Hex Rules & Implementation

CMPUT 355: Games, Puzzles, and Algorithms

Lecture Outline

1. Logistics & Recap
2. Rules of Hex
3. Implementation Issues

Logistics

- **TA Office hours:** Every **Thursday** from **1:00pm-2:00pm** in **UCOMM 3-136**
  - Drop in basis; just show up and ask questions
  - Started yesterday (how'd it go?)
- **Practice quiz questions:** Released later today (**Jan 16**)
  - Answers released Tuesday (Jan 20)
- **Quiz 1:** Friday, **Jan 23**
  - In-class, full 50 minutes
  - No need to email if you have to miss it; up to 3 replaced by final exam automatically
  - Questions will be very similar to practice questions
  - (at least 3 will be *suspiciously* similar!)

Logistics: Code walkthroughs

Question: Which style of code walkthrough did you prefer?

go_helper.py

go_helper.py [master]

```
156   def makemove(self, where, color):
157     assert (self.brd[where] == EMPTY), 'that point is not empty
    '
158     self.brd = change_string(self.brd, where, color)
159
160     cap = []
161     for j in self.nbr_offsets:
162       x = where + j
163       if self.brd[x] == opponent(color):
164         cap += self.captured(x, opponent(color))
165
166     if (len(cap)>0):
167       #print('removing captured group at', point_to_alphanum(wh
    ere, self.C))
168       for j in cap:
169         self.brd = change_string(self.brd, j, EMPTY)
170       return cap, True  # move ok sofar
171
172     if self.captured(where, color):
173       print('whoops, no liberty there: not allowed')
```

hexgo/stone_board.py

Track groups and liberties instead of searching after each move:

```
self.stones   = [set(), set()]  # start with empty board
self.blocks   = {}  # point (block name) -> stones in block
self.nbrs     = {}  # point (block name) -> neighbors
self.liberties = {}  # point (block name) -> liberties
self.parents  = {}  # point -> parent in block
```

Use find and union to update tracking after each move:

```
def add_stone(self, color, point):
    self.stones[color].add(point)
    self.blocks[point].add(point)

    for n in self.nbrs[point]:
        if n in self.stones[color]: # same-color nbr
            self.merge_blocks(n, point)
        if n in self.stones[Cell.opponent(color)]: # opponent nbr
            self.remove_liberties(n, point)
```

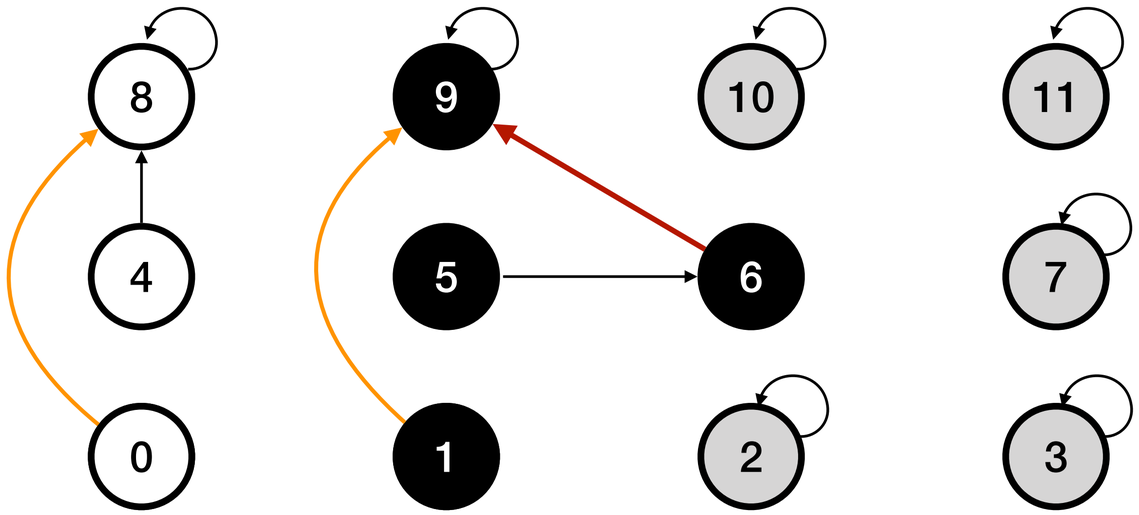
Union to update block membership:

```
def merge_blocks(self, p, q):
    proot, qroot = UF.union(self.parents, p, q)
    self.blocks[proot].update(self.blocks[qroot])
    self.liberties[proot].update(self.liberties[qroot])
    self.liberties[proot] -= self.blocks[proot]
```

Find to update block liberties:

```
def remove_liberties(self, p, q):
    proot = UF.find(self.parents, p)
    qroot = UF.find(self.parents, q)
    self.liberties[proot] -= self.blocks[qroot]
    self.liberties[qroot] -= self.blocks[proot]
```

Recap: Union-Find Operations in Go



After 5 is placed, we check neighbours 6, 9, 4, 1 for merge operations

1. Join 5's block (i.e., 5) to o 6's block, so it points to 6's block's root (i.e., 6)
2. Next, join 5's block (i.e., 6) and 9's block (i.e., 9).
3. Without the union-by-rank optimization, that could mean *either* 6 points to 9 or 9 points to 6; I chose to show 6 points to 9.
4. But with the union-by-rank optimization, we would have to make 9 point to 6, because it has strictly lower rank than 6.

- Black stone on 6
- White stone on 8
- Black stone on 9
- White stone on 4
- Black stone on 5
- White stone on 0
- Black stone on 1
  - 0's block's liberties become empty

Recap: hexgo/stone_board.py

Track groups and liberties instead of searching after each move:

```
self.stones   = [set(), set()]  # start with empty board
self.blocks   = {}   # point (block name) -> stones in block
self.nbrs     = {}   # point (block name) -> neighbors
self.liberties = {}  # point (block name) -> liberties
self.parents  = {}   # point -> parent in block
```

Use find and union to update tracking after each move:

```
def add_stone(self, color, point):
    self.stones[color].add(point)
    self.blocks[point].add(point)

    for n in self.nbrs[point]:
        if n in self.stones[color]: # same-color nbr
            self.merge_blocks(n, point)
        if n in self.stones[Cell.opponent(color)]: # opponent nbr
            self.remove_liberties(n, point)
```

Union to update block membership:

```
def merge_blocks(self, p, q):
    proot, qroot = UF.union(self.parents, p, q)
    self.blocks[proot].update(self.blocks[qroot])
    self.liberties[proot].update(self.liberties[qroot])
    self.liberties[proot] -= self.blocks[proot]
```

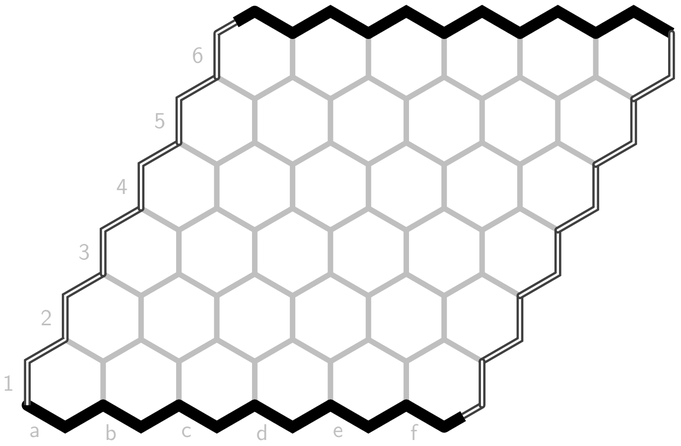
Find to update block liberties:

```
def remove_liberties(self, p, q):
    proot = UF.find(self.parents, p)
    qroot = UF.find(self.parents, q)
    self.liberties[proot] -= self.blocks[qroot]
    self.liberties[qroot] -= self.blocks[proot]
```

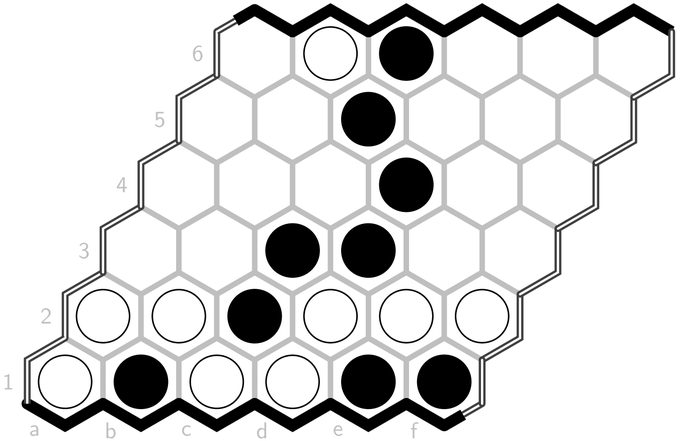
Hex Rules

- There are two players, Black and White
- Each player makes a **move** in alternation, starting with Black
  - A move is placing a stone in an Empty hexagonal cell
- Two cells are **adjacent** if they share a side
  - Each cell has 2-6 neighbours
- Two facing **borders** of the board are Black,
  the other two edges are White
  - Each bottom cell is adjacent to the bottom border, etc.
  - Each **corner** cell is thus adjacent to **two** borders
- The game ends when one player has joined the two edges of their own colour with a path of stones
  - The player who joins their edges **wins**

Empty Hex board

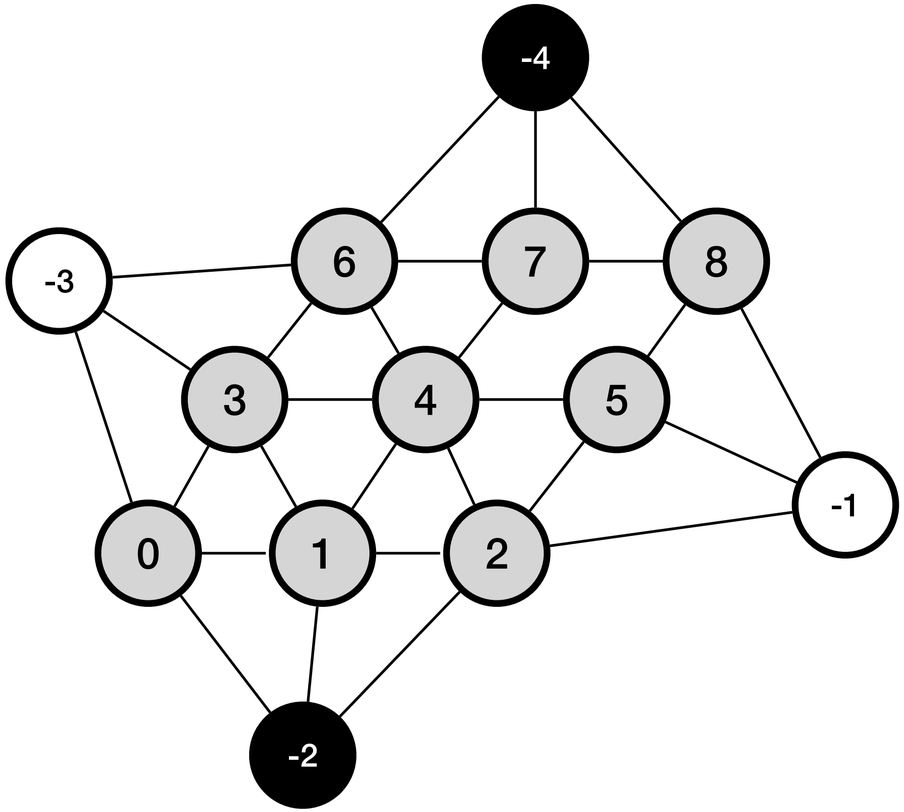


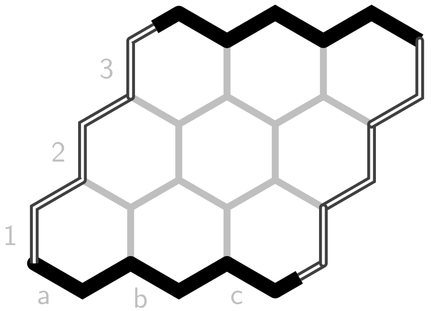
Winning position for White



Graph Representation

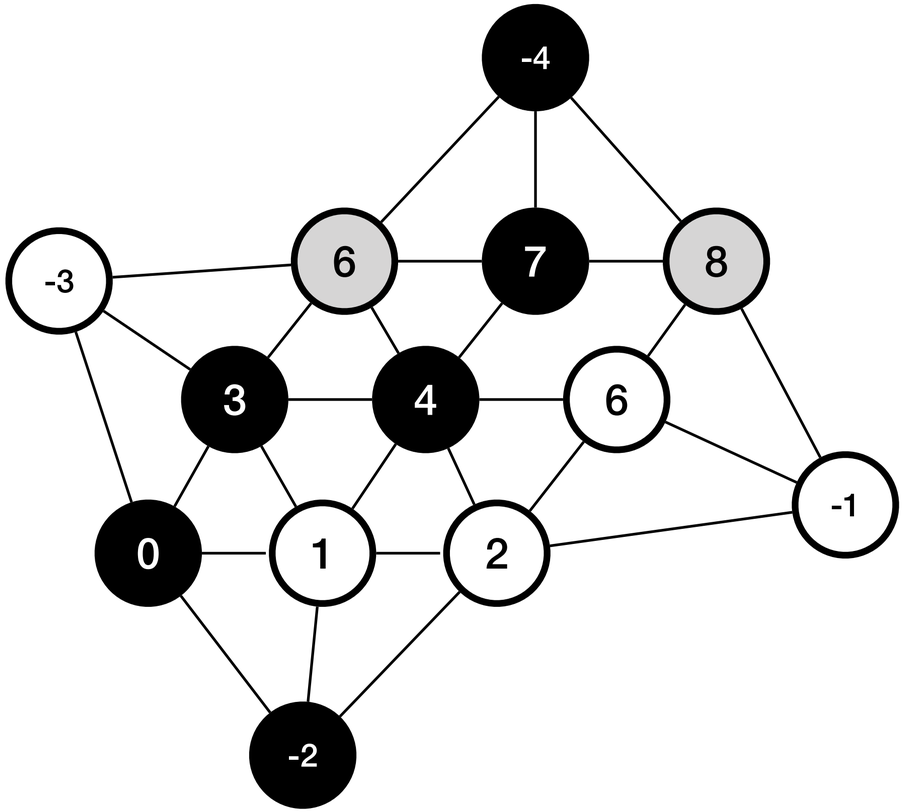
- As with Go, can represent the board as a graph
  - One node per hex cell
  - Edge between each pair of adjacent cells
  - Plus four extra nodes for borders
- Each node can be coloured Black, White, or Empty
- **Question:** How to check for a winning condition on the graph?

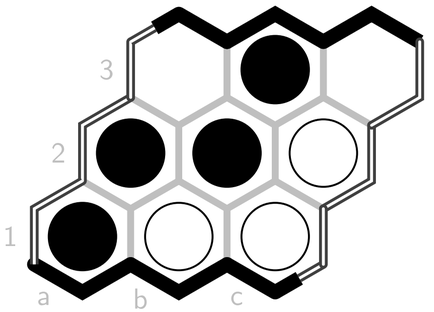




Graph Representation

- As with Go, can represent the board as a graph
  - One node per hex cell
  - Edge between each pair of adjacent cells
  - Plus four extra nodes for borders
- Each node can be coloured Black, White, or Empty
- **Question:** How to check for a winning condition on the graph?



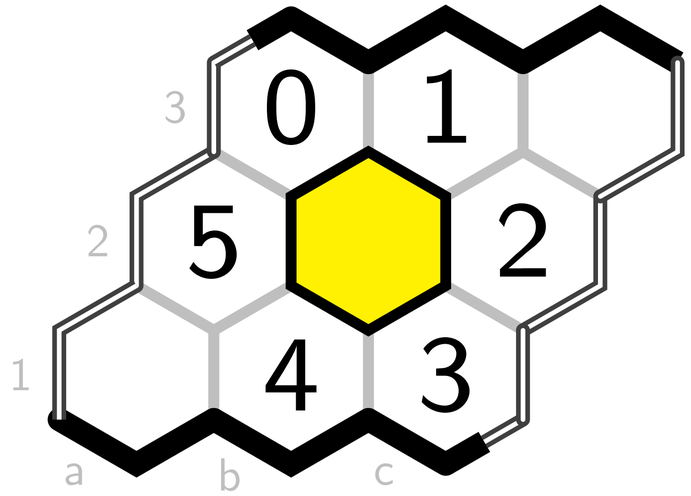


Neighbours in **hexgo/stone_board.py**

Instead of calculating a cell's neighbours every time we need them, we compute all of them **in advance** and then look them in **self.nbrs** whenever we need them. The following code is all in **Stone_board.__init__**:

Each cell has (up to) six neighbours:

```
if gt: # hex game
    self.top, self.rgt, self.btm, self.lft = -4, -3, -2, -1
    self.border = range(self.top, 0) # -4, -3, -2, -1
    self.p_range = range(self.top, self.n) # -4, ..., rows*cols-1
    self.nbr_offset = ((-1,0),(-1,1),(0,1),(1,0),(1,-1),(0,-1))
    #   0 1
    #  5 . 2
    #   4 3
```



Create an empty set to store each cell and border's neighbours:

```
    for point in self.p_range:
        self.nbrs[point] = set()
```

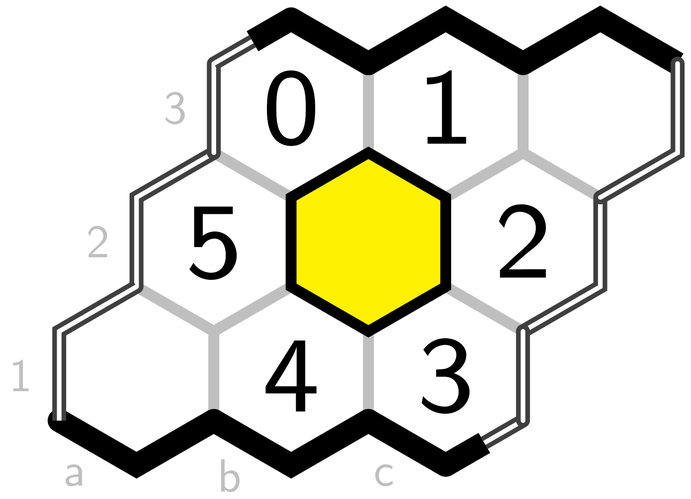
Initializing Neighbours

First record all the neighbouring **cells**:

```
r_range, c_range  = range(self.r), range(self.c)
for r in range(self.r):
    for c in range(self.c):
      for (y,x) in self.nbr_offset:
        if r+y in r_range and c+x in c_range:
          self.nbrs[Pt.rc_point(r,c,self.c)].add(Pt.rc_point(r+y,c+x,self.c))
```

Then the neighbouring **borders**:

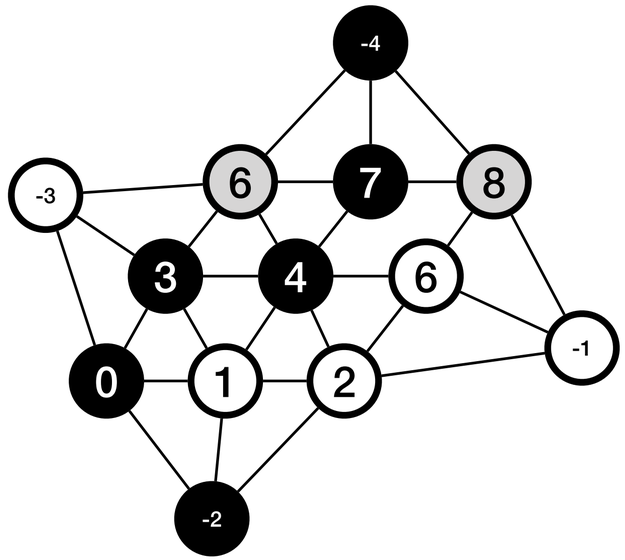
```
    for j in range(self.c):
        self.nbrs[self.top].add(j)
        self.nbrs[j].add(self.top)
        self.nbrs[self.btm].add(self.n-j-1)
        self.nbrs[self.n-j-1].add(self.btm)
    for k in range(self.r):
        self.nbrs[self.lft].add(k*self.c)
        self.nbrs[k*self.c].add(self.lft)
        self.nbrs[self.rgt].add(k*self.c+self.c-1)
        self.nbrs[k*self.c+self.c-1].add(self.rgt)
```



Paths

- A path in hex is just a group of adjacent cells with the same colour
  - It's a **block**!
- Can use *exactly* the same union-find approach to track paths as in Go:

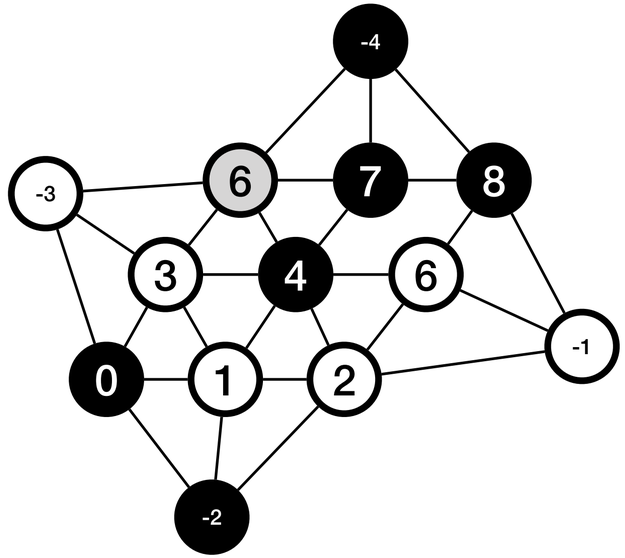
```
def add_stone(self, color, point):
    self.stones[color].add(point)
     self.blocks[point].add(point)

     for n in self.nbrs[point]:
         if n in self.stones[color]: # same-color nbr
             self.merge_blocks(n, point)
         if n in self.stones[Cell.opponent(color)]: # opponent nbr
             self.remove_liberties(n, point)
```

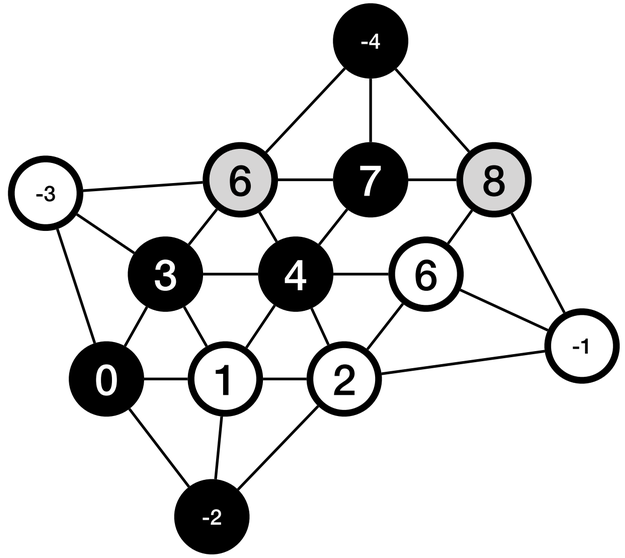


End of Game

- **Question:** How can we detect the end of the game using our union-find approach?
  - **Black win:** There is a path from the top to the bottom, so there must be a block that contains both the top and bottom borders
  - **White win:** Must be a block containing both the left and right borders
- So just check if top and bottom (or left and right) are in the same block:

```
def hex_win(self, cell_color):
    if self.game_type != Game.hex_game:
        return False
    if cell_color == Cell.b:
        return UF.in_same_block(self.parents, self.top, self.btm)
    return UF.in_same_block(self.parents, self.lft, self.rgt)
```





Summary

- Rules of Hex:
  - Players alternately place stones in hexagonal cells
  - First player to connect their two borders with a path of stone of their colour wins
- Implementing Hex in **hexgo/stone_board.py**:
  - Same graph representation as Go:
    Each point gets an index on a (sort of) rectangular grid
  - Borders represented by special "cells" that have negative indices
  - Paths represented by blocks tracked with Union-Find datastructure
    (exactly the same code as Go!)
  - Win condition: Both of a player's borders are in the same block