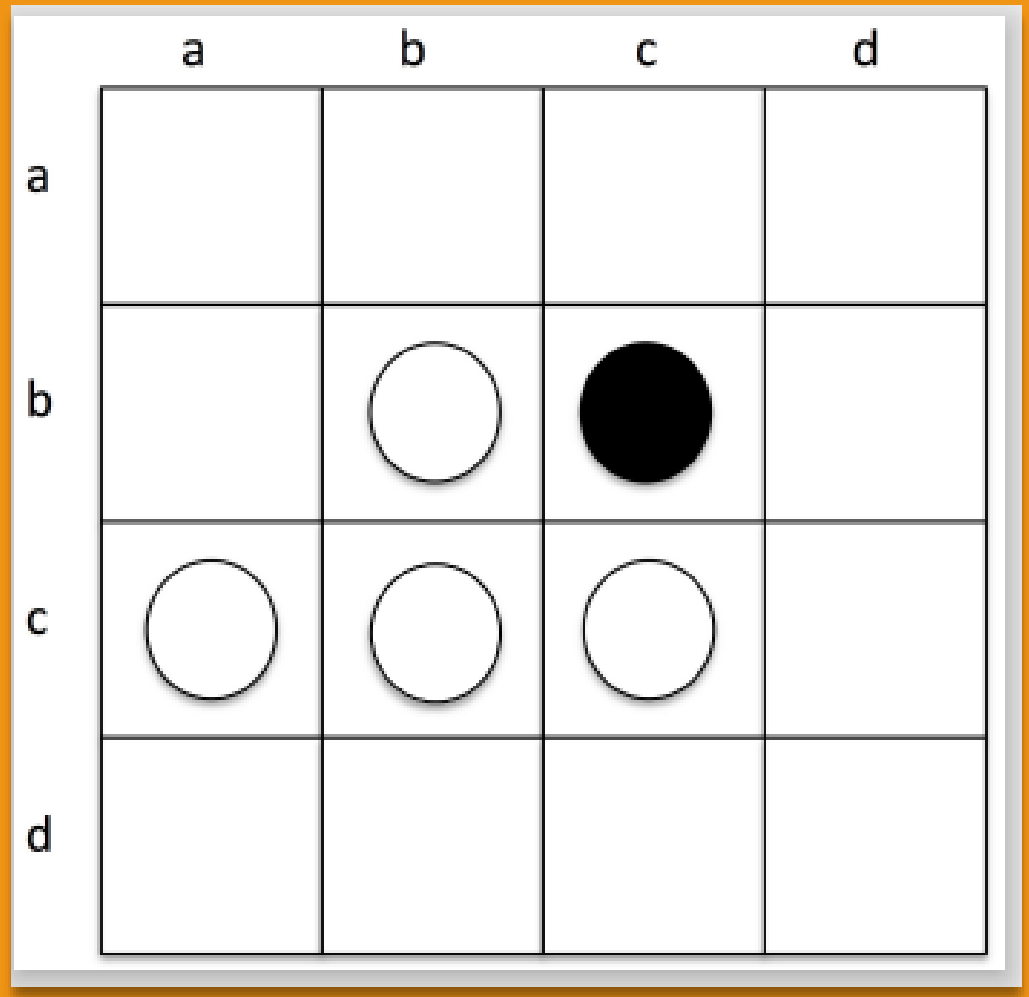
Lab 8 and (again) Planning larger programs

Phil Anderson P.Eng, PhD
Associate Professor Teaching Stream Emeritus



Lab 7 gave basics inside game, Lab 8 implements the control of the game

Lab 8:

1. play user vs program
2. play program vs online

REMINDER

PLAN!!!

- Don’t just code → *think first*.

Almost all programmers find it difficult at first, getting into the rhythm of programming

REMINDER

DO NOT JUST START TO CODE <<<< BIG BIG MISTAKE.

LEADS TO AWFUL CODE AND HUGE PROBLEMS IN DEBUG AND UNDERSTANDING

REMINDER

Start by breaking the problem into parts or modules.

- Look at the flow between modules.
- Look for commonality between modules.

Don’t worry if a module seems much too complicated –

>>>> you repeat this process on each module by itself

>>>> since it’s by itself, you don’t have to worry about the other modules!!

REMINDER

Start by breaking the problem into parts or modules.

- Look at the flow between modules.
- Look for commonality between modules.

Don’t worry if a module seems much too complicated –

>>>> you repeat this process on each module by itself

>>>> since it’s by itself, you don’t have to worry about the other modules!!

IMPORTANT: Every module you create should be tested by itself.
Do NOT code large sections and then test/debug.
(Sounds like overkill but it will save significant time)

NEW

REMINDER

Write pseudocode (descriptive phrases sort of like code) that do what you want

Be aware that every part will need an introduction, processing and possibly a windup

REMINDER

What I often do:

- write the pseudocode as comments
- add subroutine/function calls to implement the pseudocode, then comment out the call
- bring them in one at a time for testing as I write the subroutine/function. Add/remove “support” code as needed for testing

What I often do:
- draft a “user’s manual” to let me know how the user will expect to use the software

NEW

Lab 7 does some of the background work:

1. load a board
2. print out the board (uses a text format)
3. show legal moves for white
4. show legal moves for black
5. put a move on the board and flip the pieces for the move appropriately

For lab 8 we need to add:

1. control of the action (who moves and how)
2. for part 2: a strategy to play well

Notes on testing a module

1. Comment out other code that is not needed. Remember /* ... */ to comment out large sections.
2. Breakpoints to check intermediate values
3. Don’t be afraid to add code for
    i. Input/output of values
    ii. Printing of results and intermediate results
    iii. Allowing breakpoints when certain conditions are detected/determined.
4. Check boundary conditions, bad inputs, special cases as makes sense (Don’t assume there is success if you don’t see smoke)

Strategy: Convert problem to pseudocode

Startup
Initialize board
Determine who plays first

Body
while(play is possible)
    if(user's turn)
        until valid input get user move
    else //computer's turn
        look at moves for best valid input
    print board
    change to other user

Windup
indicate winner
exit

Convert subproblem to pseudocode

play is possible (true/false)

```
if(there is a valid move for white OR
     there is a valid move for black
then    play is possible
else    play is not possible
```

Strategy: Go partway

Entire problem is quite complicated with a lot of things to track and take care of.

Is there a problem that is simpler, that we can then revise and augment to make into the given problem?

Convert simplified problem to pseudocode

Startup
Initialize board
~~Determine who plays first~~
user first and user is white

Body
loop forever
~~while(play is possible)~~
    if(user's turn)
        until valid input get user move
    else //computer's turn
        look at moves for ~~best~~ any valid input
    print board
    change to other user

Windup
indicate winner
exit

Get this working first, then save a copy then modify to solve the more complex problem

Expand and repeat – is move available? >> break this function into parts

This will need a function such as

isValidMove(colour, row, column)

returning a true iff that colour can be placed on that spot on the board.

Sounds similar to something we've already done for lab 7!!

Expand and repeat – is move available? >> break this function into parts

Once we have *isValidMove(...)* we can use this to write *moveAvailable(...)* that takes as input a colour and determines if ANY move is available ANYWHERE for that colour.

moveAvailable(...) will have to go through all combinations of rows and columns to see if that move is possible for the given colour.

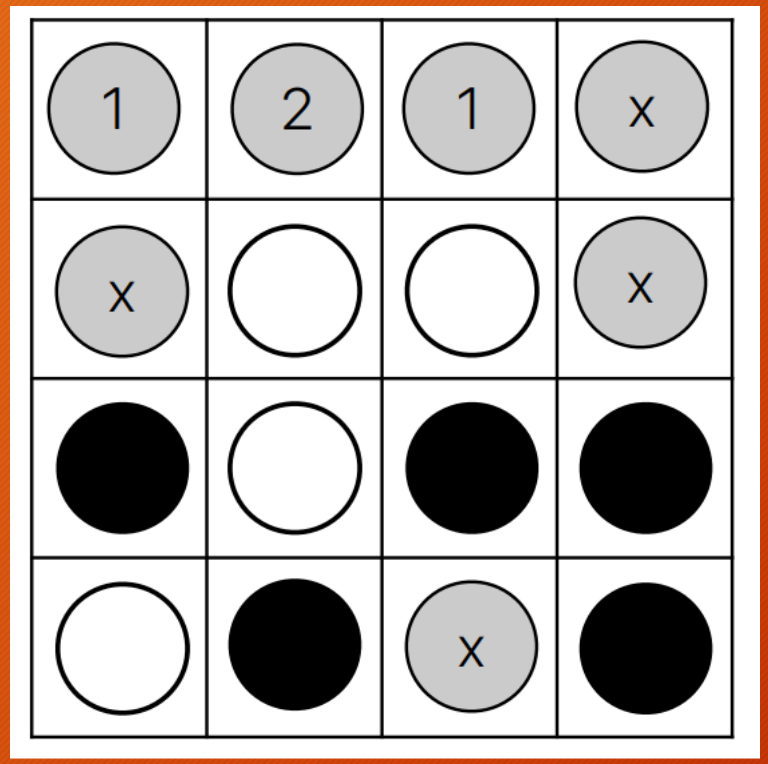
Clearly, using *isValidMove(...)* is the way to check each combination!

>>>>Reminder<<<<

Now test/test/test/test. Make sure this works before moving on!!!

Once we get the basic piece working, we can move to reintroduction of the complex bits. Let's start with finding the next move that turns over the most opposing pieces

Black to move

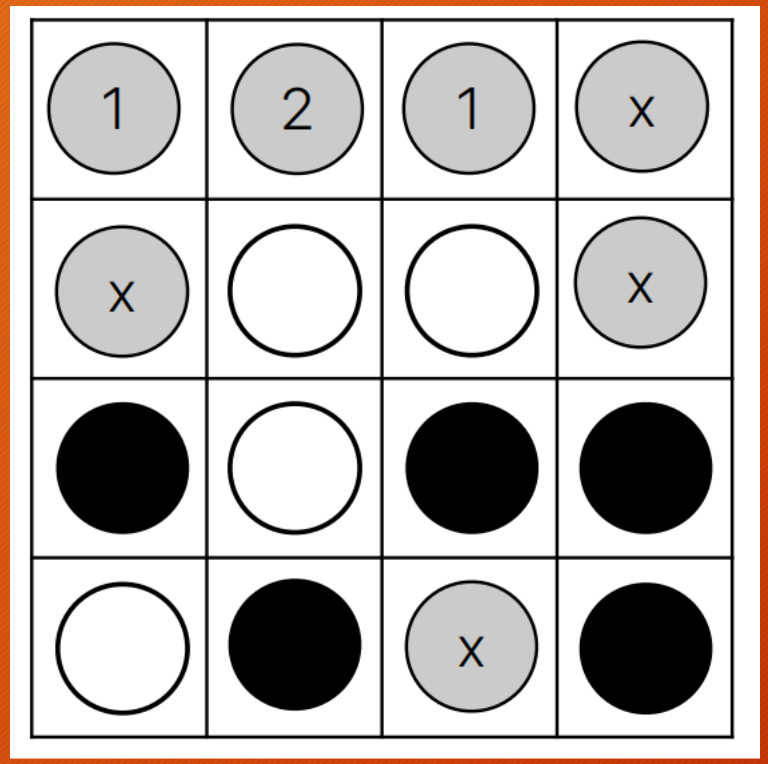


What are the steps to do this?

- loop over row
  - within this loop, loop over cols
  - for every row/col see
    - if legal move and, if so,
      - what its score is (how many opposite pieces are flipped)

Once we get the basic piece working, we can move to reintroduction of the complex bits. Let's start with finding the next move that turns over the most opposing pieces

Black to move

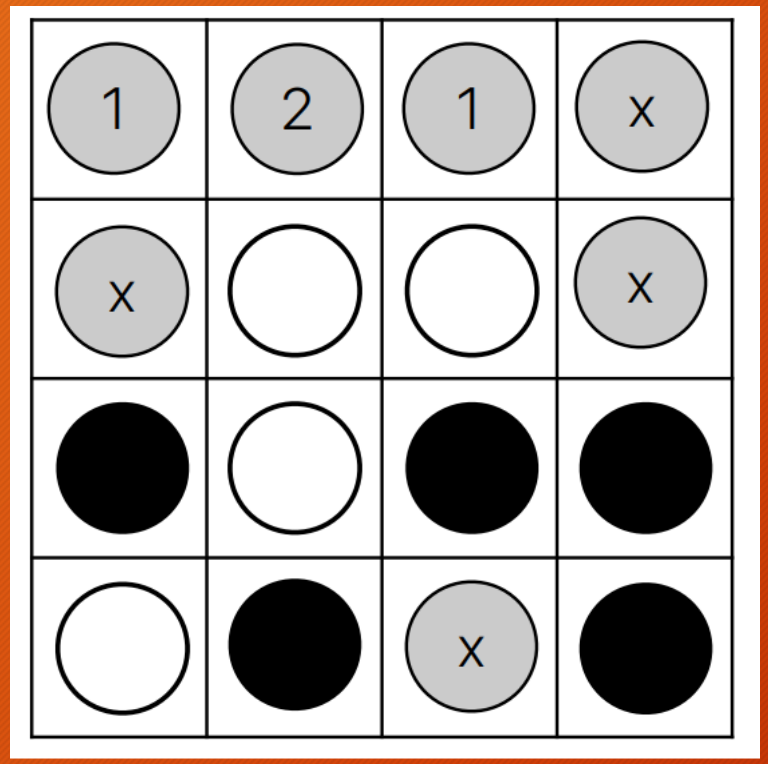


What are the steps to do this?

- keep track of the row/col of the highest score
- update these as move through the board

Once we get the basic piece working, we can move to reintroduction of the complex bits. Let's start with finding the next move that turns over the most opposing pieces

Black to move

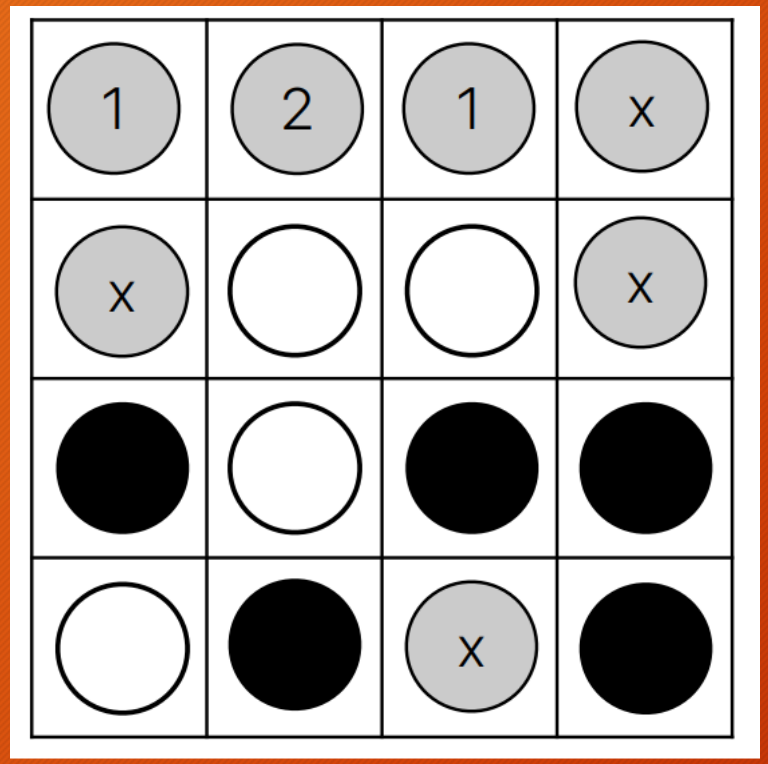


New problem: How to find score of a legal move

- Note lab 7 did flipping
- update the lab 7 routine to count instead of flipping!
- Thought: if returns count==0 then it was not a valid move!

Once we get the basic piece working, we can move to reintroduction of the complex bits. Let's start with finding the next move that turns over the most opposing pieces

Black to move

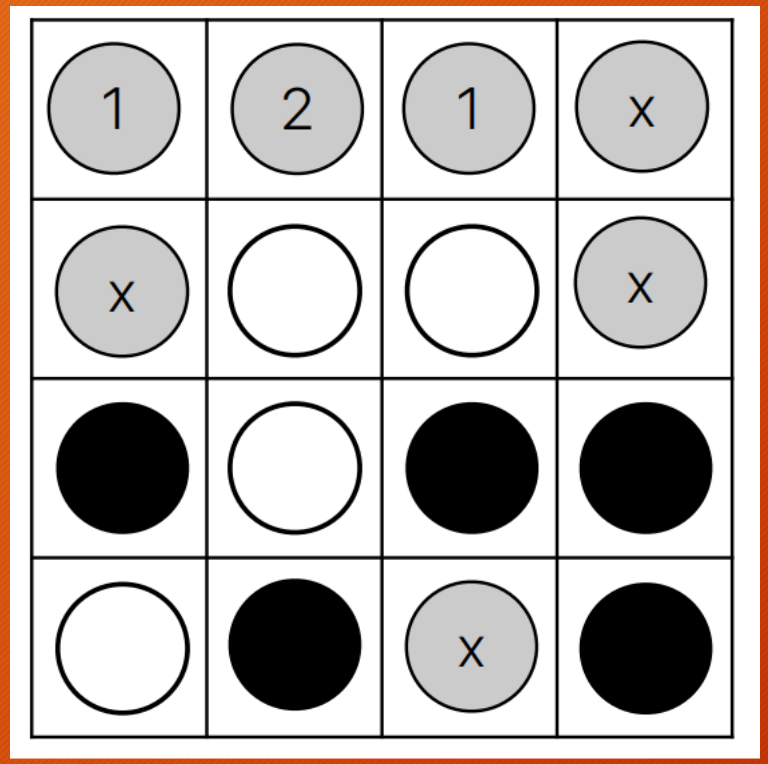


New problem: How to find score of a legal move

- Alternate solution: take a copy of the board, count the pieces of the colour, do the flipping, then count the pieces again to find the difference. Copy back the board (or use a copy to flip)

Once we get the basic piece working, we can move to reintroduction of the complex bits. Let's start with finding the next move that turns over the most opposing pieces

Black to move

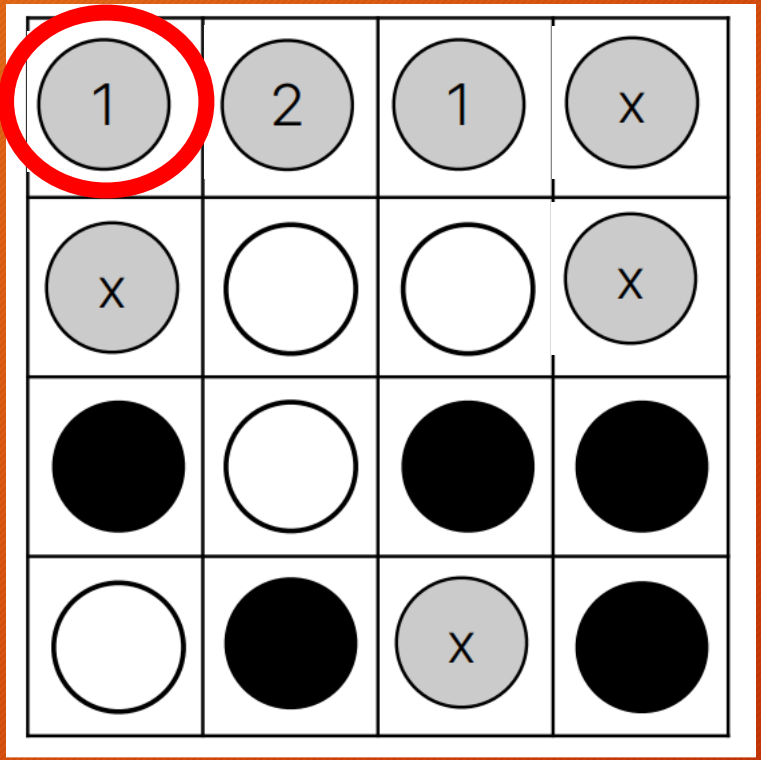


"Finding the biggest" is called "Greedy" (here, the biggest number of flipped opponent pieces).

Part II of lab 8 – game strategies!

For part II a simple "Greedy" strategy may not be best and even then may produce superior choices with the same score

Black to move



For instance:

- corners are great (can't be changed by opponent)
- sides are good (limited opponent possibilities)
- maybe play to anticipate opponent better...
  - how many moves then available?
  - how many good moves for me after these??

You might consider giving a move a “weight” and choosing the move with the most “weight”

weight = a * #flips +
b * #corners +
c * #opponentMovesEliminated

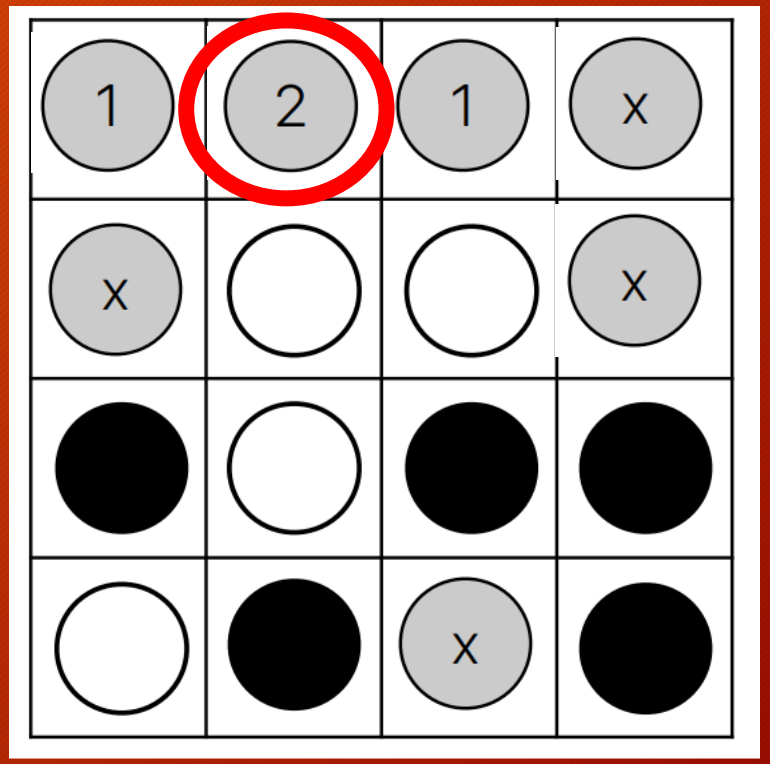
(a, b and c are constants you set empirically)

Clearly, more complex strategies need more complex calculations to implement

For weighting as given, we need to find the number of moves possible by the opponent for each of our possible moves!

Note that the non-corner move leaves the same, single move for white!! Better than the corner??

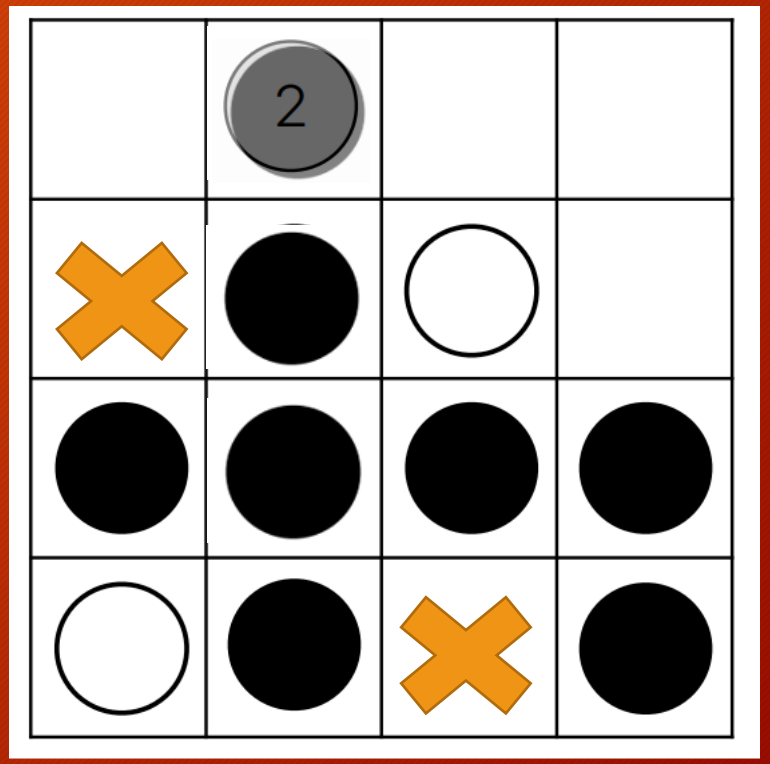
Black to move



How do we find out how many moves an opponent can make for any move we make??

- make a copy of the board
- on the copy, make the move you are considering
- change to opponent's colour and count the number of moves

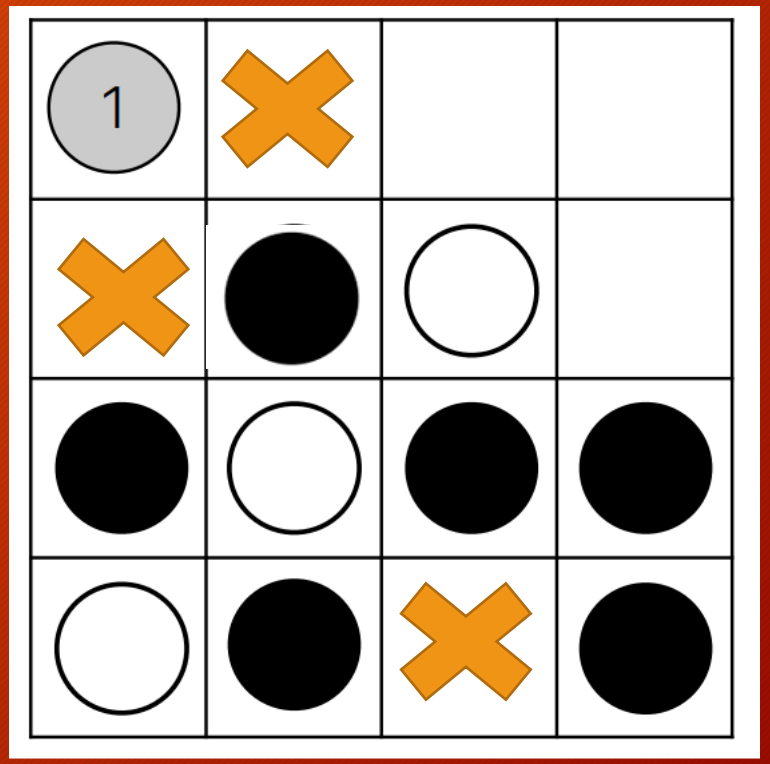
Black to move



How do we find out how many moves an opponent can make for any move we make??

- make a copy of the board
- on the copy, make the move you are considering
- change to opponent's colour and count the number of moves
- repeat for other moves

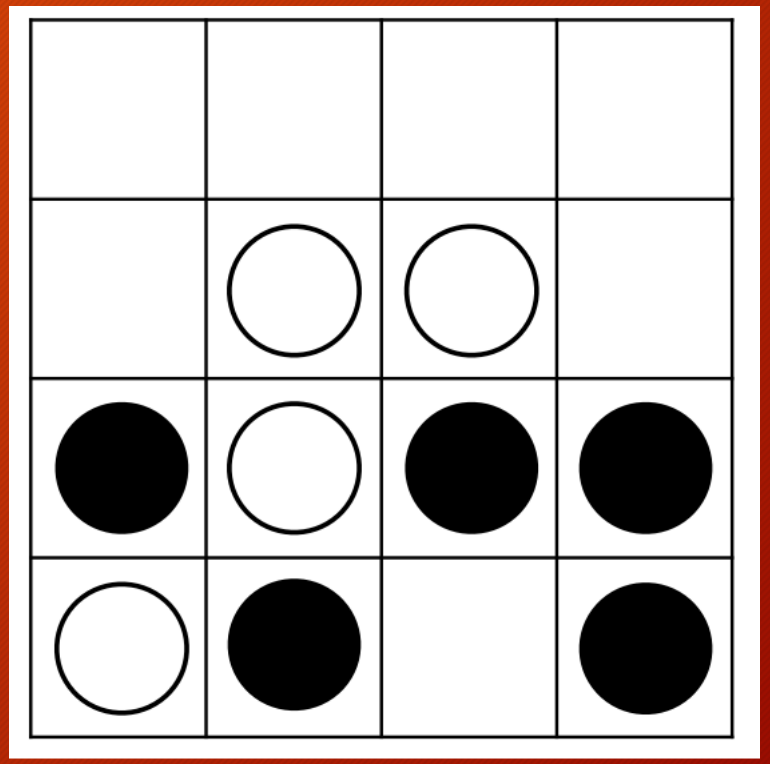
Black to move



A more sophisticated approach uses a "game tree"

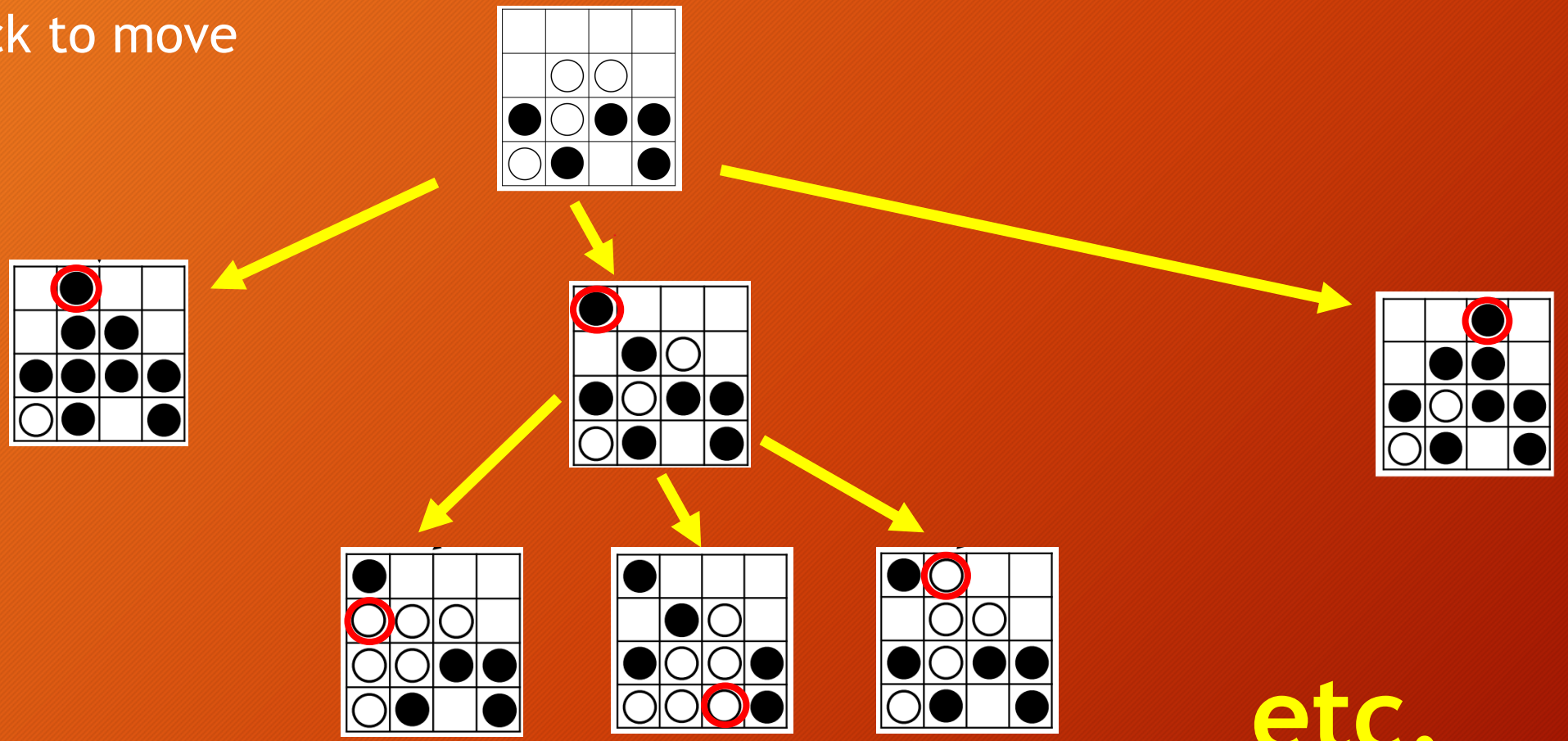
- make a copy of the board
- on the copy, make the move you are considering
- change to opponent's colour and make one possible move
- continue moving alternate pieces for a set number of moves then evaluate board
- repeat for other moves at each step

Black to move



A more sophisticated approach uses a "game tree"

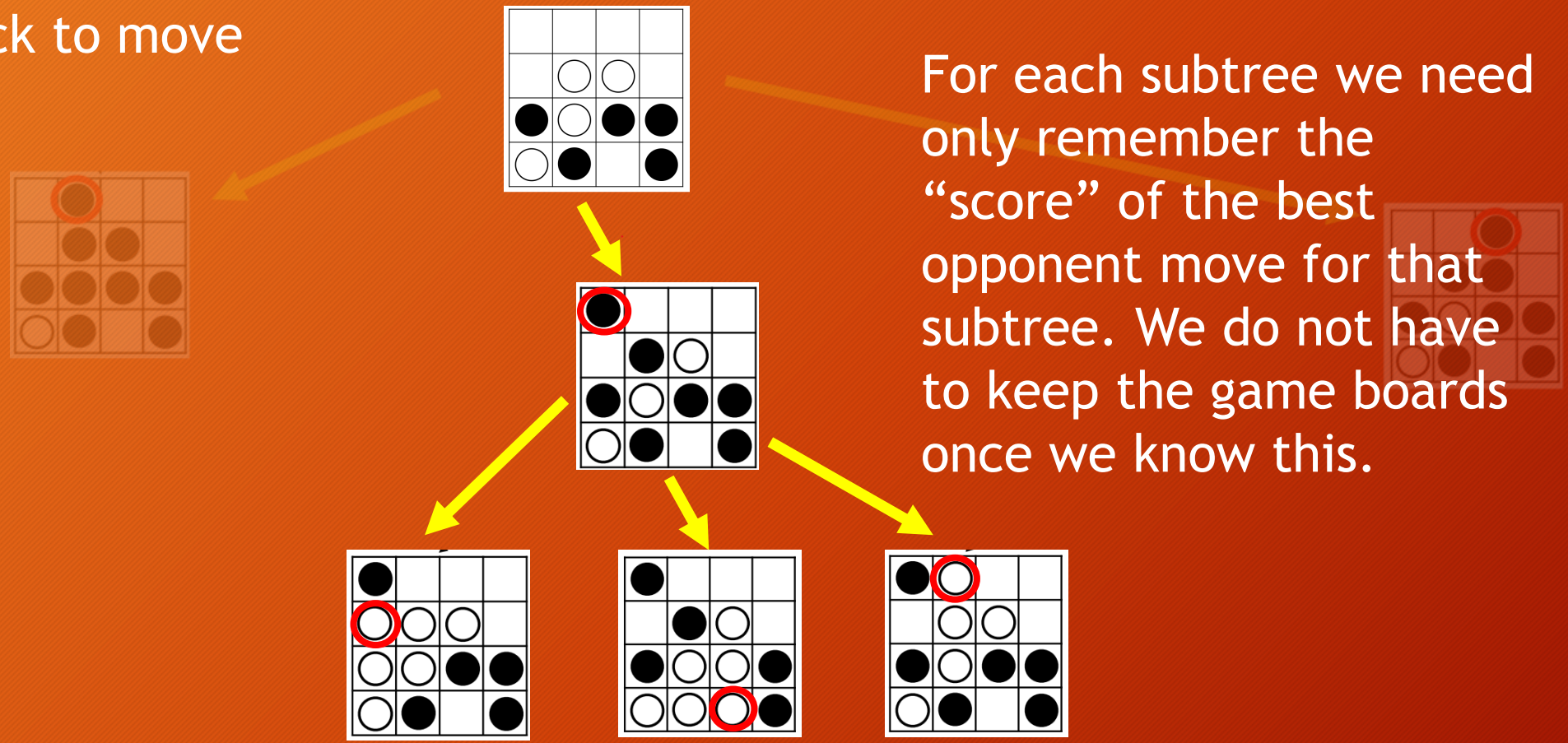
Black to move



etc.

A more sophisticated approach uses a "game tree"

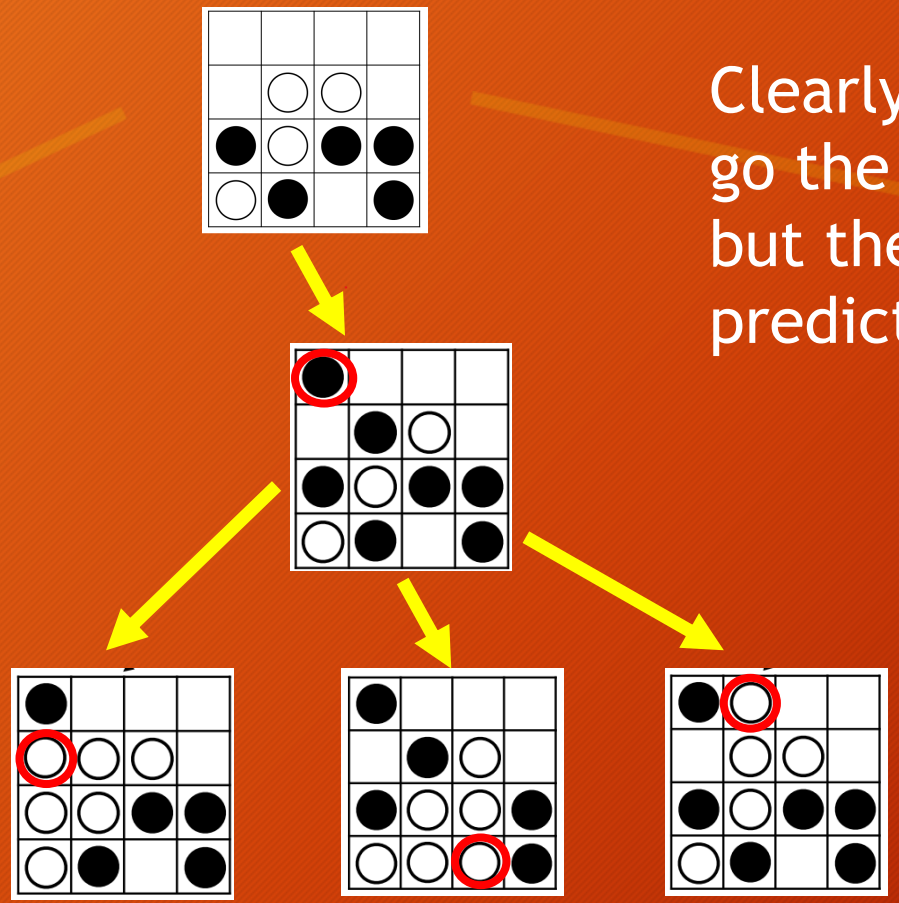
Black to move



For each subtree we need only remember the "score" of the best opponent move for that subtree. We do not have to keep the game boards once we know this.

A more sophisticated approach uses a "game tree"

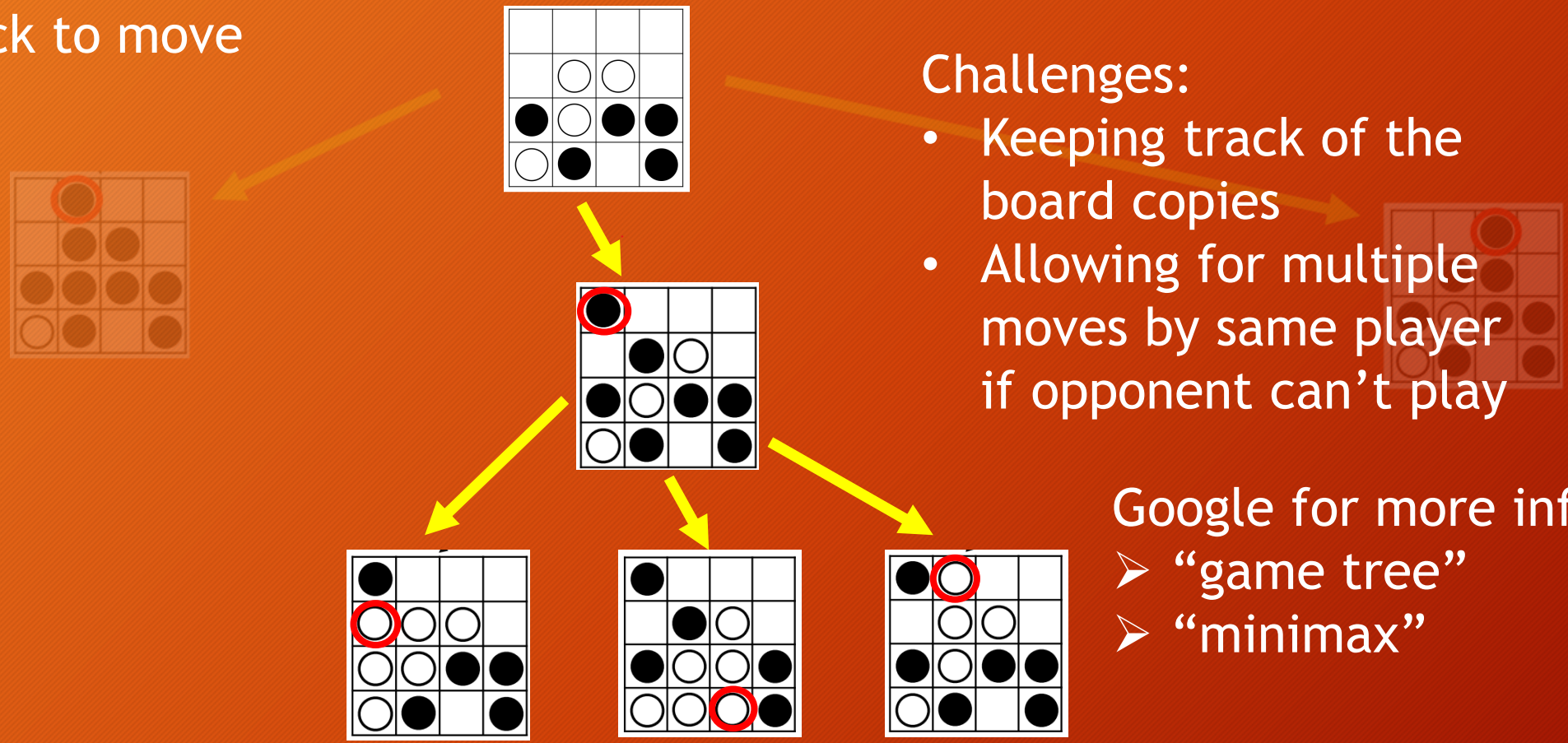
Black to move



Clearly, the "deeper" we go the more work we do, but the better the predictions and scores.

A more sophisticated approach uses a "game tree"

Black to move



Challenges:

- Keeping track of the board copies
- Allowing for multiple moves by same player if opponent can't play

Google for more info:

- "game tree"
- "minimax"

For part II, all responses that pass the test cases will be entered in a competition

- Entered automatically when you submit your Lab 8 Part 2 to examify.ca, if your submission passes the test cases
- Pairwise competitions between leaderboard participants
- Two games are played between each pair of finalists, and the results are scored and ranked
- Continuously run every several days, submit as many times as you wish

The lab is a major programming challenge!
Enjoy!!

As has been said….

So long and thanks for all the fish.

– Douglas Adams, The fourth book in the HitchHiker's Guide to the Galaxy trilogy