Figure IV.9) offers a nice way of iteratively reducing the domains of variables in order to ensure arc consistency at every step of our backtracking search. Whenever we remove a value from the domain of X_i with respect to X_j (the REVISE operation in the AC3 algorithm), we need to recheck all of the neighbors of X_i, i.e. add all of the edges of the form (X_k, X_i) where $k \neq j$ to the checking queue of the AC3 algorithm. We have seen in class that the AC3 algorithm runs in $\mathcal{O}(n^2d^3)$ time. In general, finding

```
function AC-3( csp) returns false if an inconsistency is found and true otherwise
  inputs: csp, a binary CSP with components (X, D, C)
  local variables: queue, a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    (X_i, X_j) ← REMOVE-FIRST(queue)
    if REVISE(csp, X_i, X_j) then
      if size of D_i = 0 then return false
      for each X_k in X_i.NEIGHBORS - {X_j} do
        add (X_k, X_i) to queue
  return true
```

```
function REVISE( csp, X_i, X_j) returns true iff we revise the domain of X_i
  revised ← false
  for each x in D_i do
    if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j then
      delete x from D_i
      revised ← true
  return revised
```

Figure IV.9: The AC3 Algorithm

a satisfying assignment (or deciding that one does not exist) for a CSP with n variables and domain size bounded by d takes $\mathcal{O}(d^n)$ via backtracking search; however, we have seen in class that if the constraint graph is a tree (or a forest more generally), we can find one in $\mathcal{O}(nd^2)$ time.

END OF EXAM PAPER