NATIONAL UNIVERSITY OF SINGAPORE

SCHOOL OF COMPUTING

MIDTERM ASSESSMENT FOR
Semester 2 AY2019/20

CS3243: INTRODUCTION TO ARTIFICIAL INTELLIGENCE

March 7, 2020 Time Allowed: 2 Hours

INSTRUCTIONS TO CANDIDATES

1. This assessment paper contains **FOUR (4)** 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. Allowed materials: an NUS approved calculator and a single A4 page of your own notes.
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	35	
II	15	
III	38	
IV	12	
TOTAL	100	

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 IV consists of True/False questions.

Part I: Uninformed and Informed Search

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

1. (20 points) Recall that a search problem can be defined by a *transition graph* $G = \langle \mathcal{S}, E, c \rangle$ where nodes are possible states $\mathcal{S}$, and there is a directed edge (n, n') if some action a can make state n transition to state n'. The weight of the edge (n, n') is given by the cost $c(n, a, n')$. For the following items, we consider a **consistent heuristic** h under a given transition graph $\langle \mathcal{S}, E, c \rangle$.

   (a) (10 points) Suppose we **add** new edges to the transition graph, leaving the heuristic h **unchanged**. Is the heuristic h still consistent? Prove this claim or provide a counterexample.

   > **Solution:** This is not true. Consider three states s, n, t with cost 1 to transition from $s \to n$ and from $n \to t$. The heuristic is going to be just the optimal heuristic, i.e. $h(s) = 2$, $h(n) = 1$ and $h(t) = 0$. If we add the edge $s \to t$ whose cost is 1, then the heuristic is not even admissible: $h(s) = 2$ but the true cost is 1.

   (b) (10 points) Suppose we **remove** edges from the transition graph, leaving the heuristic h **unchanged**. Is the heuristic h still consistent? Prove this claim or provide a counterexample.

   > **Solution:** This claim is true. If we remove an edge, the cost of getting from a node n to another node n' can only increase. Let us call the new cost function $\bar{c}(n, a, n')$, and we know $\bar{c}(n, a, n') \geq c(n, a, n')$. Since h is consistent, we have $h(n) \leq c(n, a, n') + h(n')$. But then: $h(n) \leq c(n, a, n') + h(n') \leq \bar{c}(n, a, n') + h(n')$. Thus, h is also consistent with respect to the new transition graph as well.

2. (15 points) In a variant of the game of Shanghai, there are 36 types of Mahjong tiles used. There are four tiles of each type, for a total of 144 tiles. These 144 tiles are placed in an 8×18 grid (see Figure I.1).

   In order to solve the puzzle, the player must remove all the tiles from the board. The player may only remove pairs of identical tiles at the *edges* of the grid — i.e., the player may only remove similar pairs of tiles, where both tiles in the pair have at least one edge unblocked (see Figure I.1 for details).

   In this problem, the **states** are different 8×18 grids; the **initial state** is a grid with all tiles in place; an **action** is a legal removal of a single pair of identical tiles (i.e. both with an exposed edge); the **transition model** results in the two tiles omitted from the grid; the **goal state** is an empty board; the **transition cost** is 1, **except** for states (boards) where no pairs can be removed, in which case the cost of all actions is ∞.

   Design an admissible heuristic for this search problem. Your heuristic may not be $h(n) = 0$ for all n, the (abstract) optimal heuristic, or a linear combination/simple function thereof. You may assume that the tile layout in the initial grid is solvable i.e. there is some path to a goal state. You must prove that your heuristic is admissible.

   > **Solution:** There are several heuristics the students can propose. One option is $\frac{\texttt{\# of tiles on board}}{2}$, but this heuristic is not 'informative' in the sense that it outputs the same value for all transitions from a state s (Students were awarded marks for this answer, as the oversight was our own, however next year's students will not get an easy way out!). Another alternative is `# of pairs with exposed edges`. A more refined option is for a given pair, count its depth - i.e. the minimal number of tiles that need to be removed before it has an exposed edge.

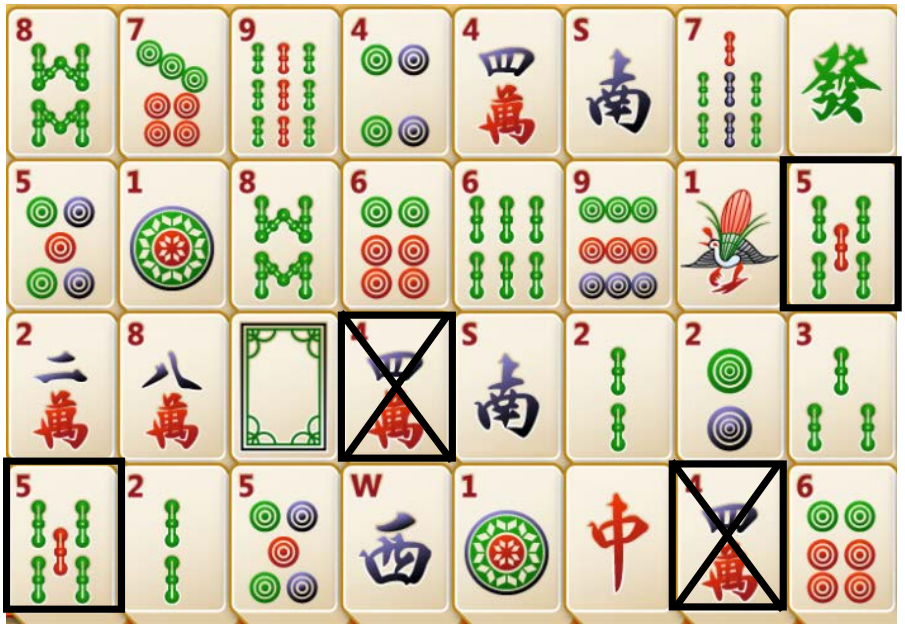


Figure I.1: A portion of the board in the Shanghai game. It is possible to remove the bottom left and rightmost '5' tiles (outlined in black), but it is impossible to remove the two '4' tiles (the crossed out blocks), as one of the '4' tiles has no exposed edge.

Part II: Adversarial Search

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

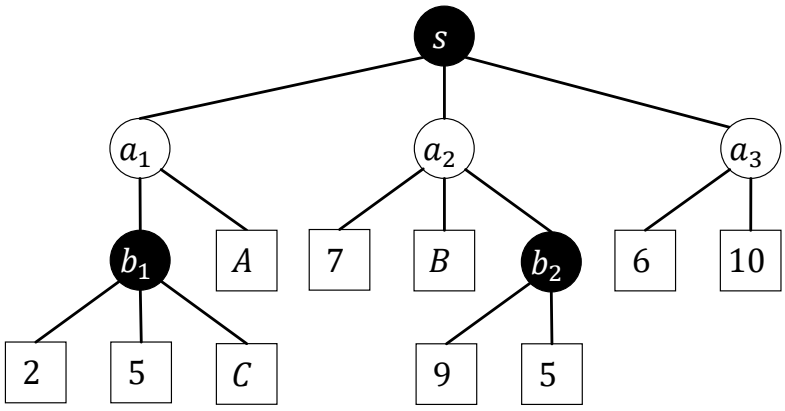


Figure II.1: Extensive form game with missing values.

1. (15 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, a_2 and a_3. 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 V.7), in the direction from **right to left** to prune the search tree.

   (a) (10 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: Let us start with B. When exploring B we know that $\beta(a_2) = 9$, and that $\alpha(s) = 6$. Thus, B must be strictly greater than 6 for us to continue. Note that since $B > 6$, $\min\{B, 9\} \geq 7$ and thus $\beta(a_2) = 7$ in the end of exploring this subtree, and $\alpha(s)$ is updated to $\alpha(s) = 7$. Now that we know this, it must be the case that $A > 7$, or the subtree rooted in b_1 is not explored at all. Finally, if $C \geq A$ then the nodes 5 and 2 aren't explored, as $\beta(a_1) = A$, thus $C < A$. Note that saying $C < 8$ is not sufficient, this is just a minimal value for A given what we did before.

(b) (5 points) Compute the minimax payoff to player 1, assuming that $A = 4, B = 9, C = 1$. You do not need to explain your answer.

Solution: The value is 7.

Part III: Constraint Satisfaction Problems

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

1. (20 points) An instance of the BIPARTITEMATCHING problem is given by two sets of vertices A and B, such that $|A| = |B| = n$, and a set of edges E (i.e. vertex pairs), where for every edge $\{a, b\}$, we have $a \in A$ and $b \in B$; E is given as a list of edges. A matching M is a subset of edges in E that are disjoint from one another (i.e. they share no vertices). Formally, if $\{a, b\}$ and $\{a', b'\}$ are in M, then $a \neq a'$ and $b \neq b'$. A matching M is complete if every vertex $a \in A$ belongs to some edge in the matching M.

   In what follows you may **only** use binary variables of the form $x_{a,b}$ where $x_{a,b} = 1$ if the edge $\{a, b\}$ is part of the matching, and is 0 otherwise. You may assume that the variable $x_{a,b}$ is non-existent when $\{a, b\} \notin E$.

   Constraint Language: when writing the constraints you may **only use**

   - Integer constants $(0, 1, 2, \dots)$.
   - Standard mathematical operators $(+, -, \times, \div, \sum, \prod, \dots)$.
   - Standard logical and set operators $(\forall, \exists, \vee, \wedge$ and $x \in X, X \subseteq Y$, `true/false` etc.).

   (a) (10 points) Write down the constraints describing the BIPARTITEMATCHING problem.

   Solution: M **is a valid matching:**

   $$x_{a,b} + x_{a,b'} \leq 1, \forall a \in A$$
   $$x_{a,b} + x_{a',b} \leq 1, \forall b \in B$$

   We can also write logical constraints

   $$x_{a,b} \vee x_{a,b'} = 0, \forall a \in A$$
   $$a_{a,b} \vee x_{a',b} = 0, \forall b \in B$$

   M **is a complete matching:**

   $$\sum_{a \in A} x_{a,b} \geq 1, \forall a \in A$$

   As a logical constraint this would be:

   $$\bigvee_{a \in A} x_{a,b} = 1, \forall a \in A$$

   Explanation:

   (b) (10 points) Items i and ii below refer to Figure III.1.

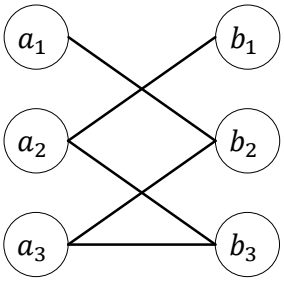


Figure III.1: An instance of the BIPARTITEMATCHING problem.

i. (5 points) Express the instance of the BIPARTITEMATCHING problem described in Figure III.1 using the constraints you have written in Part (a). For your convenience you may write $x_{i,j}$ instead of x_{a_i,b_j}.

Solution: A complete specification would be:

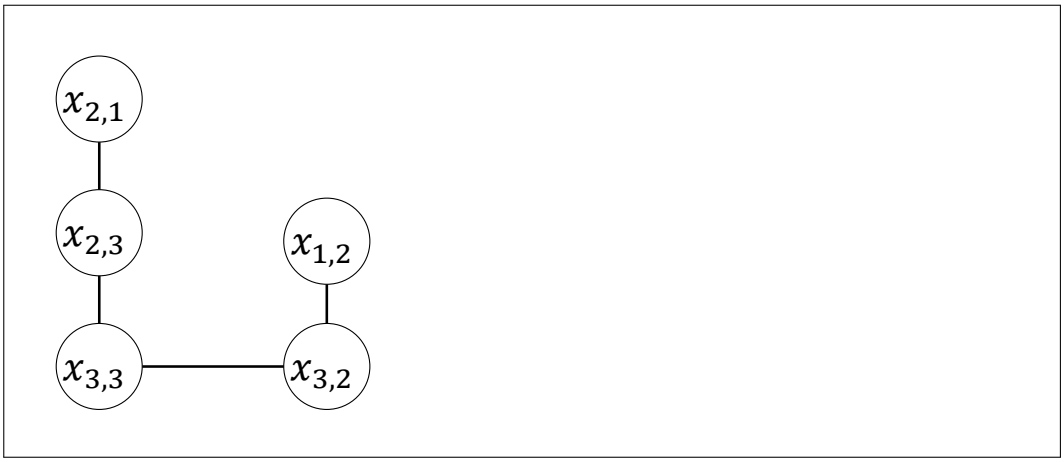
$$\begin{aligned} x_{1,2} &\leq 1 \\ x_{2,1} + x_{2,3} &\leq 1 \\ x_{3,2} + x_{3,3} &\leq 1 \\ x_{2,1} &\leq 1 \\ x_{1,2} + x_{3,2} &\leq 1 \\ x_{2,3} + x_{3,3} &\leq 1 \\ x_{1,2} &= 1 \\ x_{2,1} + x_{2,3} &= 1 \\ x_{3,2} + x_{3,3} &= 1 \end{aligned}$$

But there are redundant constraints, which we can reduce to:

$$\begin{aligned} x_{2,1} &\leq 1 \\ x_{1,2} + x_{3,2} &\leq 1 \\ x_{2,3} + x_{3,3} &\leq 1 \\ x_{1,2} &= 1 \\ x_{2,1} + x_{2,3} &= 1 \\ x_{3,2} + x_{3,3} &= 1 \end{aligned}$$

ii. (5 points) Draw the constraint graph for the BIPARTITEMATCHING instance described in Fig. III.1.

Solution:



2. (18 points) Consider the following CSP: there are four variables x_1, x_2, x_3, x_4 whose domain is $D = \{1, 2, 3\}$. They have the following constraints:

$$x_1 + x_2 \leq 3 \qquad x_1 + x_3 \geq 4$$
$$x_2 + x_3 \leq 3 \qquad x_3 + x_4 \leq 2$$

(a) (3 points) Draw the constraint graph for this CSP.

Solution: This is trivial - just an edge for each of the binary constraints above, meant to ensure that students know at the minimum what a constraint graph looks like (vast majority of the class got this right).

(b) (15 points) Before starting our backtracking search for a solution, we run the `AC3` Algorithm on this instance, with the initial edge queue below, where (x_1, x_2) is the start of the queue:

$$[(x_1, x_2), (x_1, x_3), (x_2, x_1), (x_2, x_3), (x_3, x_1), (x_3, x_2), (x_3, x_4), (x_4, x_3)]$$

Describe the run of the `AC3` algorithm for this instance; state what domains are reduced (write `none` if unchanged), and what edges are added to the end of the queue. You may use the additional slots below to describe additional edges in the queue, if need be.

Solution:

Edge	Reduced Domain (`none` if unchanged)	Edges Added to Queue
(x_1, x_2)	$D_1 = \{1, 2\}$	(x_3, x_1)
(x_1, x_3)	`none`	`none`
(x_2, x_1)	$D_2 = \{1, 2\}$	(x_3, x_2)
(x_2, x_3)	`none`	`none`
(x_3, x_1)	$D_3 = \{2, 3\}$	$(x_2, x_3), (x_4, x_3)$
(x_3, x_2)	$D_3 = \{2\}$	$(x_1, x_3), (x_4, x_3)$
(x_3, x_4)	$D_3 = \emptyset$	`none`
(x_4, x_3)	not reached	not reached

The `AC3` Algorithm will output: `false`

Part IV: True/False Questions

(12 points) For each one of the questions below, mark True/False in the appropriate box. Provide a short explanation for your answer in the space provided.

1. (3 points) The BFS algorithm is complete if the state space has infinite depth but finite branching factor.

   True: ☐ False: ☐

   Solution: this is true. Follows immediately from definitions of BFS and completeness. Recall that a search algorithm is complete if whenever there is a path from the initial state to the goal, the algorithm will find it. In particular, the existence of a path means that the goal exists in a finite depth, and therefore BFS eventually reaches it (finite branching factor means you don't get lost in infinite breadth search at some level).

2. (3 points) Given that a goal exists within a finite search space, the BFS algorithm is optimal if all step costs from the initial state are non-decreasing in the depth of the search tree. That is, for any given level of the search tree, all step costs are greater than the step costs in the previous level.

   True: ☐ False: ☐

   Solution: This is false. We could have two goal nodes in the same level with different costs but with the same depth. For example, single source node s, and two goals t_1, t_2 with $c(s, t_1) = 1$ and $c(s, t_2) = 7$. Arbitrary tie-breaking of the BFS algorithm may result in t_2 reached before t_1.

3. (3 points) Suppose that the A$\star$ search algorithm utilizes $f(n) = w \times g(n) + (1-w) \times h(n)$, where $0 \leq w \leq 1$ (instead of $f(n) = g(n) + h(n)$). For any value of w, an optimal solution will be found whenever h is a consistent heuristic.

   True: ☐ False: ☐

   Solution: this is false. When $w = 1$ we just get greedy search which is suboptimal (as seen in class).

4. (3 points) The α-β pruning algorithm returns a Subgame-Perfect Nash Equilibrium strategy.

   True: ☐ False: ☐

   Solution: this is false. α-β pruning may not return a valid strategy, let alone an SPNE.

Part V: 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.

V.1 Search Problems

V.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 V.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 V.2: The graph search algorithm

variants in class, tree search (Figure V.1) and graph search (Figure V.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 V.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 V.3: The uniform cost search algorithm

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

```
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 V.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 V.5: The iterative deepening search algorithm

V.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 V.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 V.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.

V.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.

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

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 V.3 (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 V.4 (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 V.5 (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 V.6: The minimax algorithm (note a typo from the AIMA book - s should be $state$).

Figure V.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 V.7).

V.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 $C(X_i, X_j)$.
In class we discussed *backtracking search* (Figure V.8), which is essentially a depth-first search assigning variable values in some order.

Heuristics for CSPs: Within backtracking search, we can employ several heuristics to speed up our 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).

```
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 V.7: The α-β pruning algorithm.

```
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 V.8: Backtracking search

(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.

Inference in CSPs: 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 V.6. 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 V.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.

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 V.9: The AC3 Algorithm

Solving CSPs: In general, finding 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. This is done by first topologically sorting the constraint graph, then ensuring that every parent node is consistent with its children, and finally assigning values to variables from the root down.
Alternatively, we can run a local search: start from an arbitrary variable assignment. If it is consistent with the constraints, return it. If not, pick a variable at random, set a value that maximally reduces the number of constraint violations, and repeat until `max_steps` is reached, see Figure V.10.

function MIN-CONFLICTS(*csp*, *max_steps*) **returns** a solution or failure
  inputs: *csp*, a constraint satisfaction problem
    max_steps, the number of steps allowed before giving up

  current ← an initial complete assignment for *csp*
  for i = 1 to *max_steps* **do**
    if *current* is a solution for *csp* **then return** *current*
    var ← a randomly chosen conflicted variable from *csp*.VARIABLES
    value ← the value v for *var* that minimizes CONFLICTS(*var*, v, *current*, *csp*)
    set *var* = *value* in *current*
  return *failure*

Figure 6.8 The MIN-CONFLICTS algorithm for solving CSPs by local search. The initial state may be chosen randomly or by a greedy assignment process that chooses a minimal-conflict value for each variable in turn. The CONFLICTS function counts the number of constraints violated by a particular value, given the rest of the current assignment.

Figure V.10: The Min-Conficts algorithm for solving CSPs by local search.

END OF EXAM PAPER