cmput 355 2025 practice questions 3 (with answers)

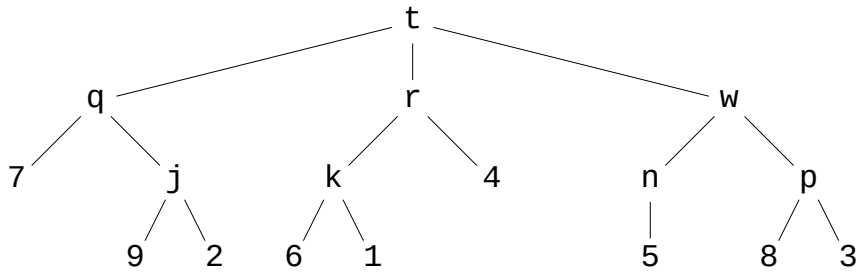
1. One measure of the effectiveness of a particular 2-player-game strategy is its average case performance. Another measure is worst-case performance. a) Explain how minimax is a worst-case performance measure. b) Explain why we learned minimax instead of average case analysis.

   Answer. a) The minimax score is the minimum the player will score, over all possible opponent strategies. b) Minimax is easier to compute than average case performance, requiring only time proportional to the number of nodes in the game graph. Average case analysis requires an initial decision about which assumptions to make about the distribution of opponent strategies (equi-probable? biased towards perfection?) and usually requires significantly more computation time.

2. a) In `alphabeta.py`, how is the depth parameter used? b) In `negamax.py`, how is the depth parameter used?

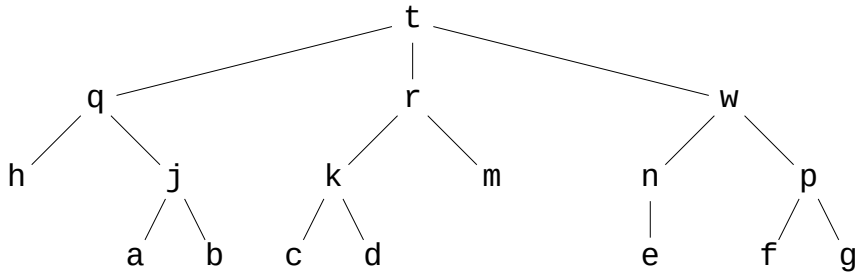
   Answer. a) Depth parity is used to determine whether the node is MAX or MIN. b) Depth is used only for printing, in order to determine the horizontal offset, which then shows the depth of the node in the recursions tree.

3. Give the minimax value for each non-terminal node.



   Answer. j 9 q 7 k 6 r 4 n 5 p 8 w 5 t 7

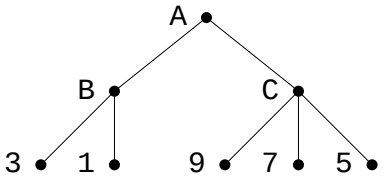
4. a) For this game tree, assuming children are ordered left to right, list nodes in the order that dfs learns minimax values. b) Repeat if children are ordered right to left.



Answer. dfs postorder: h a b j q c d k m r e n f g p w t

g f p e n w m d c k r b a j h q t

5. In each question, all node values are for MAX. Unless otherwise stated, MAX plays first.



a) For each node in the game tree, give the minimax value.

b) Repeat a) if MIN plays first (node values are for MAX).

c) For each node in the game tree, give the negamax value.

d) Repeat c), but first negate all leaf values.

Answer.

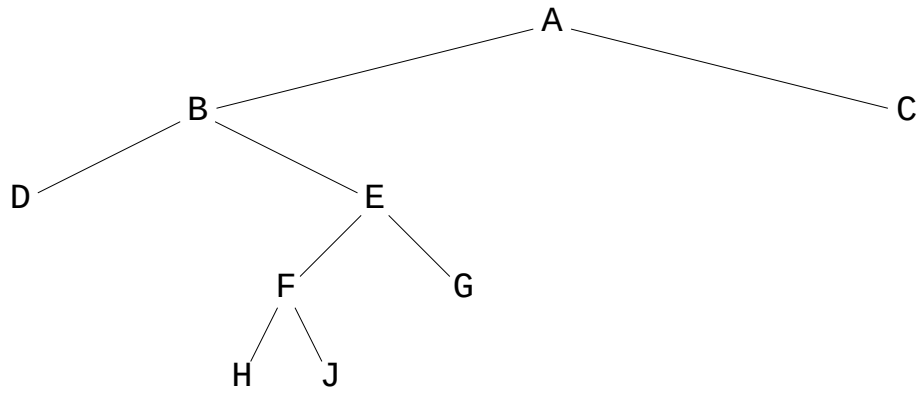
a) A _5_ B _1_ C _5_

b) A _3_ B _3_ C _9_

c) A _5_ B -1_ C -5_

d) A -3_ B 3_ C 9_

6. Complete the following table. For each node when it is first reached in alphabeta search, show the path to the root and each already-searched move option on this path.

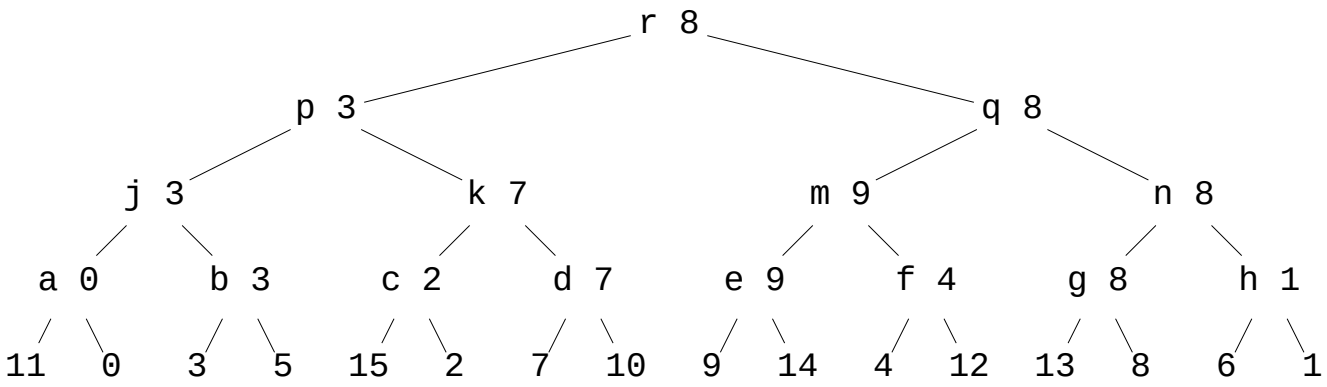


node	path-to-root	already-searched move-options on path-to-root
A	(A)	--
B	(B, A)	--
D	(D, B, A)	--
E	(E, B, A)	B-D
F	(F, E, B, A)	B-D
H	(H, F, E, B, A)	B-D
J	(J, F, E, B, A)	B-D, F-H
G	----------	----------
C	----------	----------

Answer.

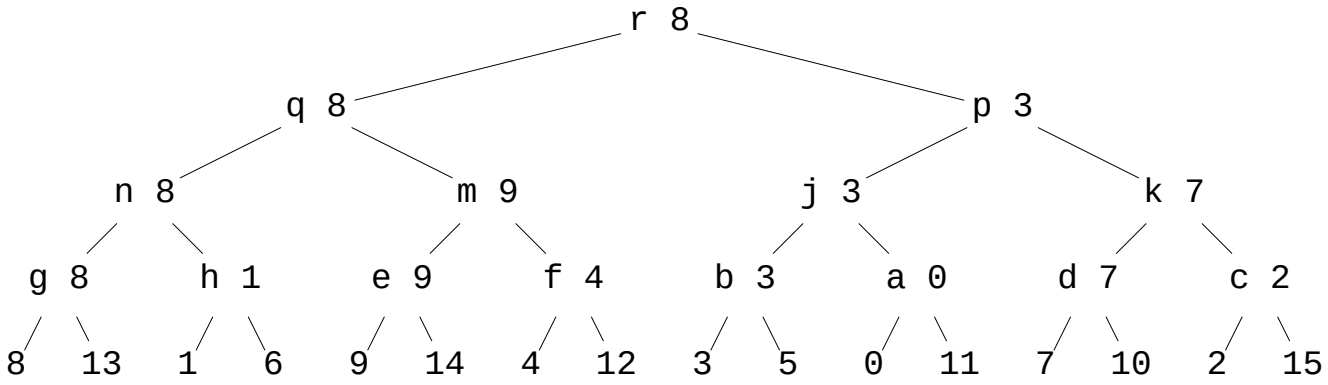
G	(G, E, B, A)	E-F, B-D
C	(C, A)	A-B

(To check these answers, you can give each leaf a value and trace alphabeta search.)



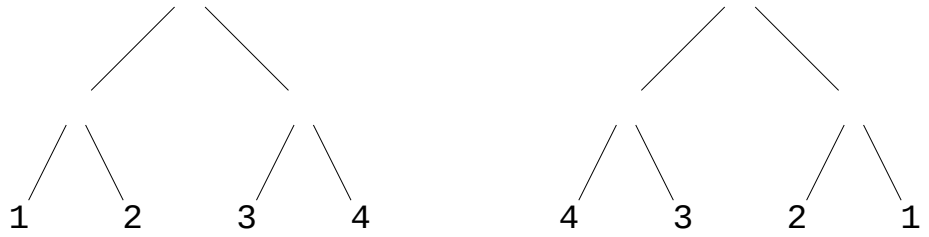
7. Below, redraw the above minimax tree so that each node's best move is its left child. E.g. at `r`, MAX's best move is to q (MAX minimax value 8), so the left child of `r` is `q` (as shown). Now you label the rest of the nodes.

Answer.

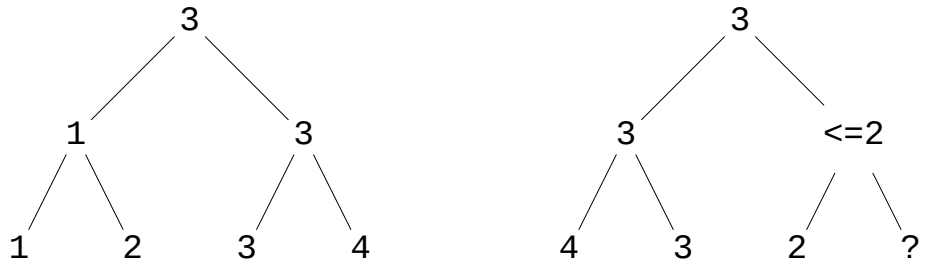


8. a) For each game tree, how many nodes are cut off by alpha-beta search?

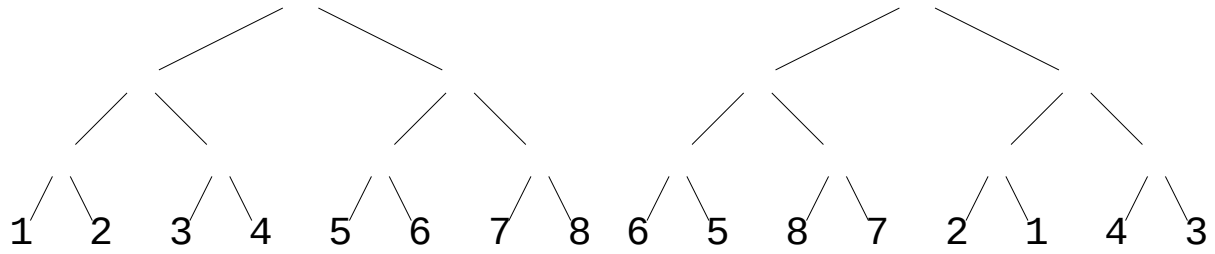
   b) After $\alpha\beta$-minimax runs, at each node give what is known about the minimax value there, e.g. 11, ≥ 3, ≤ 7, or ? if nothing is known.



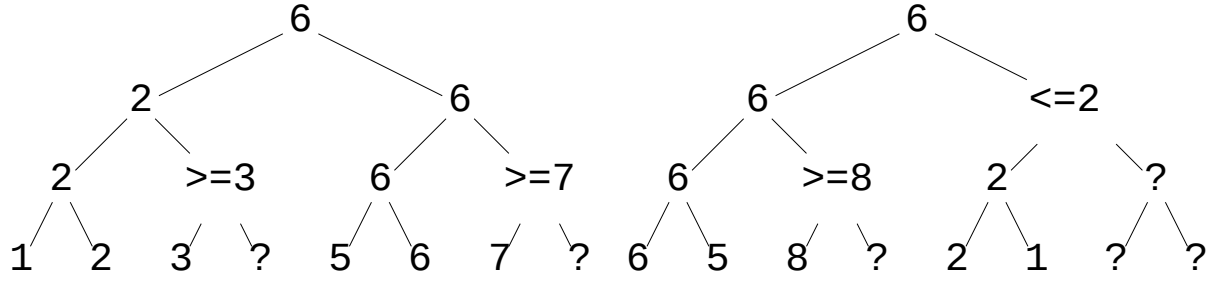
Answer. a) 0 (left tree) 1 (right) b) see below



9. Repeat the previous question for these game trees.



Answer. a) 2 (left tree) 4 (right) b) see below

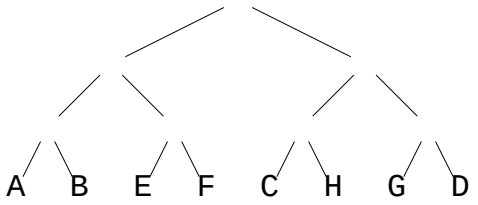


10. (a) Repeat the previous two questions if we change all leaf values to 1 and we use the test `alpha > beta ?` to check for cutoffs.

    (b) Repeat the previous two questions if we change all leaf values to 1 and we use the test `alpha >= beta ?` to check for cutoffs.

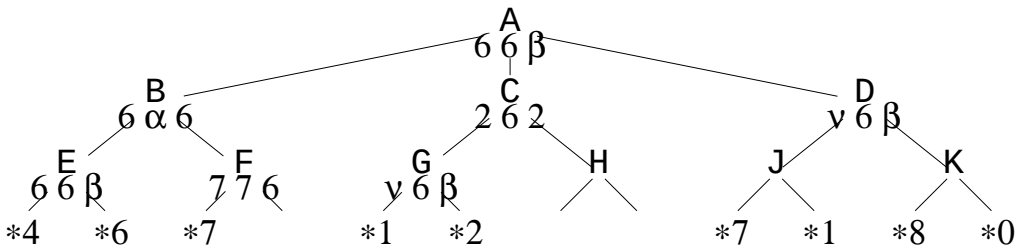
    Answer. (a) 0 cutoffs (b) same number of cutoffs as with perfect move ordering

11. All leaf nodes have value 1. Which leaf nodes are reached in an alphabeta search with cutoff test $\alpha \geq \beta$?



    Answer. A B E (not F) C H (not G, D)

12. This alphabeta search has just reached J. At each node, values are shown: minimax, alpha, beta. In order, in the rest of the search, at each non-leaf node where a change is made, show current mmx, alpha, beta values. We have shown you change 1 (there might be fewer than 8 changes).



    Answer.

change	node	minimax	alpha	beta	change	node	minimax	alpha	beta
1	J	?	6	inf	5	_K_	_8_	_8_	_7_
2	_J_	_7_	_7_	inf	6	_A_	_7_	_7_	inf
3	_D_	_7_	_6_	_7_	7	___	___	___	___
4	_K_	_?_	_6_	_7_	8	___	___	___	___

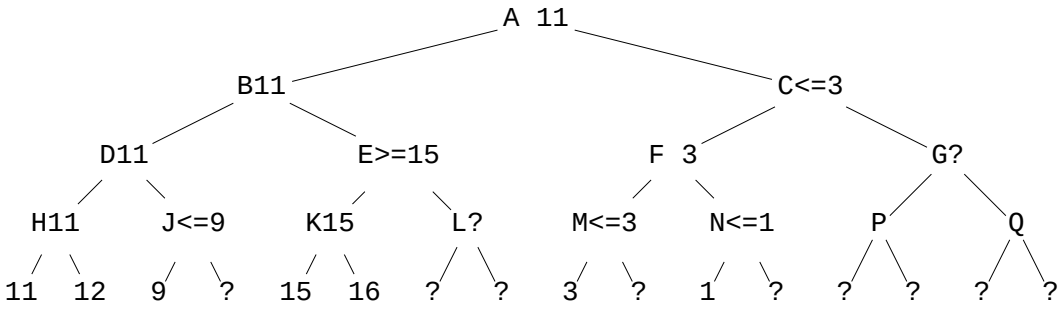
13. Fill in the blanks (missing code from `negamax.py`).

```
def negamax(d, T, V, v): # leaf scores for player-to-move
  if isTerminalNode(v,V): return V[v]
  val = NEGINF
  for c in T[v]: # for each child c of v

    val = max(__________________ , ________________________)
  return val
```

Answer. `val = max(val, -negamax(d+1, T, V, c))`

14. In this alphabeta search, right subtrees of J,E,M,N,C were pruned. Explain the reason for each prune by filling in the blanks. We have given the answer for node J.



Answer.

J: on path-to-root J-D-B-A, MAX has move option D-H so alpha >= 11,
MIN has move option J-9 so beta <= 9, so alpha >= beta

E: on path-to-root E-B-A , MAX has move option E-K so alpha >= 15,
MIN has move option B-D so beta <= 11 so alpha >= beta

M: on path-to-root M-F-C-A , MAX has move option A-B so alpha >= 11
MIN has move option M-3 so beta <= 3 so alpha >= beta

N: on path-to-root N-F-C-A , MAX has move option A-B so alpha >= 11
MIN has move option N-1 so beta <= 1 so alpha >= beta

C: on path-to-root C-A , MAX has move option A-B so alpha >= 11
MIN has move option C-F so beta <= 3 so alpha >= beta

15. Show the output when `negamax` is called on the game tree in question 5.

```
def negamax(d, T, V, v): # leaf scores for player-to-move
  print(d*'  ', v)
  if isTerminalNode(v,V):
    val = V[v]; print(d*'  ', v, 'leaf', val); return val
  val = NEGINF
  for c in T[v]: # for each child c of v
    val = max(val, -negamax(d+1, T, V, c))
  print(d*'  ', v, val)
  return val
```

Answer.

```
A
   B
     D
     D leaf 3
     E
     E leaf 1
   B -1
   C
     F
     F leaf 9
     G
     G leaf 7
     H
     H leaf 5
   C -5
 A 5
```

16. Run `alphabeta.py` on `t6.in`: it return a minimax value of 8 (last line of output: `A 8 8 999`). But run `negamax.py` on `t6.in`: it returns a minimax value of -1 (last line of output: `A -1`).

    a) Why did `negamax.py` not print out the alpha and beta values?

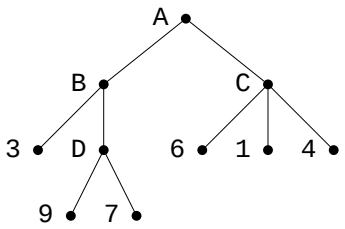
    b) Why did the two programs return different minimax values for the above input? Is one of them wrong? Explain.

    Answer. a) `negamax` is a modification of minimax (two code fragments, one for MAX and one for MIN, are replaced with one code fragment for player-to-move): it is not a version of alpha-beta minimax, so it does not use alpha/beta values and it does not prune any subtrees.

    (b) `alphabeta.py` assumes that all leaf node scores are for MAX. `negamax.py` assumes that all leaf node scores are for the player-to-move. For this input file, all leaf nodes are at depth 3, where ptm is the second-player, MIN. If you negate these leaf values you get file `t6nega.in`. When you run `negamax.py` on `t6nega.in`, you get the same minimax value as when you run `alphabeta.py` on `t6.in`.

17. The root is a MAX node. Leaf scores are for MAX. Give minimax values below.

    `A ___  B ___  C ___  D ___`



    Answer. `A _3_  B _3_  C _1_  D _9_`

18. Repeat the previous question if leaf scores are for MAX but root is MIN.

    Answer. `A _6_  B _7_  C _6_  D _7_`

19. Explain how to use `alphabeta.py` to answer the previous two questions.

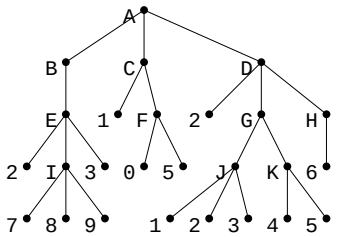
Answer. For the first question, just create the input graph (`td.in`) and run the program. For the second question:

method 1: on the input graph, negate all scores (`tdneg.in`), run the program, negate the answer.

method 2: create a new tree with one extra node: a new root that has the old root as its only child. now run the program on the new tree.

method 3: make a copy of the program and modify the copy so that the root player is assumed to be MIN.

20. The root is a MAX node. Leaf scores are for MAX. Give minimax values below.



A ___ B ___ C ___ D ___

E ___ F ___ G ___ H ___

I ___ J ___ K ___

Answer.

A _7_ B _7_ C _1_ D _2_

E _7_ F _5_ G _4_ H _6_

I _7_ J _1_ K _4_

21. Repeat the previous question if leaf scores are for MAX but root is MIN.

Answer.

A _1_ B _2_ C _1_ D _6_

E _2_ F _0_ G _3_ H _6_

I _9_ J _3_ K _5_

22. a) For a two-player game, a *principal variation* is a sequence of moves from the start of the game to a terminal position, where each move by each player is best possible. For question 18, give a principal variation. E.g. for question 17, a principal variation is 1.MAX A-B, 2.MIN B-3. Repeat for questions b) 20 and c) 21.

Answer.

a) 1.MIN A-C, 2.MAX C-6.

b) 1.MAX A-B, 2.MIN B-E, 3.MAX E-I, 4.MIN I-7.

c) 1.MIN A-C, 2.MAX C-1.

23. Using the blank-to-0, x-to-1, o-to-2 representation, express each board position below as a 9-digit number base 2 and then as a decimal number. Recall that $3^9 = 19683$. Show your work.

```
- - -        - - -        o o o
- - -        - - -        o o o
- - -        x - o        o o o
```

Answer. 000000000 000000102 2222222, 0 11 19682

24. In tic-tac-toe assume `x` moves first: thus for any reachable position, we can determine the player-to-move from the number of `x`'s and `o`'s. For each position below, give player-to-move minimax value (win, loss, draw). Use any program from class to answer this question.

```
   a b c
1  x - -    - x -    - - -    - - o    - x o    - - o
2  - - -    - - -    - x -    x - -    x - -    x o -
3  - - -    - - -    - - -    - - -    - - -    - - x
```

Answer. draw, draw, draw, win, draw, win.

25. For this tic-tac-toe position with `x` to play, how many nodes are in a proof tree that shows that `o` can at least draw? Give the number of nodes at each level of the tree: there will be one node at the root, corresponding to the position in the diagram below. (Such a tree will have all `x`-moves from the root, then for each a winning `o`-reply, then all `x`-moves, then for each a winning `o`-reply, and so on.)

```
. x .
. o o
. x .
```

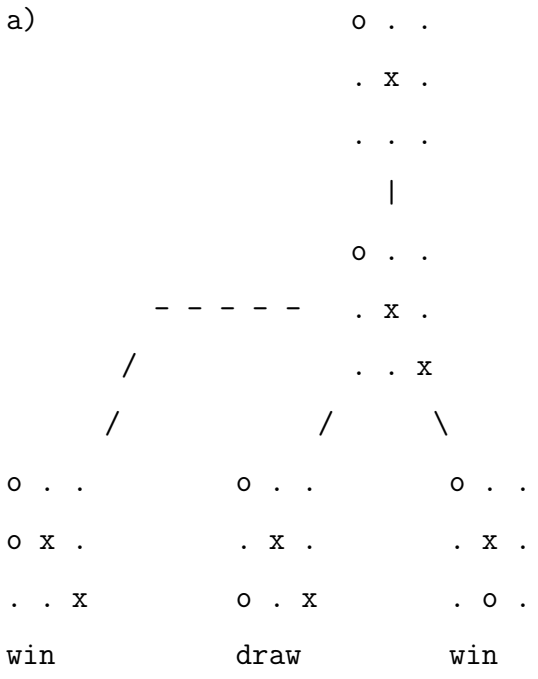
Answer.

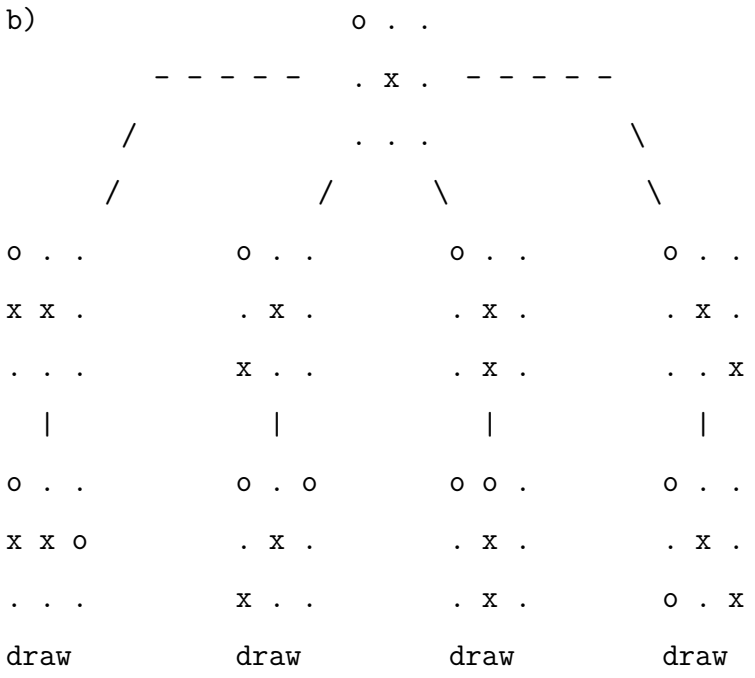
$1 + 5 + 5 + 1 + 3 + 3 + 3 = 21$ nodes. There are 5 children of the root, one for each possible `x`-move. Four of these moves lose right away. The fifth move is where `x` blocked `o`. There is only one move at level 2.

26. a) For this position with `x` to play, draw the next two levels of a proof tree that shows that `x` can at least draw: at level 1 of the tree you can prune isomorphic positions. Below each leaf of your drawing, give the x-minimax value (win/lose/draw) of that subtree.

b) Repeat the question for a proof tree that shows that `o` can at least draw.

Answer. There are many correct answers. Every `x`-move draws: to minimize the time needed to write this anwer, I picked a strategy for `x` that prunes three subtrees. In `o`'s strategy, three of the four moves are forced: in the fourth case, `o`'s move is unique (up to symmetry): the other four moves lose. You can check this using `tt.py`.




b)
o . .
- - - - - . x . - - - - -
/ . . . \
/ / \ \
o . . o . . o . . o . .
x x . . x . . x . . x .
. . . x . . . x x
| | | |
o . . o . o o o . o . .
x x o . x . . x . . x .
. . . x . . . x . . o . x
draw draw draw draw

27. When running `tt.py`, you can call function `see_positions` by typing `#`. Here is some output:

```
3139 psns with x to move            2907 psns with o to move
occupied cells   number psns        occupied cells   number psns
    0               1                   0               -
    1               -                   1               9
    2               72                  2               -
    3               -                   3               252
```

a) Explain why the numbers at levels 0,1,2,... start 1,9,72,...: where have we seen this pattern before, and what does it represent?

b) From a), we might expect that the number at level 3 will be 504, but it is 252: why?

c) In `tt.py`, solving empty board returns `x minimax result 0 nodes 549946`. Explain message.

d) Modify `tt.py`: in function `negamax`, uncomment line below.

```
for cell in L:
    psn.brd[cell] = ptm
    nmx, c = negamax(use_tt, use_iso, MMX, 0, d+1, psn, opponent(ptm))
    so_far, calls = max(so_far, -nmx), calls + c
    psn.brd[cell] = Cell.e
    # if so_far == 1: break
```

Now solving returns `x minimax result 0 nodes 94978`. Explain.

Answer. a) These are the numbers of nodes in the breadth-first-search tree for tic-tac-toe. At level k, they represent the number of positions of x,o,empty cells that have k cells on the board.

b) Up to level 3, you can create all the nodes at each level like this:

level 0: all cells empty, only 1 way to do this.

level 1: one x- cell, all others empty, 9 ways to place the x-

level 2: one x- cell, one y-cell: $9 * 8$ ways to place these (place the x- 9 ways, then place the o- 8 ways)

level 3: two x-cells, one y-cell. you can get each position in this way: place the first x- in (9 ways), then place the o- (8 ways), then place the second x- (7 ways), total $9 * 8 * 7 = 504$. But we have created each position 2 times, because each two x-cells was created with one of them placed first, and then later with the same one placed second.

c) minimax result when x plays first is 0 (a draw). the number of nodes in the recursion tree (so total number of recursive calls) is 549 946. This is around what we expect: if the game always continued until the board was full, there would be about 1 000 000 nodes. But sometimes the game ends before the board is full.

d) Now the loop ends as soon as we find a child node where the next player to move has a win. This prunes any remaining siblings of the winning child. Now we find the minimax value when exploring only 94 978 nodes.

28. In `tt.py`, how is `Isos` used?

```
Isos = ((0,1,2,3,4,5,6,7,8), (0,3,6,1,4,7,2,5,8),
        (2,1,0,5,4,3,8,7,6), (2,5,8,1,4,7,0,3,6),
        (8,7,6,5,4,3,2,1,0), (8,5,2,7,4,1,6,3,0),
        (6,7,8,3,4,5,0,1,2), (6,3,0,7,4,1,8,5,2))
```

Answer. It is used to find the isomorphic representative of a position p. For each of the 8 permutations, the permutation is applied to p and p's base-3 representation (as a decimal number) is computed. The minimum of these numbers is the representative.

---

end of practice questions

extra questions (will not appear on the quiz)

1. Using `stp_search2.py`, find all hardest 3×3 solvable STPs. Explain.

Answer. Observe: Take the unsolvable-positions component, say from the breadth-first-search graph (it's a graph, not necessarily a tree, because there will be transpositions, namely nodes at some depth k that have two or more parents, because there are two or more paths from the root to that node: in a tree, each node has only one parent) starting from any unsolvable position. Now, pick two consecutive-valued tiles (e.g. 1,2 or 2,3 or 3,4 ...) x, y and, in the component, for each position, switch the locations of x and y. Then the resulting component is again a breadth-first-search graph, and it contains all solvable positions. The two components are isomorphic, so if there is a path of some length t, then the corresponding path in the other component (found by switching the locations of x and y on every position in the path) is again a path. Using this idea, we find all longest paths in the STP search space that lead to the target position by first looking at the unsolvable component.

So start with any unsolvable STP with a blank in the same position as the target position (last position in bottom row): we want it there because we are going to map this position to the solvable target location, so the blank must be in the correct location. Run the algorithm. Find the set S of positions at the deepest level of the breadth-first-search. Relabel the start position so that it becomes the target position. Use this relabelling for all positions in S. Each relabelled position will be a hardest (shortest solution is maximum) solvable STP.

An easy way to do this is to use 213456780 as the unsolvable STP (because the relabelling needed to make this solvable is simple: swap 1 and 2). On the next page is relevant output.

```
 181438 iterations, level 31 has 2 nodes
 6 4 7
 8 5
 3 1 2


 8 6 7
 1 5 4
 3   2
181440 iterations, level 32 has 0 nodes
no solution found
```

So there will be two positions in S. Relabelling these positions (swap 1 and 2) gives the following positions, each with a shortest solution of 31 moves:

```
 6 4 7     8 6 7
 8 5       2 5 4
 3 2 1     3   1
```

We check our answer using `stp_search2.py`. It confirms that each of these is solvable, each takes 31 moves.