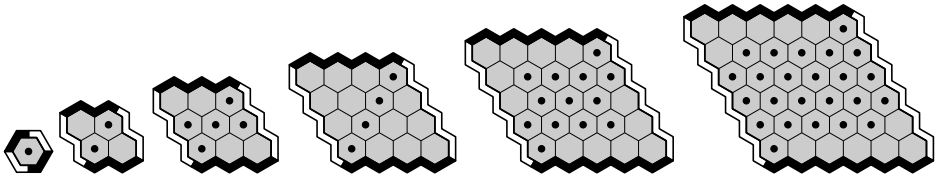
cmput 355 2025 practice questions 5 (with answers)

Explain each answer, show all work.

1. For $n \times n$ hex for each $n \leq 4$, show all winning 1st black moves. Repeat for white.

Answer.

Here are answers for black. Use symmetry to find answers for white.



2. Using negamax solver `hex/hex_simple.py` on your favorite machine, for `x`-to-play and then for `o`-to-play, solve 3×3, 3×4 (3 rows, 4 columns), 4×4 hex: you will need to change ROWS and/or COLS. **If this takes more than 1 minute, just kill the process and ignore that part of the question.** In each case, give

   a) who wins
   b) a winning move if 1st-player wins
   c) the number of calls made in solving
   d) the runtime, if it is given.

Answer. Data below from mac 2021 desktop. Notice 1st player wins on 3×3, but that black (who has the shorter distance to join her two sides) wins in both cases for 3×4, confirming what we know from hex theory.

```
black-to-play, mac 2021 desktop
size   winner win-move calls    time
3x3      b      c1     5677      0.0
3x4      b      a1   161899      0.5
4x4      b       ?        ?   > 60.0

white-to-play, mac 2021 desktop
3x3      w      c1     2677      0.0
3x4      b      --  1183377      3.2
4x4      w       ?        ?   > 60.0
```

3×3 black-to-play, winning move *c1*: why? Default move ordering is row by row, left to right within each row, and *c1* is the 1st move in this order that wins for black.

3×4 black-to-play, winning move *a1*: why? Same as previous answer: here black has a winning 2nd-player strategy and in hex extra stones on the board are never a disadvantage, so when black plays 1st every 1st move wins.

3×4 white-to-play, black wins, so no winning move given.

3. In `hex_simple.py`, uncomment the lines that assign a better move order to CELLS and repeat question 2.

Answer.

```
3x3      b      b2      678      0.0
3x4      b      b2    21668      0.1
4x4      b      c2  7034997     26.3
```

4. For the 4×3 hex board, give a CELLS move order that reduces the number of calls made by `hex_simple.py` both for solving `x` and for solving `o`.

Answer.

```
4x3      w      --   456373      1.3  default order
4x3      w      --    40294      0.2  (4,7,5,6,3,2,8,9,1,10,0,11)
```

We want `o` to find a winning strategy quickly. Put 4,7 at the front of the order: then `o` always gets at least one of these and `x` is in trouble. Let me know if you can do better `:)` .

5. Repeat question 2 for Monte Carlo Tree Search player `mcts/main.py`. Also:

e) Why do different executions sometimes give different moves?

f) How can this player solve 4×4 hex faster than minimax?

Answer.

This mcts does not handle 3×4, so answers for 3×3 and 4×4 only.

a) 1st player (as expected)

b) I ran this several times. For 3×3 and 4×4, in each case the winning move was on the short diagonal.

c) 350-500, 40328 (c2 or b3), (78089 when d1 or a4).

d) 0s, about 3.6s (c2 or b3) or about 7s (d1 or a4).

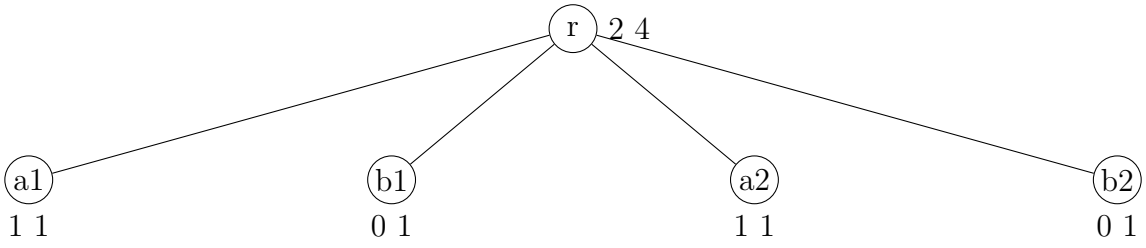
e) Each rollout is a sequence of pseudo-randomly selected moves.

f) The version of mcts we are running is `mcts1.py`, which identifies proven-win positions, either because the stones on the board join a player's sides, or recursively when a node is found with all children proven-loss, in which case the node becomes proven-win and in back-propagation it's parent becomes proven-loss. This yields two kinds of implicit pruning:

• When we learn in `expand-node` that a node is proven-win, we do not create any further siblings.

• Proven-loss nodes are never picked by `best_uct` because their uct sum is always $-\infty$, so once a node is designated proven-loss its subtree is never searched or expanded again.

6. Below is the Monte Carlo Tree Search (MCTS) tree after the simulations (iterations) 1-4 from call *mcts x* in `mcts/main.py`, and also the output from simulation 5. Show the tree after simulation 5: explain each change.




```
trv_xpnd bu * 1.4 .4 1.4 .4 5
    xpnd_nd * 5 > 6
    xpnd_nd * 5 > 9
    xpnd_nd * 5 > 10
  sim 5. * 5 10 roll 6 9 parent win
```

Answer.

See `https://webdocs.cs.ualberta.ca/~hayward/355/mcts25.pdf` .

The call `traverse_and_expand` expands all possible children of tree node `* 5` (the leftmost child of the root), creating nodes `* 5 6`, `* 5 9`, `* 5 10`, labelled *b1, a2, b2* in the tree. Then the rightmost of these children is picked as the leaf at which to simulate. The rollout yields a white win (cells 9, 10), so a win for the parent of node `* 5 10`. So the new node is labelled 1 1 (each node is labelled with its parent scores), and back propagation changes its parent from 1 1 to 1 2, and the root (where the win sums of its children are stored) from 2 4 to 2 5.

7. Repeat the previous question for each simulation from 5 to 13: after simulation s, if you are given the mcts tree at that point and the output from simulation $s + 1$ show the updated tree: explain each change.

Answer.

```
trv_xpnd bu * .5 .5 .7 .5 9 bu 9 1.5 1.5 .5 5
xpnd_nd * 9 5 > 6
sim 11. * 9 5 6 win, no more sibs
```

Simulations up to 10 are routine. In sim. 11 we see the comment *win, no more sibs* for the first time: why?

`best_uct` from root selected 9 (cell 9 = a2). `best_uct` from 9 selected 5 (cell 5 = a1), a leaf, so execution passed to `traverse_and_expand`, possible children cells 6 and 10. Child 6 is expanded and sim. 11 performed. The current position has black stones at cells 9 and 6 and a white stone at cell 5. This is a win on the board (true win, not a simulation win) so we execute the `if won` code in `expand_node`, printing message `win, no more sibs` and backpropagating value ∞ for black at node `* 9 5 6`.

Simulation 12 is similar to sim. 11.

```
trv_xpnd bu * .5 .5 .9 .5 9 bu 9 -inf -inf .5 10
xpnd_nd * 9 10 > 5
sim 13. * 9 10 5 win, no more sibs
```

Simulation 13 is simular to sim. 12. Backpropagation from node `* 9 5 10*`: that node is a proven black win (∞), so parent node is proven loss ($-\infty$), so grandparent node is set to $-\min\{-\infty, -\infty, -\infty\} = \infty$: this node is a proven win because all of its children are proven losses. Finallly, root node score is set to max of all child scores, so ∞.

8. a) Which of these features does `hex_simple.py` use? For each unused feature: would it significantly improve runtime?

   prune isomorphic positions transposition table alphabeta search

   b) In `hex_simple.py`, what algorithm does `has_win()` use to determine whether a player has won the game?

Answer. a) none. there are generally fewer symmetries in hex than ttt: the most common is rotation by 180 degrees, you are unlikely to reduce the search space by more than .5. There are lots of transpositions in hex, the dag of all continuations (DOAC) would probably have less than .5 the nodes of the TOAC. There are only two outcome values in hex (win/loss), compared to three in ttt (win/loss/draw), so alphabeta won't save as much as it does with ttt, especially because `hex_simple` is faster than pure minimax, as it returns as soon as a winning child is found.

9. In `mcts1.py`, in `def best_uct(...`, change line `uct = mean_res + ...` to `uct = mean_res`. a) Explain how this changes the mcts algorithm. b) In the previous question, instead of output `trv_xpnd bu * 1.4 .4 1.4 .4 5`, what would the output be? c) With `MCTS_TIME = 1`, how does this change modify algorithm behaviour on 2×2 and 3×3 boards? On the 4×4 board?

Answer.

a) There is no additive uct factor, so this is a purely best-1st algorithm.

b) `trv_xpnd bu * 1.0 .0 1.0 .0 5`. Notice that the same node is picked here, because the algorithm picks the 1st max-value child.

c) There is little observed change on the 2×2 and 3×3 boards: this is `mcts1.py`, which backs up wins and losses, so in 1 2nd the algorithm can usually solve any position on these boards. On the 4×4 boards the search tree is big enough that it can, after a few unlucky rollouts, end up picking a bad move, such as shown below. On larger boards the performance degrades even more.

```
move  sims  wins
   7     1     0
   8     1     0
   9     1     0
  10     1     0
  13     1     0
  14     1     0
  15     3     1
  16     1     0
  19     3     1
```

```
   20     1     0
   21 14359  8846
   22     3     1
   25     3     1
   26   213    91
   27     1     0
   28   511   219
total 15104 -5944

    a b c d
  1  . . . . o
   2  . . . . o
    3  . . x . o
     4  . . . . o
         x x x x
```

10. `https://webdocs.cs.ualberta.ca/~hayward/355/mcts25.pdf` shows output from an execution of `mcts/main.py` (`mcts x`) for the empty 2×2 hex board.

    a) After search ended, what criterion was used to pick the move to play?

    b) Is the move from a) provably winning, probably winning, provably losing, probably losing, or none of these?

    c) Repeat the previous two questions if search had ended after simulation 10.

    d) Find a simulation in which the leaf selection was made by picking a most-likely-winning child at each step, or explain why there is none.

    e) Find a simulation in which the leaf selection was made by not always picking a most-likely-winning child, or explain why there is none.

Answer.

a) `mcts1.py` returns a proven win if one is found, in which case the output shows the number of wins as ∞. if there is no proven win then, among all nodes that are not proven losses (or among all nodes if they are all proven losses), `get_best_move` returns the first node found with the most simulations. here move 9 (cell a2) is discovered as a proven win: it is returned

b) provably winning

c) there is no probably winning node. moves a1 (found first) and a2 both have 4 simulations, so move a1 is returned. sad, because a1 is provably losing.

d) sim'ns 5 to 13

e) in each case the uct boost for inferior moves was not enough to exceed the score for a node with best mean score (wins/sims).

11. a) Run `python3 main.py` in `mcts` on the 4×4 board. What move does it make? Is this a winning move? b) In `mcts/mcts1.py`, replace this line

    ```
    uct = mean_res+(self.c*sqrt(log(self.root_node.sims)/child.sims))
    ```

    with line `uct = mean_res` and repeat a). How do your results compare with a)?

Answer. a) the program is non-deterministic, so the winning move varies. I ran the program 20 times with 1s runtime, it found a winning move 19 times. Each run performed about 11000 simulations.

b) the number of simulations increased from around 11000 to around 16000, which is to be expected since the floating point function calls to square root and logarithm take significant time. The search changes from an exploitation-exploration algorithm to an exploitation-only algorithm, so we might expect it to make a bad move here: I ran it 20 times on the 4×4 board, it found a winning move 15 times.

12. Here is one way that the mcts algorithm in `mcts/` could be improved: at a node, whenever a true win (i.e. minimax win, so in hex once a player has joined their two sides) is detected, prune their siblings.

    a) For any two player game, what else could be done at this point?

    b) For hex, what else could be done at this point?

Answer. Assume that a node p in the game tree has a child v, that v has a child c, and that c is a win. Then v is a loss, so we can use this info in the rest of our search. For example, we might check whether v has any siblings, and if so, just prune v.

b) In hex (also tic-tac-toe) once we know that c wins we can prune all children of p except moving to cell c, because if we don't then the opponent can just move to c after we move to not-c.

13. For each of the winning 3×3 hex 1st-player strategies below:

    a) is this a 1st-player or 2nd-player strategy? and for black or for white or for both?

    b) explain the strategy in words

    c) Draw this strategy as an and-or tree.

    $b2 \wedge (b1 \vee c1) \wedge (a3 \vee b3)$

    $a3 \wedge (a2 \wedge (a1 \vee b1) \ \vee \ c1 \wedge (b2 \vee c2 \wedge (b3 \vee c3)))$

    $a2 \wedge (a1 \vee b1) \wedge (a3 \ \vee \ c2 \wedge (b2 \vee c1) \wedge (b3 \vee c3))$

    d) Explain why this strategy notation is called opponent-oblivious.

Answer.

We answer for 2nd connection: we leave the other two to you.

a) this strategy works for black (top-to-bottom). it supplies an initial move, so it is for the 1st player.

b) black plays at $a3$, leaving a black top-to-bottom safe connection. this vc is formed by two weak-connections (1st-player) S_1 and S_2.
S_1 is this: black plays at a2 and then plays one of $\{a1, b1\}$ (white cannot take both).
S_2 is this: black plays at c1 and then plays either the one-cell sc $S_3 = \{b2\}$ or the three-cell sc S_4.
S_4 is this: black plays at c2 and then plays one of $\{b3, c3\}$.

c) and d) See `https://webdocs.cs.ualberta.ca/~hayward/355/sspace.pdf`

14. See slide 32 in

    `https://webdocs.cs.ualberta.ca/~hayward/talks/hex.someques.pdf`

    a) Explain why the black stone is safely connected to the top. Write this safe connection in logical form using the notation from the previous question.

    b) Explain why the black stone is safely connected to the bottom. Write this safe connection in logical form using the notation from the previous question.

    c) Explain why the top is safely connected to the bottom. Write this safe connection in logical form using the notation from the previous question.

Answer.

a) the cell set is $\{c1, d1\}$: if white plays at one of these, play can play at the other to join the black stone at $c2$ to the top.
b) the cell set is $\{d2, b3, c3, d3, a4, b4, c4, d4\}$. this is the 4-3-2 side template: if white plays in $\{b3, a4, b4\}$ black plays at $d3$; if white plays in $\{d2, c3, d3, c4, d4\}$ or outside the cell set black plays at $b3$.
c) combine the two safe connections. logic form is left for you.

15. See slide 45 in

    `https://webdocs.cs.ualberta.ca/~hayward/talks/hex.someques.pdf`

    a) This position is white-to-play. If white's move is not in one of the 9 dark cells, explain how black can win.

    b) If white's move is not at the 1 dark cell in slide 48, explain how black can win.

    c) For a given hex position and given player-to-move, explain what a *mustplay region* is.

Answer.

a) the diagram shows the cells of a black win-threat: after black plays at the dot (e4), play has a top-to-bottom vc using the remaining 8 grey cells.

b) slides 45, 46, 47 each show a black win-threat. the intersection of the cell sets of these 3 win-threats is the one cell shown on slide 48 (d5). If white plays at any other cell cx, then cx does not intersect all 3 win-threats, so it misses at least one win-threat, so after white plays at cx, black can play at any win-threat that cx did not iterfere with, turning the win-threat into a winning vc.

c) given a set of opponent-win-threats (opponent weak connections), a player's *mustplay region* is the intersection of the cell sets of the opp't-win-threats

16. Go to `https://webdocs.cs.ualberta.ca/~hayward/355/asn/hexviz/`. Click *back*, click *4 by 4*, click *options*, click *rule decomp*, click *brain*, click *done*. You will see the 4×4 hex board with 1 blue cell, 1 brown cell, 3 pink cells, 5 green cells. What does the diagram show? What do the colored cell sets represent? Use the terms *safe connection* and *weak connection* in your answer.

Answer. Diagram shows a winning (so joining top-left and bottom-right sides) safe connection for black. Cells represent subconnections: black stone has safe connection to top-left (union of blue and brown weak connections) and safe connection to btm-right (union of pink and green weak connections).