
RYATTA GROUP
Captain's Mistress Workshop
David Andrews
Ruby Hack Night
Part 1: July 29, 2015, Part 2: August 26, 2015

Captain's Mistress Workshop

During his long sea voyages, Captain Cook would retire to his cabin for extended periods. The crew used to joke that he had a mistress hidden away there. They soon discovered that the Captain had been playing a game with the ship's scientist.

The game came to be known as
'The Captain's Mistress'.

RYATTA GROUP

Rules of Captain's Mistress

1. The game consists of balls and a rack.
2. There are 2 coloured sets of 21 balls each, coloured black and white.
3. The rack has 7 channels (columns) and 6 rows.
4. The rack is oriented vertically so that the balls create 7 stacks.
5. Two players take turns dropping balls into the channels.
6. A ball falls until it lands on top of the existing stack, or the bottom of the rack.
7. Players cannot drop balls into channels that are full.
8. The winner is the first player to create a line of four balls in any direction.

RYATTA GROUP

Do you recognize it?

Masters Traditional Games

RYATTA GROUP

WORLD GAMES

ALL THE FUN IN THE WORLD

Connect Four

VERTICAL CHECKERS GAME

7-Adult

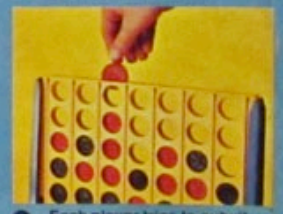
2- Players

A MB Game

Manufactured under licence with Milton Bradley Company U.S.A. by

WORLD GAMES COMPANY LTD. AUCKLAND, N.Z.

1 Players alternate dropping checkers down slots.



2 Each player tries to outwit his opponent.

3 Four in a row wins: across, up-and-down, or diagonally.

Our needs from the very high-level

- The Game and 2 Players:
- The Game entity
  - manages the game board,
  - enforces the rules (including flow of play), and
  - game state (including winning conditions).
- The Player entity
  - answers the question: what is your next move?
- The Player entity will want to ask questions about the game state in order to make decisions about the next move. How do Players ask questions? Through an API!

Divide and conquer

Split up into three teams:

1. Strategy Team 1
2. Strategy Team 2
3. Game Core Team

RYATTA GROUP

Plan for development

Phase 1

- Game Core Team - use TDD to develop the game components
- Strategy Team 1 & 2 - research play strategies

Phase 2

- All - work together to define the API

Phase 3

- Game Core Team - build the API
- Strategy Team 1 & 2 – implement several strategies using the API

Phase 4

- All - pit players against each other and play!

RYATTA GROUP

Useful resources

Game Core Team

- https://www.pivotaltracker.com/n/projects/1396446

Strategy Teams

- http://gizmodo.com/heres-how-to-win-every-time-at-connect-four-1474572099
- https://en.wikipedia.org/wiki/Connect_Four

RYATTA GROUP

Discussion of the solution – the rack

- The team decided the best datatype for storing the rack is an array of channels, each channel storing the ball colour in order from bottom to top
- This datatype has the benefits that Array#push can be used to place new balls, and balls naturally "fall" to their correct locations in the stack. Also, checking the rack for full channels and fullness overall is easy using Array#length
- This datatype has the drawback that it requires manipulation to output. We felt it was preferable to have all of this manipulation in one place rather than spreading checks and tests required by other datatypes across the code.

RYATTA GROUP

Discussion of the solution – printing the rack

- To print the rack we need each row of the rack in order from top to bottom. This requires a bit of manipulation.
- The rack is a compressed (i.e. empty rack spaces are not stored) representation of the channel contents bottom-to-top
- To get rows sorted top-to-bottom we have to do three things:
- 1. expand the rack (i.e. insert the empty rack spaces)
- 2. Array#transpose the contents turning channels into rows
- 3. Reverse the row order so it is top-to-bottom

Discussion of the solution – win detection

- There are four patterns of four balls that need detection: horizontal (in a row), vertical (in a channel), diagonal right (top-left to bottom-right), diagonal left
- The solution uses array manipulation to create four "views" of the rack, each optimal for examining contents in these four orientations
- Balls that cannot participate in a win for a specific orientation are discarded to avoid false positives
- Detection is as simple as finding a continuous string of four balls of the same colour

Discussion of the solution - automatons

- Our computer players, or automatons, were more difficult to program than we had anticipated
- We created one "super easy" automaton that randomly picks an open channel, dubbed "George"
- With a small amount of effort, we should be able to repurpose the win detection code into an API useful for examining the rack for opportunities and threats
- A simple next step would be to build a "three ball" detector and have the automaton chase the opportunity (or block the threat)
- In the meantime, it's fun to play against George, or pit him against himself

RYATTA GROUP

Discussion of the solution – the Game

- The Game entity grew by leaps and bounds near the end of our session
- It has three problems:

1. Lack of focus
2. Many long and complicated methods
3. Insufficient test coverage

RYATTA GROUP

The Rack has two representations

How we store it

1A	2A	3A	4A	5A	6A	7A
1B	2B	3B	4B	5B	6B	7B
1C	2C	3C	4C	5C	6C	7C
1D	2D	3D	4D	5D	6D	7D
1E	2E	3E	4E	5E	6E	7E
1F	2F	3F	4F	5F	6F	7F

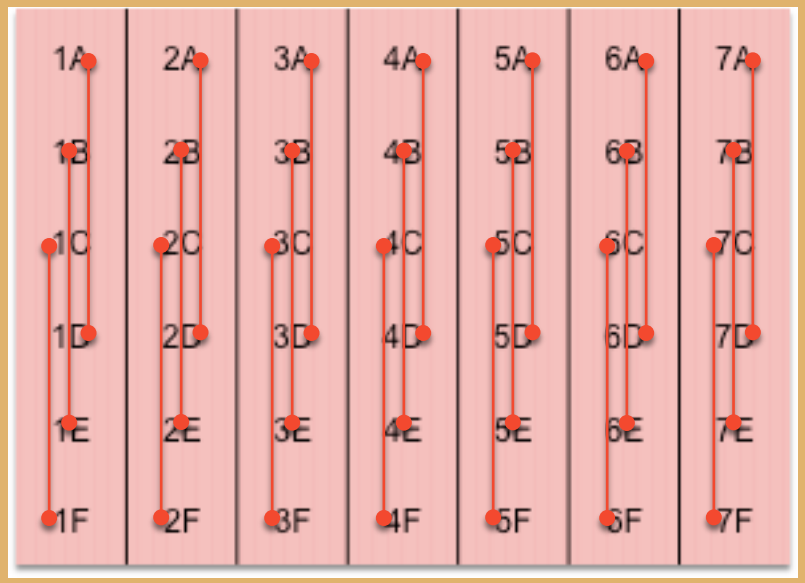
How players see it

1F	1E	1D	1C	1B	1A	
2F	2E	2D	2C	2B	2A	
3F	3E	3D	3C	3B	3A	
4F	4E	4D	4C	4B	4A	
5F	5E	5D	5C	5B	5A	
6F	6E	6D	6C	6B	6A	
7F	7E	7D	7C	7B	7A	

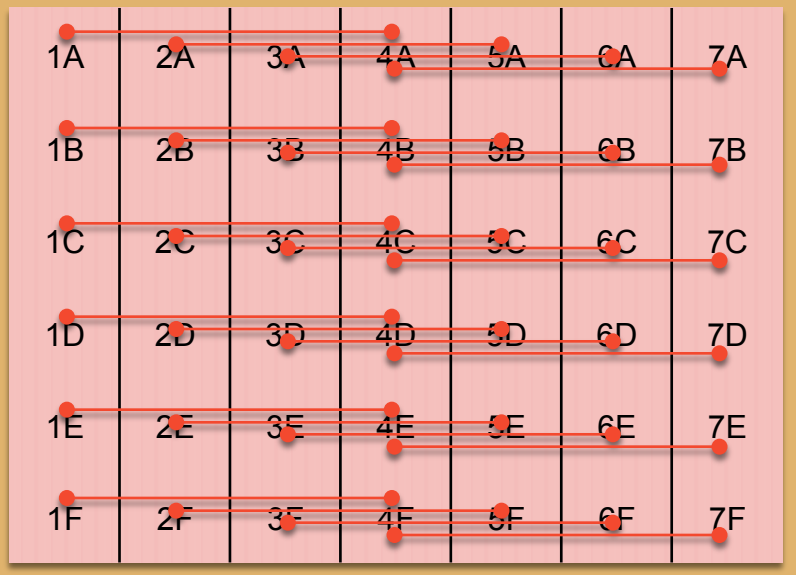
RYATTA GROUP

Enumeration of winning patterns

Vertical winners



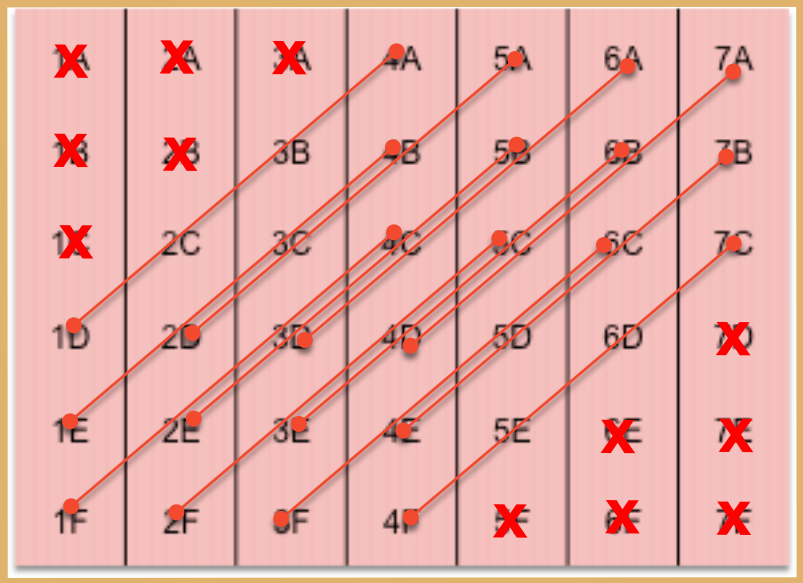
Horizontal winners



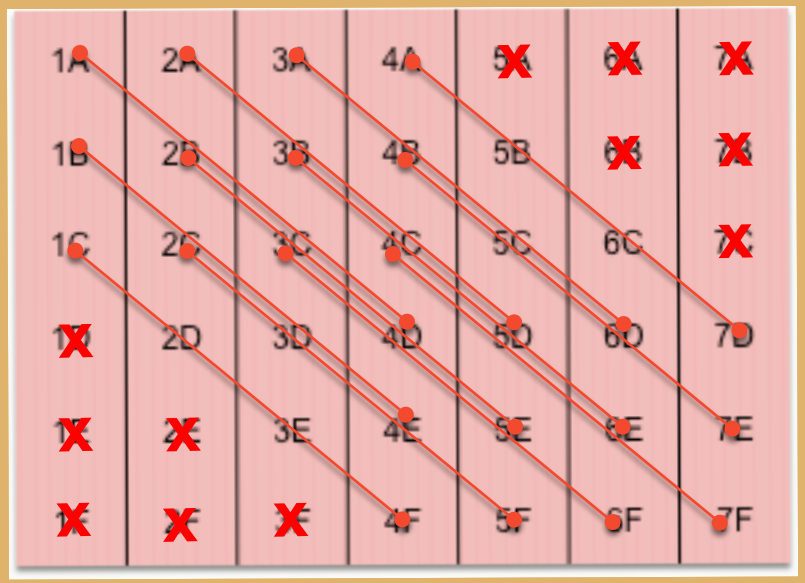
RYATTA GROUP

Enumeration of winning patterns

Top-Right winners



Top-left winners



RYATTA GROUP

Check each winning pattern

Potential winning positions for vertical stacks

1F, 1E, 1D, 1C
1E, 1D, 1C, 1B
1D, 1C, 1B, 1A
2F, 2E, 2D, 2C
2E, 2D, 2C, 2B
2D, 2C, 2B, 2A
3F, 3E, 3D, 3C
3E, 3D, 3C, 3B
3D, 3C, 3B, 3A
4F, 4E, 4D, 4C
4E, 4D, 4C, 4B
4D, 4C, 4B, 4A
5F, 5E, 5D, 5C
5E, 5D, 5C, 5B
5D, 5C, 6B, 5A
6F, 6E, 6D, 6C
6E, 6D, 6C, 6B
6D, 6C, 6B, 6A
7F, 7E, 7D, 7C
7E, 7D, 7C, 7B
7D, 7C, 7B, 7A

RYATTA GROUP

Win detection... ugh!

This approach to win detection is unweildy:

- Ugh-ly (ughful?)
- Hard to read
- Hard to debug
- Hard to validate

Idea!

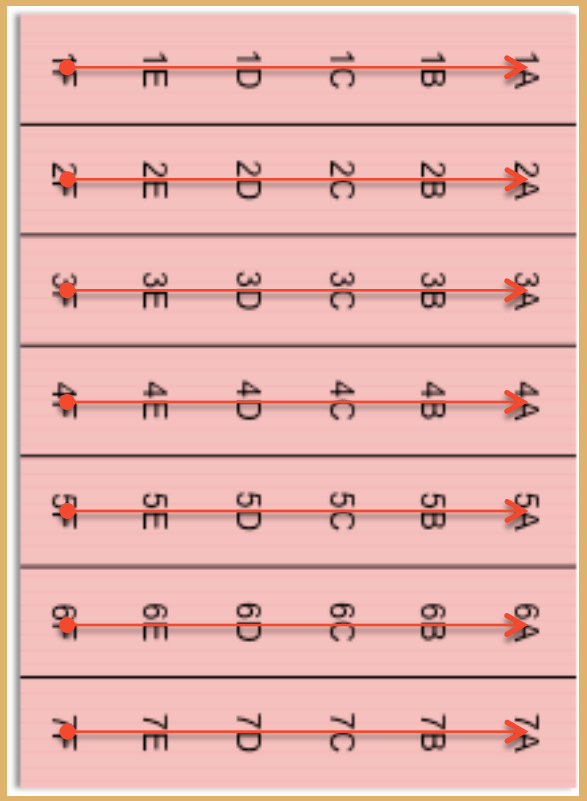
Don't use enumeration as the solution

-> use it as the test!

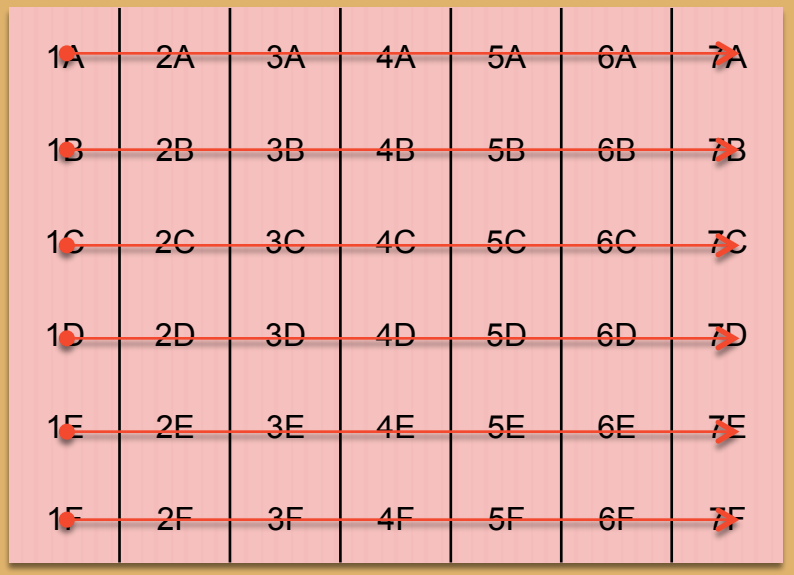
RYATTA GROUP

Rotate to find four continuous symbols

Vertical winners



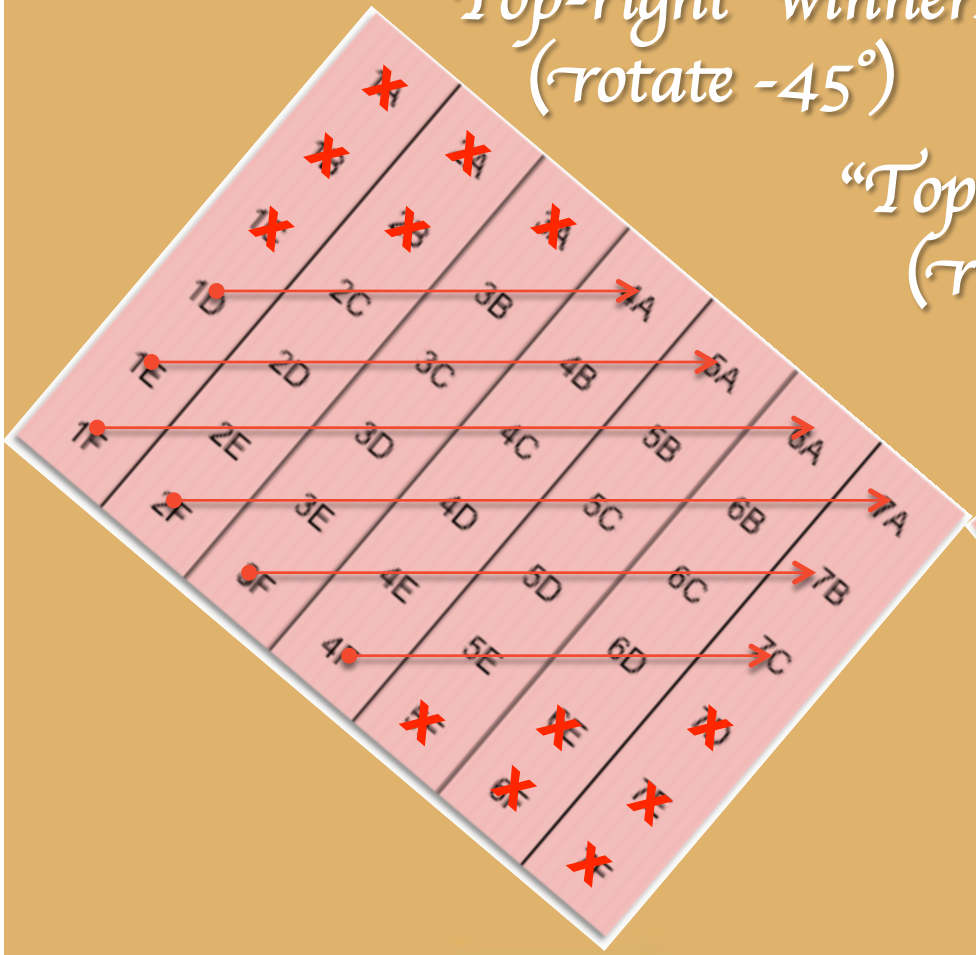
Horizontal winners (rotate -90°)



RYATTA GROUP

Rotation also works for diagonal

"Top-right" winners
(rotate -45°)



"Top-left" winners
(rotate -135°)

RYATTA GROUP

Win detection… beautiful!

- Complexity is in the rotation logic,
  Which is easy to factor out.
- Code to check for winner is same in each case,
  look for a string of four same-type symbols.
- Can be accomplished in 25 checks,
  Versus 69 enumerations.

Idea!

The question "does anyone have a line of four?" is pretty similar to the question "does anyone have a line of three?"

RYATTA GROUP

Approaching the API

We can use this approach to answer the automaton's questions about the state of the rack!

i.e. to answer the question:

Where should I play to progress or block?

Proposed Approach

Given the ability to find arbitrary patterns in the rack we can locate opportinities. What if the solution offered the ability to find spaces as well as tokens?

RYATTA GROUP

Appropriate division of responsibilities

1. #fetch_rack
   - give me the rack so automaton can find the next best move
   - perhaps more correctly named #do_it_all_yourself

2. #find_locations(<pattern>)
   - return locations of tokens and spaces in the rack

3. #find_best_location
   - return the best channel to play
   - perhaps more correctly named #just_play_for_me
   - the strategy for deciding the best next play is unclear

RYATTA GROUP

Defining the API

- Rack#find_locations(<pattern>) – return a list of locations where pattern exists e.g. finding a four-position pattern => [["1A","2B","3C","4B"], ["4F","5F","6F","7F"]]

- What then? How do we use this to help the automaton? Think about it.

Only some locations are "playable"

- Rack#playable_cells - return list of locations that are playable (bottom blank cell in each channel) => ["1A","3B","4C","5C","6F"]
- Crossing this list with the results from #find_locations would provide us with playable spaces within the patterns. Something like:
- find_locations(pattern).flatten.uniq & playable_cells => ["4C"]
- Let's call these "opportunities"

How should we decide between multiple opportunities?

- If we find multiple opportunities, we will want to decide on the "best"
- The main responsibility of any automaton is to decide on how to optimize, both which API calls to make, and how to interpret and analyze the results
- Let's start with the obvious…

RYATTA GROUP

Decision tree for offensive moves

- Goal: Complete a line of four
  - Extend a line of three with a vacant end (3A) xxx- or -xxx
    - Extend a line of two with vacancies on either end (2A) -xx-
      - Place a token with two vacancies on one end and one on the other (1A) --x- or -x--
      - Place a token with three vacancies on one end (1B) ---x or x---
    - Extend a line of two with two vacancies on one end (2B) xx-- or --xx
      - Place a token with two vacancies on one end and one on the other (1A) --x- or -x--
      - Place a token with three vacancies on one end (1B) ---x or x---
    - Extend and fill a line of two with one vacancy in the middle and one on the end (2C) x-x- or -x-x
      - Place a token with two vacancies on one end and one on the other (1A) --x- or -x--
      - Place a token with three vacancies on one end (1B) ---x or x---
  - Fill a line of three with a vacant middle (3B) x-xx or xx-x
    - Extend a line of two with two vacancies on one end (2B) xx-- or --xx
      - Place a token with two vacancies on one end and one on the other (1A) --x- or -x--
      - Place a token with three vacancies on one end (1B) ---x or x---
    - Extend and fill a line of two with one vacancy in the middle and one on the end (2C) x-x- or -x-x
      - Place a token with two vacancies on one end and one on the other (1A) --x- or -x--
      - Place a token with three vacancies on one end (1B) ---x or x---
    - Fill a line of two with two vacancies in the middle (2D) x--x
      - Place a token with two vacancies on one end and one on the other (1A) --x- or -x--
      - Place a token with three vacancies on one end (1B) ---x or x---

RYATTA GROUP

Decision tree for defensive moves

- **Goal: Block a line of four**
  - Extend a line of three with a vacant end (3A) ooo- or -ooo
    - Extend a line of two with vacancies on either end (2A) -oo-
      - Place a token with two vacancies on one end and one on the other (1A) --o- or -o--
      - Place a token with three vacancies on one end (1B) ---o or o---
    - Extend a line of two with two vacancies on one end (2B) oo-- or —oo
      - Place a token with two vacancies on one end and one on the other (1A) --o- or -o--
      - Place a token with three vacancies on one end (1B) ---o or o---
    - Extend and fill a line of two with one vacancy in the middle and one on the end (2C) o-o- or -o-o
      - Place a token with two vacancies on one end and one on the other (1A) --o- or -o--
      - Place a token with three vacancies on one end (1B) ---o or o---
  - Fill a line of three with a vacant middle (3B) o-oo or oo-o
    - Extend a line of two with two vacancies on one end (2B) oo-- or --oo
      - Place a token with two vacancies on one end and one on the other (1A) --o- or -o--
      - Place a token with three vacancies on one end (1B) ---o or o---
    - Extend and fill a line of two with one vacancy in the middle and one on the end (2C) o-o- or -o-o
      - Place a token with two vacancies on one end and one on the other (1A) --o- or -o--
      - Place a token with three vacancies on one end (1B) ---o or o---
    - Fill a line of two with two vacancies in the middle (2D) o--o
      - Place a token with two vacancies on one end and one on the other (1A) --o- or -o--
      - Place a token with three vacancies on one end (1B) ---o or o---

RYATTA GROUP

How should we decide between multiple opportunities?

- Offensive moves should be prioritized above defensive moves at the same "level" i.e. if you win the game by completing your line of four this move, then there is no need to block your opponent from completing their line of four
- Offensive moves in certain channels are better than those made in other channels (generally centre is better)
- Some defensive moves will actually help your opponent win – the link provided earlier in this deck discusses when and how this happens
- Let's discuss some reasonable next steps for building better automatons…

RYATTA GROUP

Proposed agenda

1. Refactor
   - implement tests
   - simplify/beautify the search code
   - split the Game class into Game and Rack
   - have an external board representation
2. Implement the API
   - given a rack, what are the playable places?
   - given a rack, what are all of the locations that contain this pattern?
3. Build automatons

RYATTA GROUP