Reversi: Implementation and Strategies

Baochun Li
Department of Electrical and Computer Engineering
University of Toronto

Revisiting engineering design

Revisiting engineering design

- **Divide-and-conquer**: involves dividing a complex problem into smaller pieces

Revisiting engineering design

- **Divide-and-conquer**: involves dividing a complex problem into smaller pieces
    - **Implement** and **test** each of the pieces

Revisiting engineering design

- **Divide-and-conquer**: involves dividing a complex problem into smaller pieces
  - **Implement** and **test** each of the pieces
  - **Compose** solutions for the smaller pieces into a solution for the overall complex problem

Revisiting engineering design

- **Divide-and-conquer**: involves dividing a complex problem into smaller pieces
  - **Implement** and **test** each of the pieces
  - **Compose** solutions for the smaller pieces into a solution for the overall complex problem
- May have many “levels” of dividing, implementing and testing (especially for a very complex problem)

Revisiting engineering design

- **Divide-and-conquer**: involves dividing a complex problem into smaller pieces
  - **Implement** and **test** each of the pieces
  - **Compose** solutions for the smaller pieces into a solution for the overall complex problem
- May have many "levels" of dividing, implementing and testing (especially for a very complex problem)
- Many times, decomposing a complex problem into manageable pieces is challenging in itself and often requires creativity!

Imagine you are playing Reversi with a friend using a board, what are the actions of play?

Game action: Pseudocode

Game action: Pseudocode

```
turn = Black // initially
```

Game action: Pseudocode

```
turn = Black // initially
computer = White // for example
```

Game action: Pseudocode

```
turn = Black // initially
computer = White // for example
while (game is not over) {
```

Game action: Pseudocode

```
turn = Black // initially
computer = White // for example
while (game is not over) {
  if (turn == computer)
```

Game action: Pseudocode

```
turn = Black // initially
computer = White // for example
while (game is not over) {
  if (turn == computer)
    computer should make a move
```

Game action: Pseudocode

```
turn = Black // initially
computer = White // for example
while (game is not over) {
  if (turn == computer)
    computer should make a move
  else
```

Game action: Pseudocode

```
turn = Black // initially
computer = White // for example
while (game is not over) {
  if (turn == computer)
    computer should make a move
  else
    human should make a move
```

Game action: Pseudocode

```
turn = Black // initially
computer = White // for example
while (game is not over) {
  if (turn == computer)
    computer should make a move
  else
    human should make a move
    check if legal, and if not, game over
```

Game action: Pseudocode

```
turn = Black // initially
computer = White // for example
while (game is not over) {
  if (turn == computer)
    computer should make a move
  else
    human should make a move
    check if legal, and if not, game over
  if (game is not over)
```

Game action: Pseudocode

```
turn = Black // initially
computer = White // for example
while (game is not over) {
  if (turn == computer)
    computer should make a move
  else
    human should make a move
    check if legal, and if not, game over
  if (game is not over)
    if (moveAvailable(findOpposite(turn))
```

Game action: Pseudocode

```
turn = Black // initially
computer = White // for example
while (game is not over) {
  if (turn == computer)
    computer should make a move
  else
    human should make a move
    check if legal, and if not, game over
  if (game is not over)
    if (moveAvailable(findOpposite(turn))
      turn = findOpposite(turn)
```

Game action: Pseudocode

```
turn = Black // initially
computer = White // for example
while (game is not over) {
  if (turn == computer)
    computer should make a move
  else
    human should make a move
    check if legal, and if not, game over
  if (game is not over)
    if (moveAvailable(findOpposite(turn))
      turn = findOpposite(turn)
    else if (moveAvailable(turn))
```

Game action: Pseudocode

```
turn = Black // initially
computer = White // for example
while (game is not over) {
  if (turn == computer)
    computer should make a move
  else
    human should make a move
    check if legal, and if not, game over
  if (game is not over)
    if (moveAvailable(findOpposite(turn))
      turn = findOpposite(turn)
    else if (moveAvailable(turn))
      turn = turn
```

Game action: Pseudocode

```
turn = Black // initially
computer = White // for example
while (game is not over) {
  if (turn == computer)
    computer should make a move
  else
    human should make a move
    check if legal, and if not, game over
  if (game is not over)
    if (moveAvailable(findOpposite(turn))
      turn = findOpposite(turn)
    else if (moveAvailable(turn))
      turn = turn
    else // neither player has a turn
```

Game action: Pseudocode

```
turn = Black // initially
computer = White // for example
while (game is not over) {
  if (turn == computer)
    computer should make a move
  else
    human should make a move
    check if legal, and if not, game over
  if (game is not over)
    if (moveAvailable(findOpposite(turn))
      turn = findOpposite(turn)
    else if (moveAvailable(turn))
      turn = turn
    else // neither player has a turn
      game is over
```

Game action: Pseudocode

```
turn = Black // initially
computer = White // for example
while (game is not over) {
  if (turn == computer)
    computer should make a move
  else
    human should make a move
    check if legal, and if not, game over
  if (game is not over)
    if (moveAvailable(findOpposite(turn))
      turn = findOpposite(turn)
    else if (moveAvailable(turn))
      turn = turn
    else // neither player has a turn
      game is over
      figure out the winner
```

Game action: Pseudocode

```
turn = Black // initially
computer = White // for example
while (game is not over) {
  if (turn == computer)
    computer should make a move
  else
    human should make a move
    check if legal, and if not, game over
  if (game is not over)
    if (moveAvailable(findOpposite(turn))
      turn = findOpposite(turn)
    else if (moveAvailable(turn))
      turn = turn
    else // neither player has a turn
      game is over
      figure out the winner
}
```

Go Partway First

Go Partway First

- Our general idea has a lot of pieces to it — seems awfully hard!

Go Partway First

- Our general idea has a lot of pieces to it — seems awfully hard!

- Often, in software development, it is useful to implement something much simpler that isn’t exactly what we want, but yet brings us closer to the thing we want

Go Partway First

- Our general idea has a lot of pieces to it — seems awfully hard!

- Often, in software development, it is useful to implement something much simpler that isn't exactly what we want, but yet brings us closer to the thing we want

- Can you think of a simpler version that isn't complete, but would allow us to test things along the way?

Go Partway First

```
turn = Black // initially
computer = White // for example
while (true) {
  if (turn == computer)
    computer makes ANY legal move
  else
    human should make a move

  turn = findOpposite(turn)
}
```

What's missing?

What's missing?

What’s missing?

- Move legality check for human player

What's missing?

- Move legality check for human player
- Smarter moves for the computer

What’s missing?

- Move legality check for human player
- Smarter moves for the computer
- Some turn-taking processing

What’s missing?

- Move legality check for human player
- Smarter moves for the computer
- Some turn-taking processing
- Game over detection

Go Partway First

Go Partway First

- The beauty of this is we can start playing right away and test the turn-taking behaviour!

Go Partway First

- The beauty of this is we can start playing right away and test the turn-taking behaviour!

- How do we make the computer play any legal move?

Go Partway First

- The beauty of this is we can start playing right away and test the turn-taking behaviour!

- How do we make the computer play any legal move?

  - In your lab 7, you already printed out all available moves for each colour. You can use that code to make the computer play the first available move.

Back to the Full Implementation

Is a Move Available for a Colour?

- How would you decide if a player has an available move?
- Let’s break it down into small pieces …

Is a Move Available for a Colour?

Is a Move Available for a Colour?

- A possible breakdown:

Is a Move Available for a Colour?

- A possible breakdown:
  - First write a function `isValidMove` that accepts a colour, row and column as input (as well as the board) and determines if the (row, column) position is a legal move for that colour

Is a Move Available for a Colour?

- A possible breakdown:
  - First write a function `isValidMove` that accepts a colour, row and column as input (as well as the board) and determines if the (row, column) position is a legal move for that colour
    - How would this operate?

Is a Move Available for a Colour?

- A possible breakdown:
  - First write a function `isValidMove` that accepts a colour, row and column as input (as well as the board) and determines if the (row, column) position is a legal move for that colour
    - How would this operate?
    - Hint: Use your functionality from Lab 7

Is a Move Available for a Colour?

Is a Move Available for a Colour?

- You now have the function `isValidMove`

Is a Move Available for a Colour?

- You now have the function `isValidMove`

- Then write another function `moveAvailable` that accepts a colour as input (as well as the board) and determines if the colour has **any** valid move

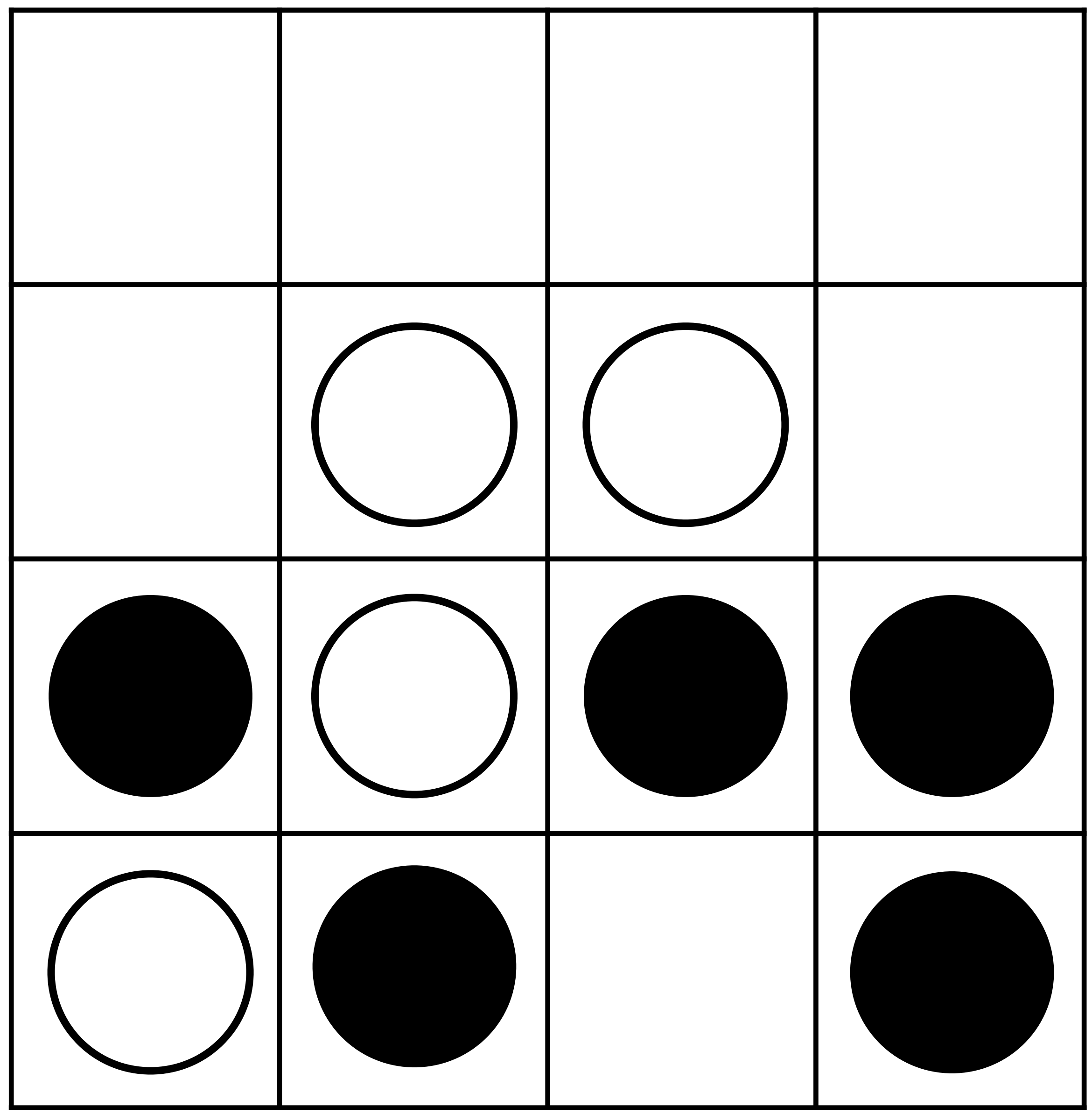
Is a Move Available for a Colour?

- You now have the function `isValidMove`
- Then write another function `moveAvailable` that accepts a colour as input (as well as the board) and determines if the colour has **any** valid move
  - How would this operate?

Is a Move Available for a Colour?

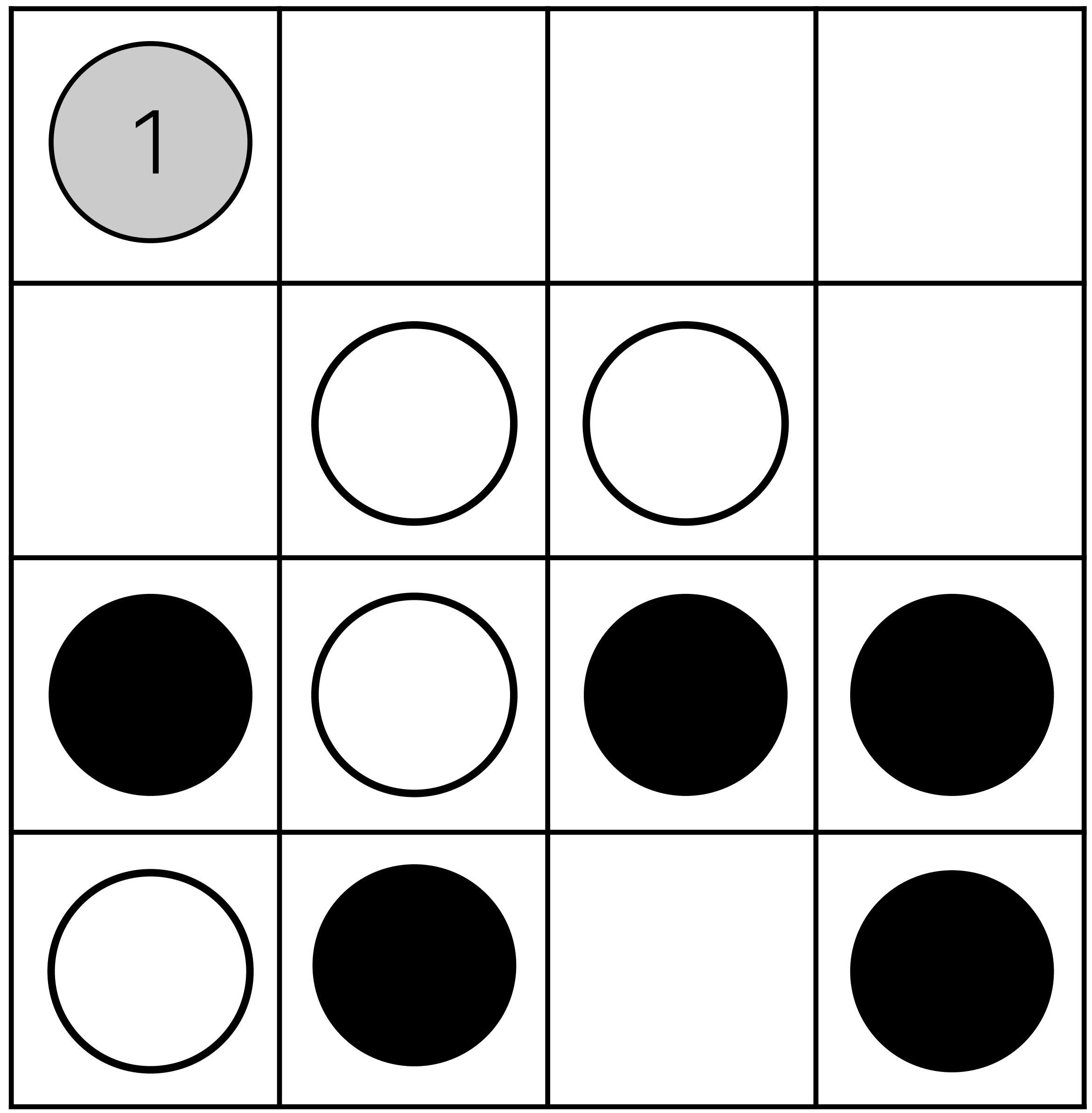
- You now have the function `isValidMove`
- Then write another function `moveAvailable` that accepts a colour as input (as well as the board) and determines if the colour has **any** valid move
  - How would this operate?
  - Hint: call `isValidMove` above

Smart Computer Moves: Part 1



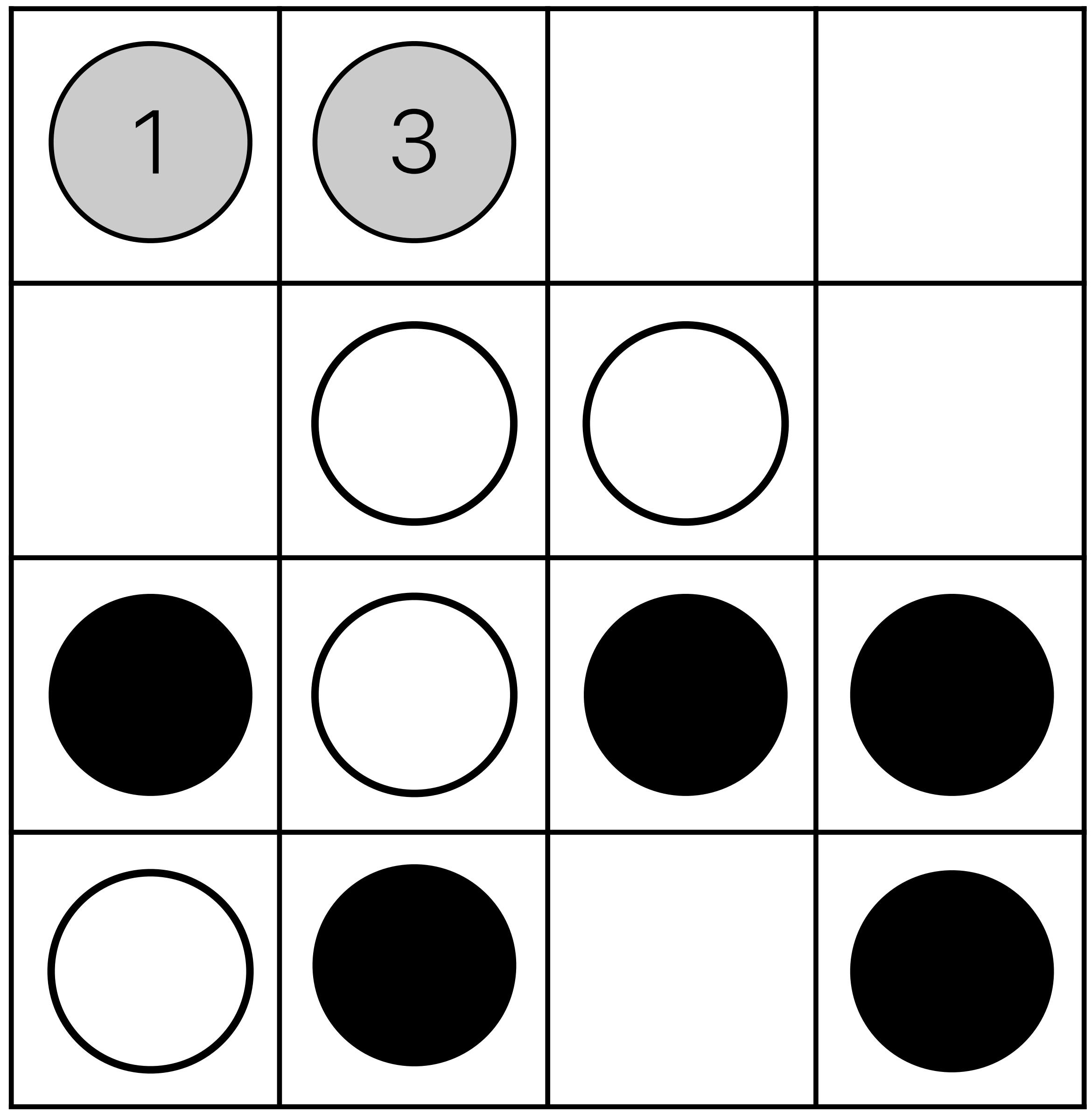
Computer plays Black

Smart Computer Moves: Part 1



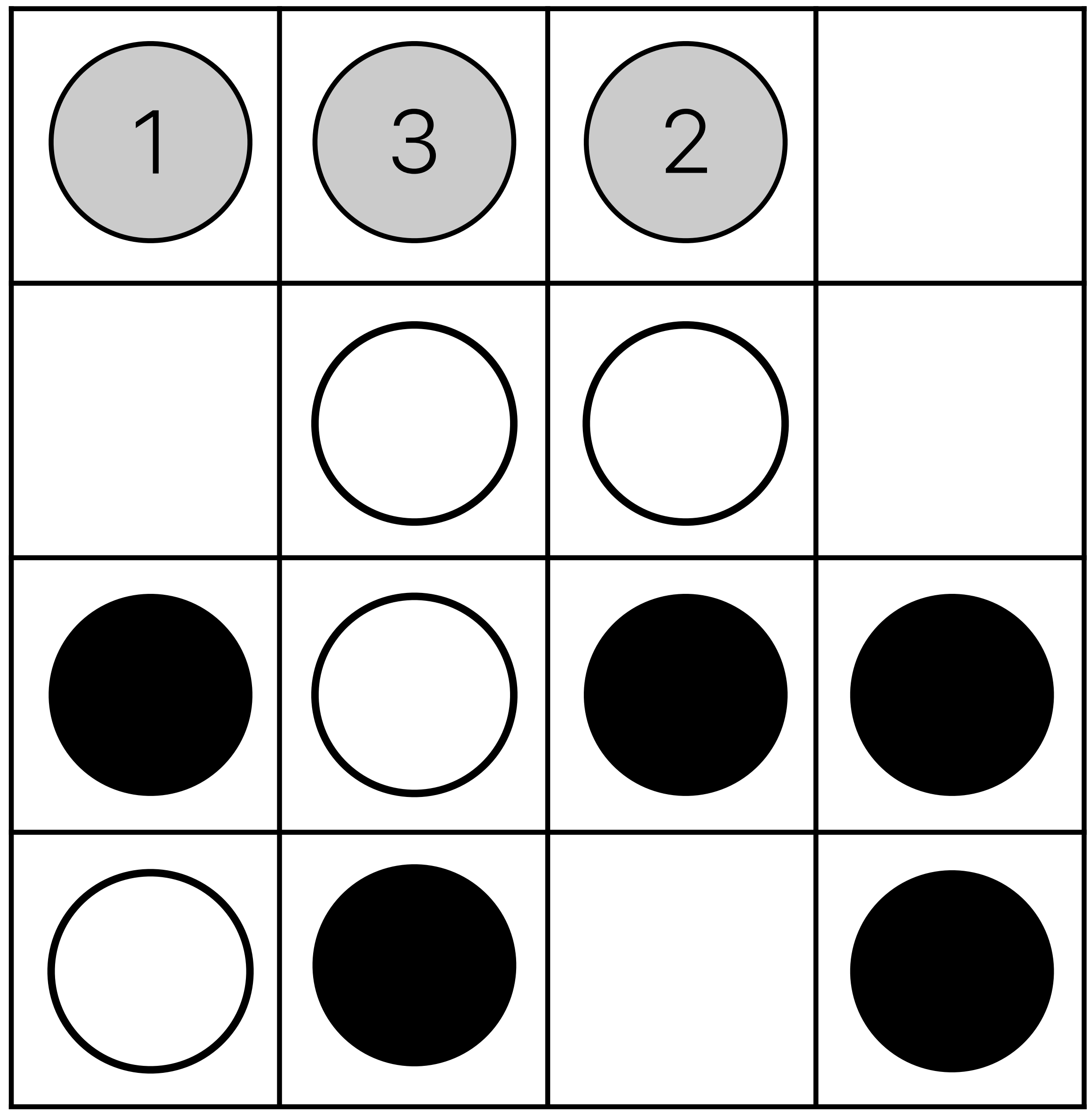
Computer plays Black

Smart Computer Moves: Part 1



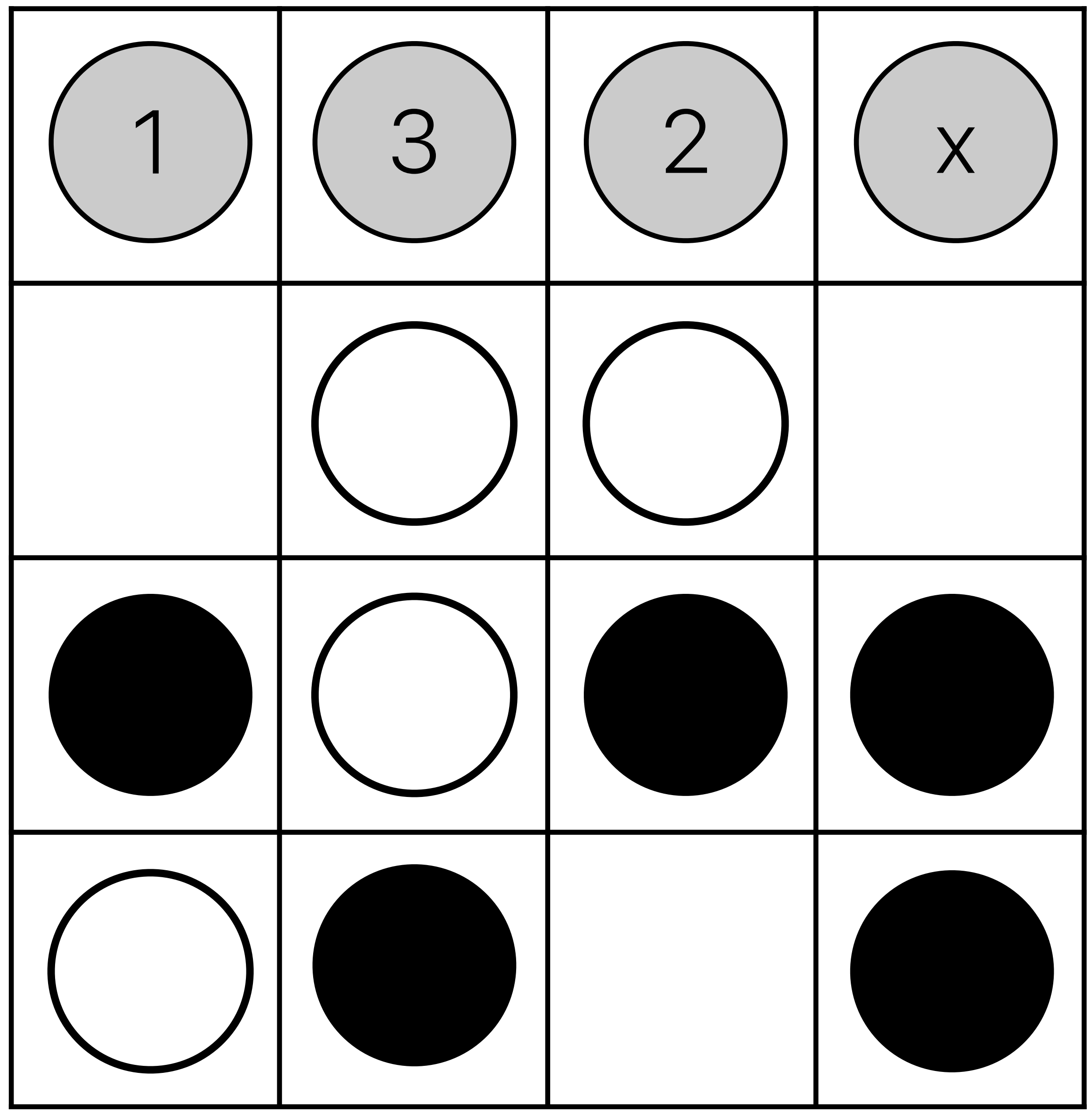
Computer plays Black

Smart Computer Moves: Part 1



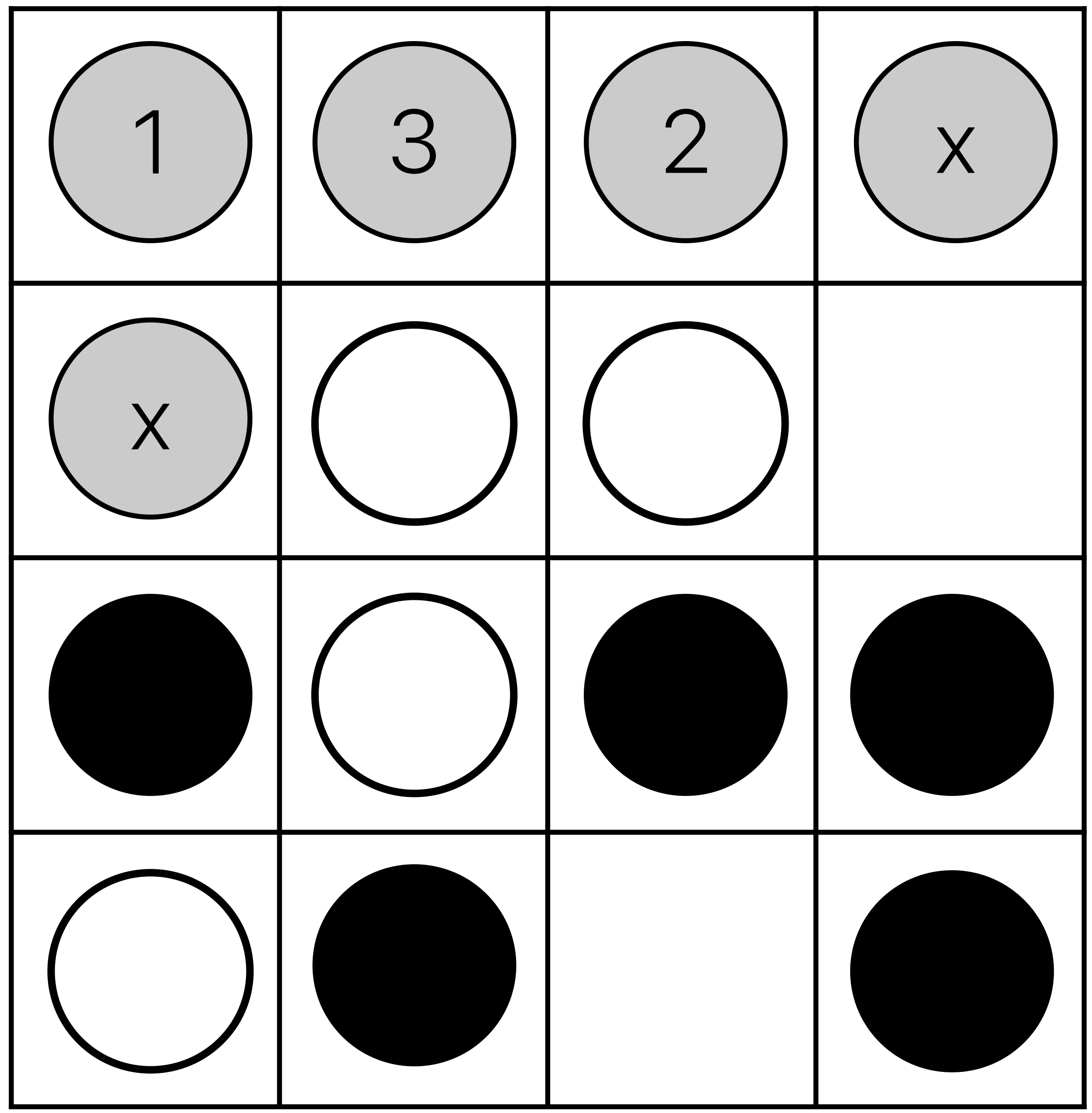
Computer plays Black

Smart Computer Moves: Part 1



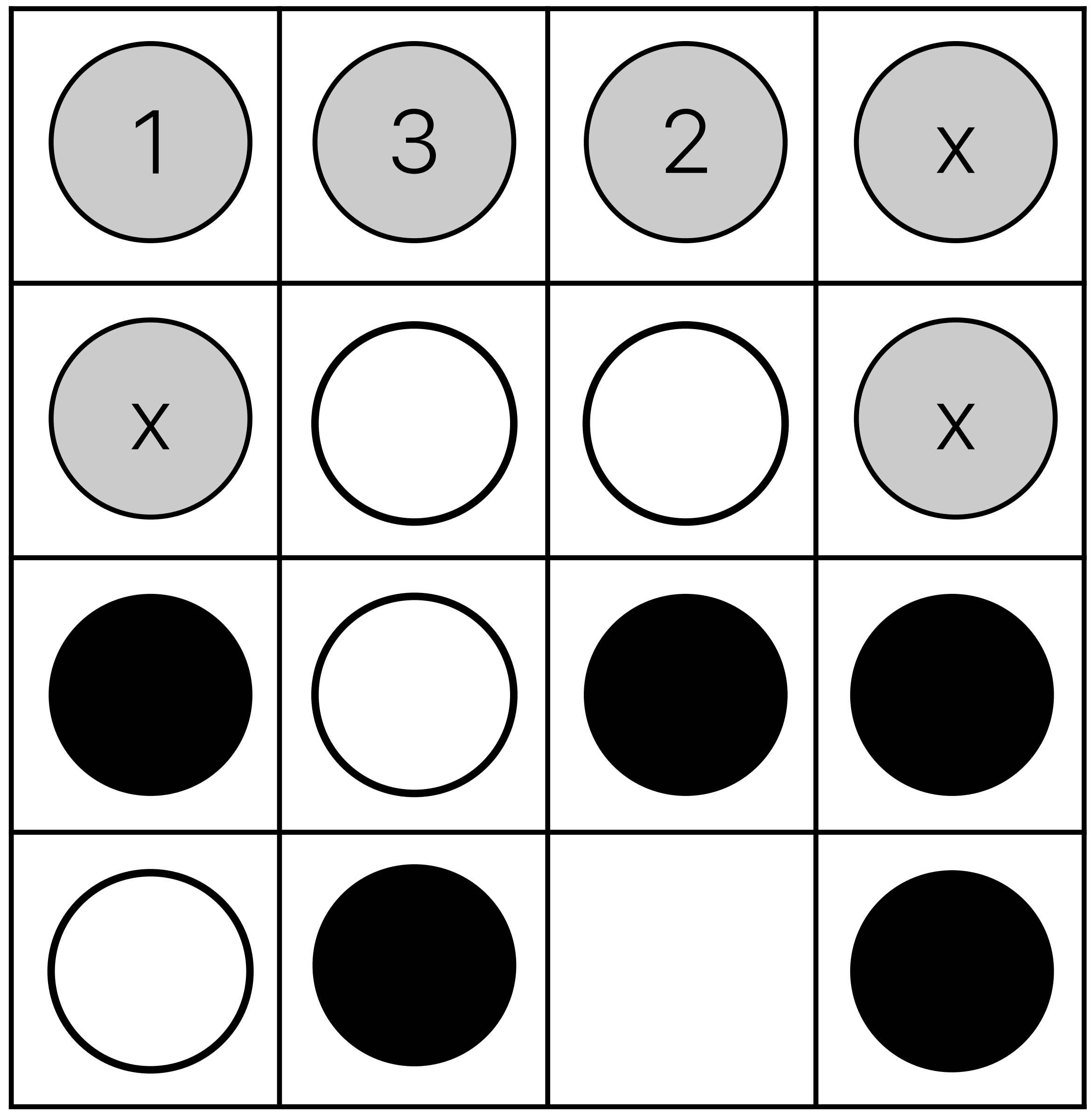
Computer plays Black

Smart Computer Moves: Part 1



Computer plays Black

Smart Computer Moves: Part 1



Computer plays Black

Computer Moves: Part I

Computer Moves: Part I

- First of all, notice that order in which the previous slides considered the locations: how to achieve that order?

Computer Moves: Part I

- First of all, notice that order in which the previous slides considered the locations: how to achieve that order?
  - Nested for loops: an outer loop over the rows, an inner loop over the columns

Computer Moves: Part I

- First of all, notice that order in which the previous slides considered the locations: how to achieve that order?
  - Nested for loops: an outer loop over the rows, an inner loop over the columns
- As the program walks the locations in order, what is happening?

Computer Moves: Part I

- First of all, notice that order in which the previous slides considered the locations: how to achieve that order?
  - Nested for loops: an outer loop over the rows, an inner loop over the columns
- As the program walks the locations in order, what is happening?
  - “Score” each location

Computer Moves: Part I

- First of all, notice that order in which the previous slides considered the locations: how to achieve that order?
  - Nested for loops: an outer loop over the rows, an inner loop over the columns
- As the program walks the locations in order, what is happening?
  - "Score" each location
  - Keep track of the highest score

Computer Moves: Part I

- First of all, notice that order in which the previous slides considered the locations: how to achieve that order?
  - Nested for loops: an outer loop over the rows, an inner loop over the columns
- As the program walks the locations in order, what is happening?
  - “Score” each location
  - Keep track of the highest score
  - Keep track of the (row, col) associated with the highest score

Computer Moves: Part I

- First of all, notice that order in which the previous slides considered the locations: how to achieve that order?
  - Nested for loops: an outer loop over the rows, an inner loop over the columns
- As the program walks the locations in order, what is happening?
  - “Score” each location
  - Keep track of the highest score
  - Keep track of the (row, col) associated with the highest score
  - In the event of a tied score, what happens?

Scoring a Location

Scoring a Location

- Say the computer is playing black, and **(row, col)** is a valid location for the computer to play

Scoring a Location

- Say the computer is playing black, and **(row, col)** is a valid location for the computer to play

- How does one determine the number of White tiles that would be flipped if Black plays at that location?

Scoring a Location

- Say the computer is playing black, and **(row, col)** is a valid location for the computer to play

- How does one determine the number of White tiles that would be flipped if Black plays at that location?

- One idea: actually play a Black tile at that location and do the flipping, as your Lab 7 program did

Scoring a Location

- Say the computer is playing black, and **(row, col)** is a valid location for the computer to play
- How does one determine the number of White tiles that would be flipped if Black plays at that location?
- One idea: actually play a Black tile at that location and do the flipping, as your Lab 7 program did
  - What's the problem with this approach? — "Undo"

Scoring a Location

- Say the computer is playing black, and **(row, col)** is a valid location for the computer to play

- How does one determine the number of White tiles that would be flipped if Black plays at that location?

- One idea: actually play a Black tile at that location and do the flipping, as your Lab 7 program did

  - What's the problem with this approach? — "Undo"

  - How do you solve this problem?

Introduce a boolean parameter

```
int checkValidAndFlip(char board[][26],
int row, int col, char colour, int n,
bool flip)

  // 'flip' tells me if I really want
  to flip the tiles when computing the
  score

  // the function returns the score for
  (row, col) position and the colour
```

An Alternative Solution

An Alternative Solution

- Before any move scoring, count the number of Black tiles on the board, call this **blackCountBeforeFlip**

An Alternative Solution

- Before any move scoring, count the number of Black tiles on the board, call this **blackCountBeforeFlip**
- For each candidate position where Black may play legally

An Alternative Solution

- Before any move scoring, count the number of Black tiles on the board, call this **blackCountBeforeFlip**
- For each candidate position where Black may play legally
  - Make a copy of the **entire** board (use nested **for** loops)

An Alternative Solution

- Before any move scoring, count the number of Black tiles on the board, call this **blackCountBeforeFlip**
- For each candidate position where Black may play legally
  - Make a copy of the **entire** board (use nested **for** loops)
    - Perhaps in a function called **copyBoard**

An Alternative Solution

- Before any move scoring, count the number of Black tiles on the board, call this **blackCountBeforeFlip**
- For each candidate position where Black may play legally
  - Make a copy of the **entire** board (use nested **for** loops)
    - Perhaps in a function called **copyBoard**
    - Play Black in the candidate position in the **board copy**

An Alternative Solution

- Before any move scoring, count the number of Black tiles on the board, call this **blackCountBeforeFlip**
- For each candidate position where Black may play legally
  - Make a copy of the **entire** board (use nested **for** loops)
    - Perhaps in a function called **copyBoard**
    - Play Black in the candidate position in the **board copy**
    - Flip tiles in the board copy using the Lab 7 solution

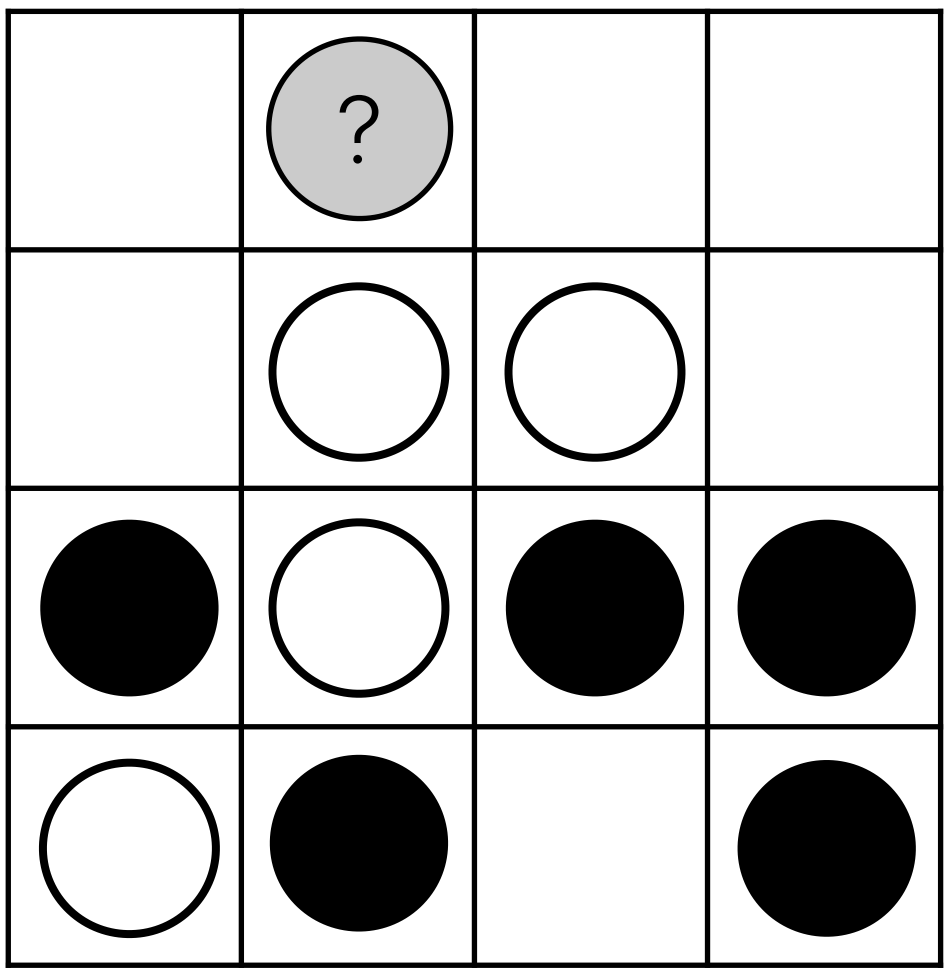
An Alternative Solution

- Before any move scoring, count the number of Black tiles on the board, call this **blackCountBeforeFlip**
- For each candidate position where Black may play legally
  - Make a copy of the **entire** board (use nested **for** loops)
    - Perhaps in a function called **copyBoard**
    - Play Black in the candidate position in the **board copy**
    - Flip tiles in the board copy using the Lab 7 solution
    - Count the number of Black tiles in the board copy, call it **blackCountAfterFlip**

An Alternative Solution

- Before any move scoring, count the number of Black tiles on the board, call this **blackCountBeforeFlip**
- For each candidate position where Black may play legally
  - Make a copy of the **entire** board (use nested **for** loops)
    - Perhaps in a function called **copyBoard**
    - Play Black in the candidate position in the **board copy**
    - Flip tiles in the board copy using the Lab 7 solution
    - Count the number of Black tiles in the board copy, call it **blackCountAfterFlip**
    - **Score** for candidate location **= blackCountAfterFlip - blackCountBeforeFlip - 1**

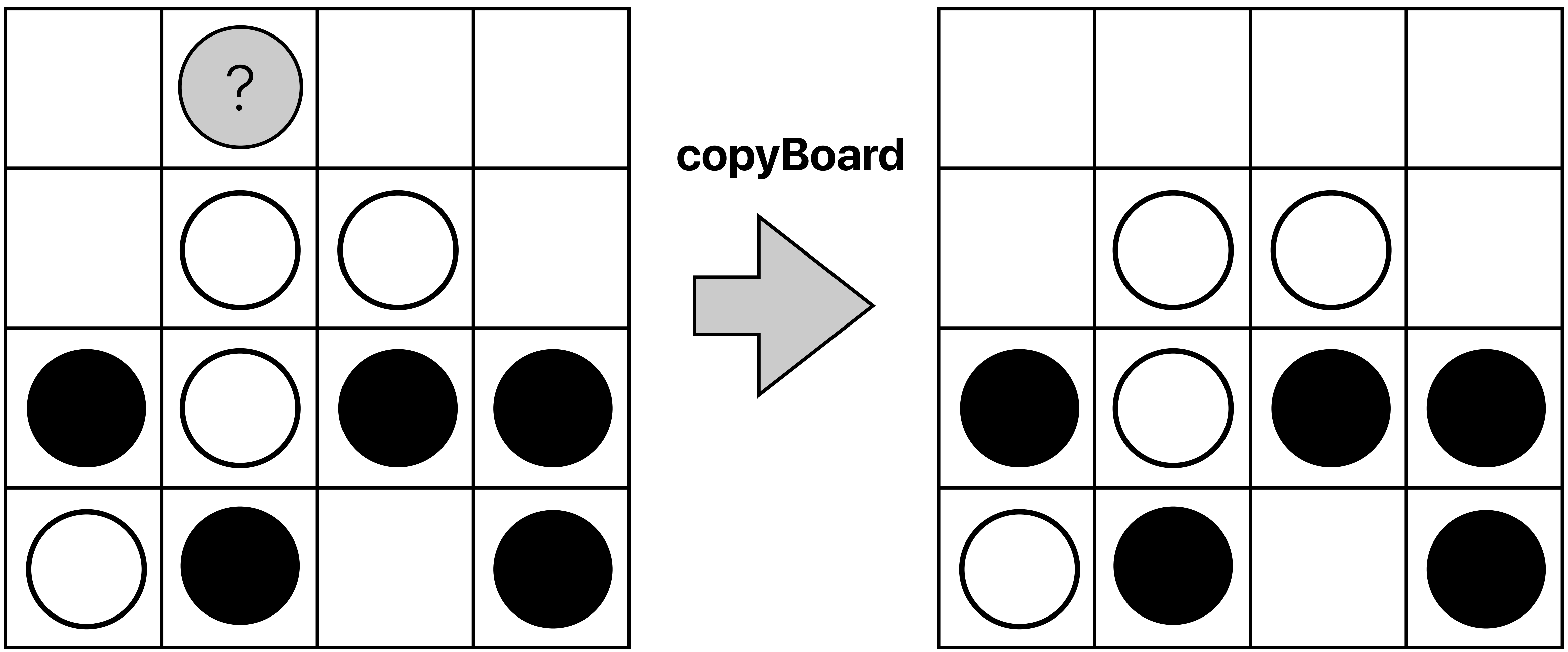
Scoring a Location



blackCountBeforeFlip = 5

Computer plays Black

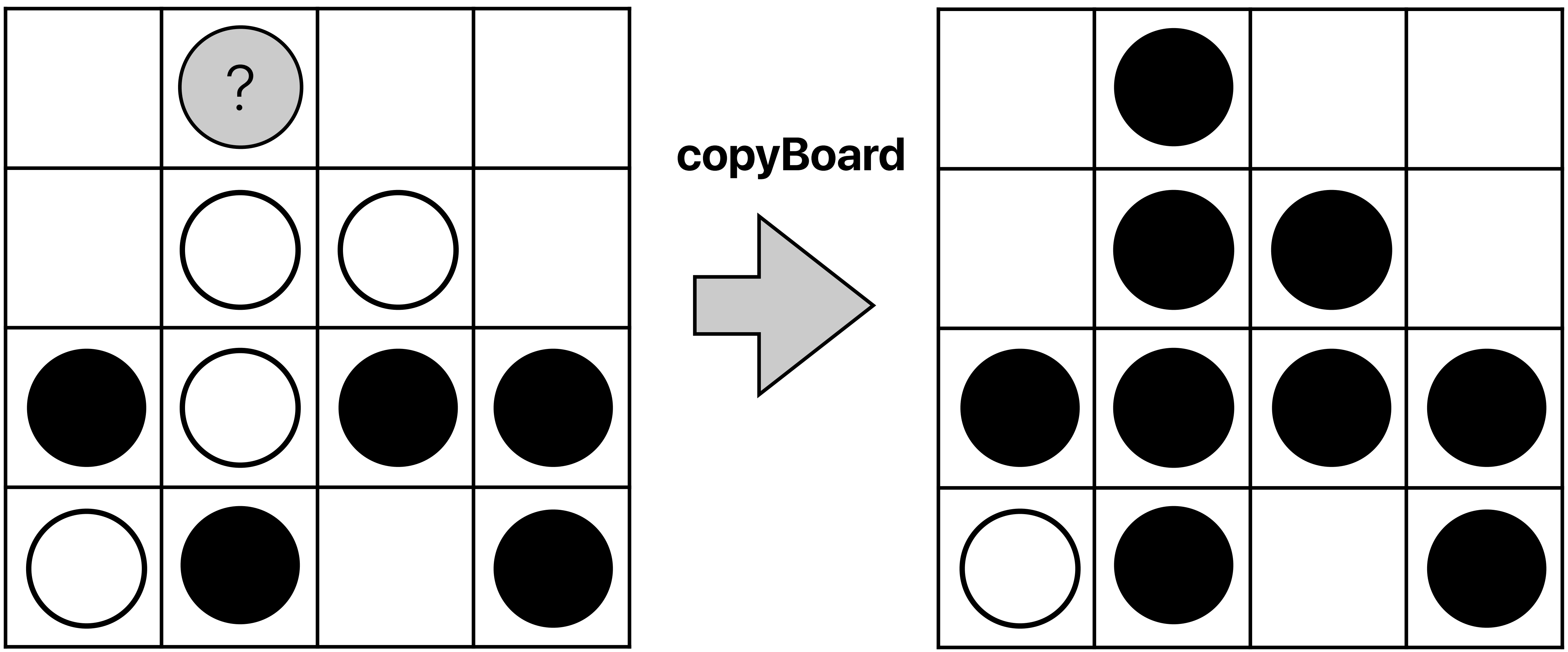
Scoring a Location



blackCountBeforeFlip = 5

Computer plays Black

Scoring a Location

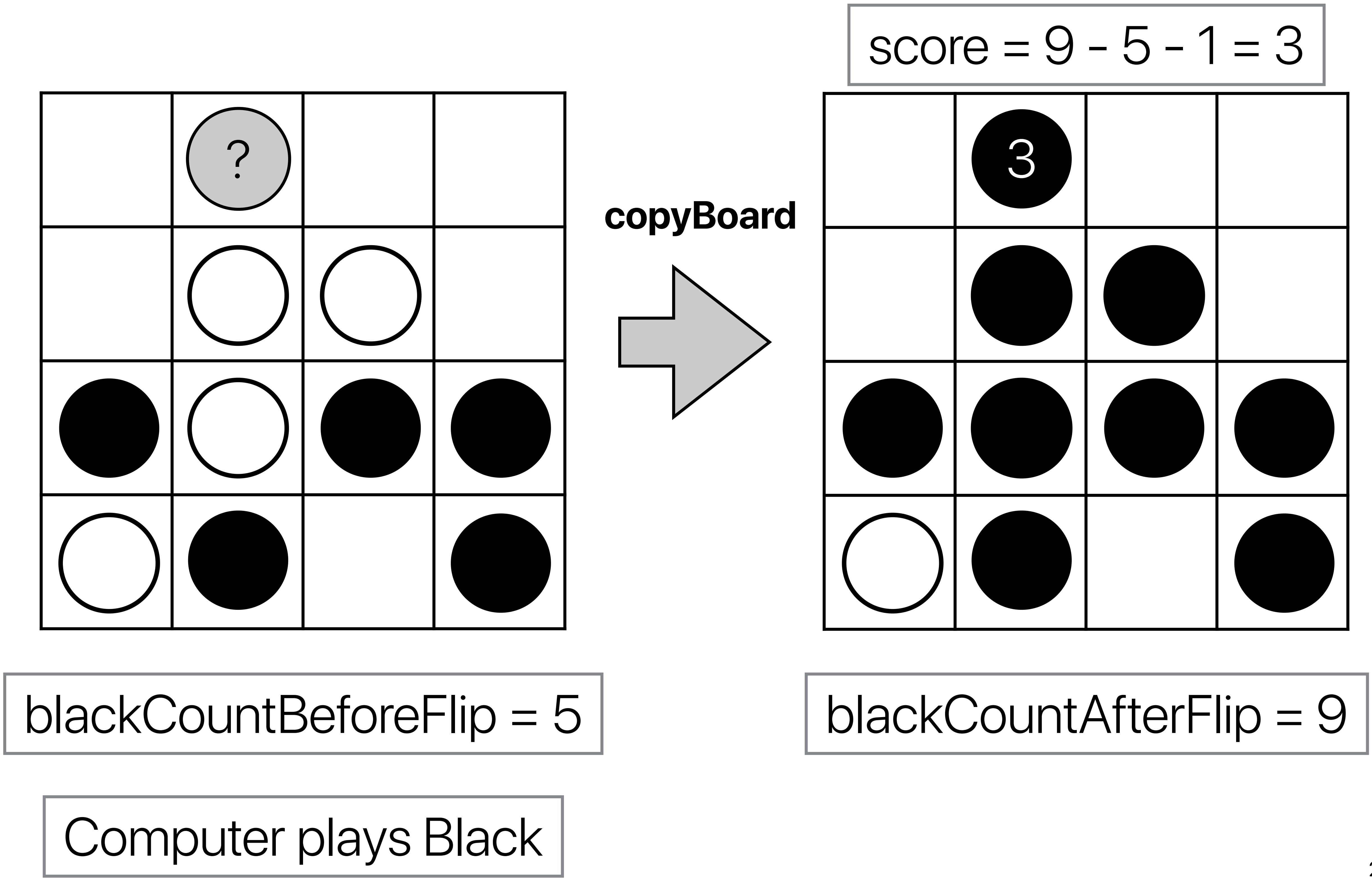


blackCountBeforeFlip = 5

blackCountAfterFlip = 9

Computer plays Black

Scoring a Location



Part II: Strategies

- This is really just about different (more clever) ways of scoring locations
- The Part I approach is called “**greedy**”
  - Goes for the maximum # of flips, regardless of consequences
- What are other factors to consider in a location’s score?

Part II: Strategies

Part II: Strategies

- This is really just about different (more clever) ways of scoring locations

Part II: Strategies

- This is really just about different (more clever) ways of scoring locations
- The Part I approach is called “**greedy**”

Part II: Strategies

- This is really just about different (more clever) ways of scoring locations
- The Part I approach is called "**greedy**"
  - Goes for the maximum # of flips, regardless of consequences

Part II: Strategies

- This is really just about different (more clever) ways of scoring locations
- The Part I approach is called "**greedy**"
  - Goes for the maximum # of flips, regardless of consequences
- What are other factors to consider in a location's score?

Part II: Strategies

- This is really just about different (more clever) ways of scoring locations
- The Part I approach is called “**greedy**”
  - Goes for the maximum # of flips, regardless of consequences
- What are other factors to consider in a location’s score?
  - Corners of the board are good

Part II: Strategies

- This is really just about different (more clever) ways of scoring locations
- The Part I approach is called "**greedy**"
  - Goes for the maximum # of flips, regardless of consequences
- What are other factors to consider in a location's score?
  - Corners of the board are good
  - Fewer available counter-moves for the opponent

Part II: Strategies

- This is really just about different (more clever) ways of scoring locations
- The Part I approach is called "**greedy**"
  - Goes for the maximum # of flips, regardless of consequences
- What are other factors to consider in a location's score?
  - Corners of the board are good
  - Fewer available counter-moves for the opponent
  - Opponent's potential response to computer's move

Composite scoring

- You can consider a “composite” scoring function:

```
score = a * #flips + b * #corners + c
* #opponentMovesEliminated

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

Finding # of Available Opponent Moves

Finding # of Available Opponent Moves

- Imagine a computer is playing Black, and two locations (r1, c1) and (r2, c2) result in 3 White tiles being flipped

Finding # of Available Opponent Moves

- Imagine a computer is playing Black, and two locations (r1, c1) and (r2, c2) result in 3 White tiles being flipped
- Say (r1, c1) leaves White with 4 locations to play, and (r2, c2) leaves White with 1 location to play

Finding # of Available Opponent Moves

- Imagine a computer is playing Black, and two locations (r1, c1) and (r2, c2) result in 3 White tiles being flipped
- Say (r1, c1) leaves White with 4 locations to play, and (r2, c2) leaves White with 1 location to play
- (r2, c2) may be a better move

Finding # of Available Opponent Moves

- Imagine a computer is playing Black, and two locations (r1, c1) and (r2, c2) result in 3 White tiles being flipped
- Say (r1, c1) leaves White with 4 locations to play, and (r2, c2) leaves White with 1 location to play
- (r2, c2) may be a better move
- How would you find, for a candidate move position, the number of locations in which the opponent could play after the flipping?

Finding # of Available Opponent Moves

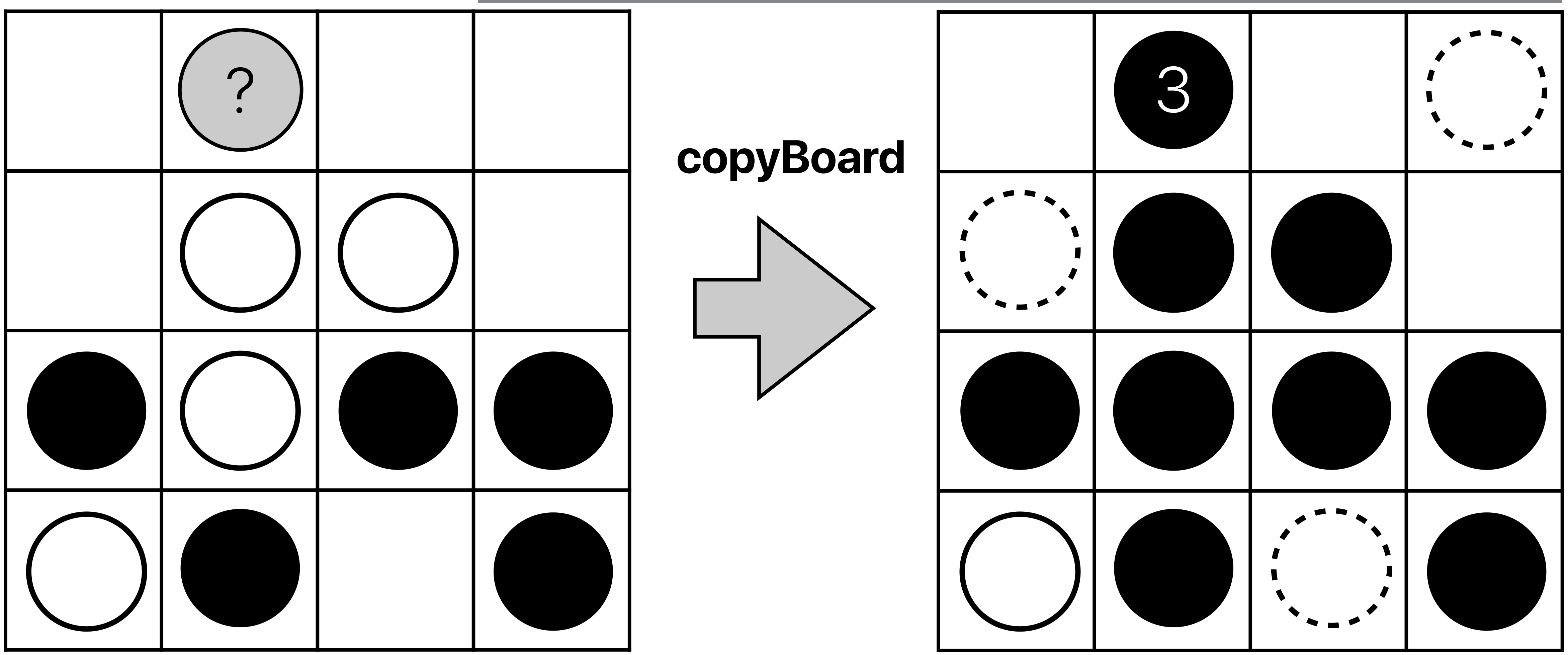
- Imagine a computer is playing Black, and two locations (r1, c1) and (r2, c2) result in 3 White tiles being flipped
- Say (r1, c1) leaves White with 4 locations to play, and (r2, c2) leaves White with 1 location to play
- (r2, c2) may be a better move
- How would you find, for a candidate move position, the number of locations in which the opponent could play after the flipping?
  - Use the copy of the board

Finding # of Available Opponent Moves

- Imagine a computer is playing Black, and two locations (r1, c1) and (r2, c2) result in 3 White tiles being flipped
- Say (r1, c1) leaves White with 4 locations to play, and (r2, c2) leaves White with 1 location to play
- (r2, c2) may be a better move
- How would you find, for a candidate move position, the number of locations in which the opponent could play after the flipping?
  - Use the copy of the board
  - Use a nested loop and the `isValidMove` function we talked about before

Scoring a Location

In how many locations can White play? **3**



blackCountBeforeFlip = 5

blackCountAfterFlip = 9

Computer plays Black

With a copy of the board as a “scratch pad,” we can call **isValidMove** to find out possible opponent moves!

More Sophisticated: Game Tree

More Sophisticated: Game Tree

- Board-game playing programs (Chess, Go, etc.) use a “game tree”

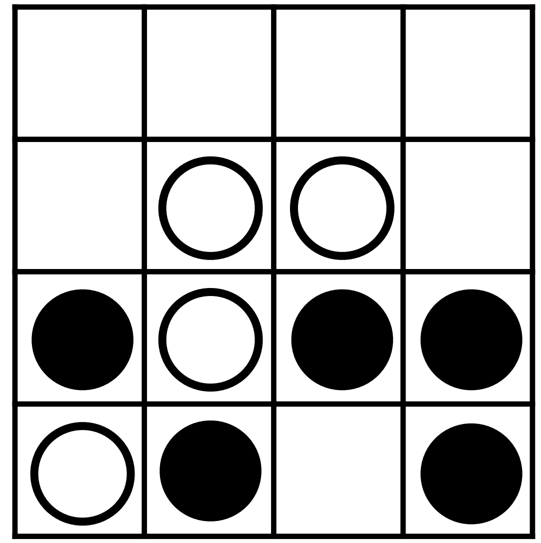
More Sophisticated: Game Tree

- Board-game playing programs (Chess, Go, etc.) use a "game tree"
- When humans play these games, good players can "look several moves into the future" to see eventual consequences of a move

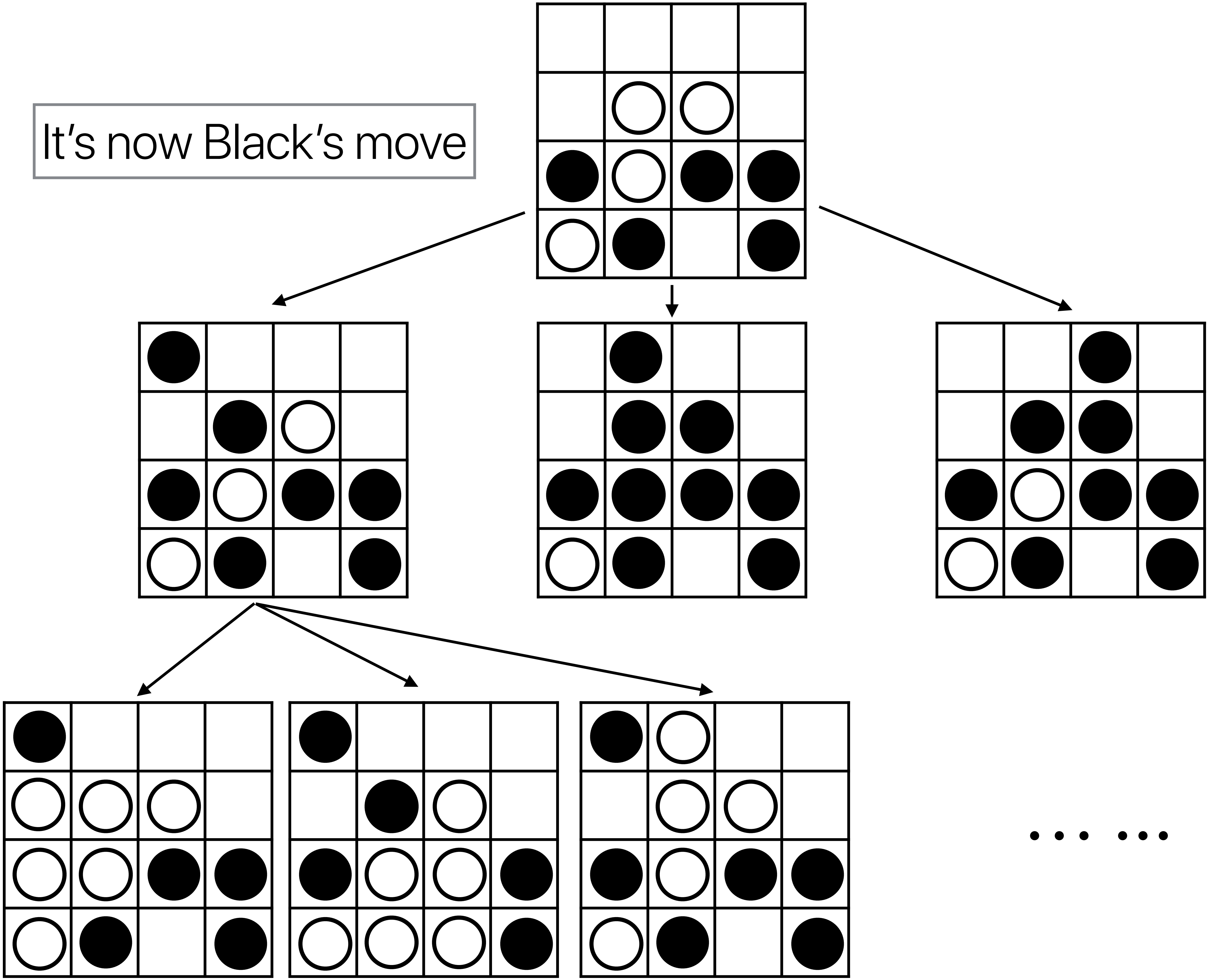
More Sophisticated: Game Tree

- Board-game playing programs (Chess, Go, etc.) use a "game tree"
- When humans play these games, good players can "look several moves into the future" to see eventual consequences of a move
- A "game tree" is a way of looking into the future within a computer program

It’s now Black’s move



It’s now Black’s move



Game Tree

Game Tree

- Still needs a way to “score” candidate moves

Game Tree

- Still needs a way to “score” candidate moves
- Our example game tree has a depth of 2, and to be smarter we need to go deeper

Game Tree

- Still needs a way to "score" candidate moves
- Our example game tree has a depth of 2, and to be smarter we need to go deeper
- For each of Black's candidate moves, expect the opponent to play its best possible response

Game Tree: Challenges

Game Tree: Challenges

- Manage copies of the board representing future configurations

Game Tree: Challenges

- Manage copies of the board representing future configurations
- Handling the case when one player gets a chance to make multiple moves in succession (the opponent has no valid moves)

Game Tree: Challenges

- Manage copies of the board representing future configurations
- Handling the case when one player gets a chance to make multiple moves in succession (the opponent has no valid moves)
- Google "game tree" and "minimax" to learn more about it

Fourth time in this course:

APS 105 competition leaderboard

http://aps105.ece.utoronto.ca:8090

The APS 105 Reversi Leaderboard

Reversi Project Top Leaderboard

Rank	Student	Score	Indicator	Status
0	SiweiHe	∞	∞ %	PASS
1	harri658	1911.05	100 %	TLE
2	dicksimo	1810.2	94.72 %	TLE
3	guptar56	1747.25	91.43 %	TLE
4	pacynkod	1746.9	91.41 %	IM TLE

The APS 105 Reversi Leaderboard

- Entered automatically when you submit your Lab 8 Part 2 to examify.ca, if your submission passes the test cases

Reversi Project Top Leaderboard

Rank	Student	Score	Indicator	Status
0	SiweiHe	∞	∞ %	PASS
1	harri658	1911.05	100 %	TLE
2	dicksimo	1810.2	94.72 %	TLE
3	guptar56	1747.25	91.43 %	TLE
4	pacynkod	1746.9	91.41 %	IM TLE

The APS 105 Reversi Leaderboard

- 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

Reversi Project Top Leaderboard

Rank	Student	Score	Indicator	Status
0	SiweiHe	∞	∞ %	PASS
1	harri658	1911.05	100 %	TLE
2	dicksimo	1810.2	94.72 %	TLE
3	guptar56	1747.25	91.43 %	TLE
4	pacynkod	1746.9	91.41 %	IM TLE

The APS 105 Reversi Leaderboard

- 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

Reversi Project Top Leaderboard

Rank	Student	Score	Indicator	Status
0	SiweiHe	∞	∞ %	PASS
1	harri658	1911.05	100 %	TLE
2	dicksimo	1810.2	94.72 %	TLE
3	guptar56	1747.25	91.43 %	TLE
4	pacynkod	1746.9	91.41 %	IM TLE

The APS 105 Reversi Leaderboard

- 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

Reversi Project Top Leaderboard

Rank	Student	Score	Indicator	Status
0	SiweiHe	∞	∞ %	PASS
1	harri658	1911.05	100 %	TLE
2	dicksimo	1810.2	94.72 %	TLE
3	guptar56	1747.25	91.43 %	TLE
4	pacynkod	1746.9	91.41 %	IM TLE

Enjoy the lab and have fun!