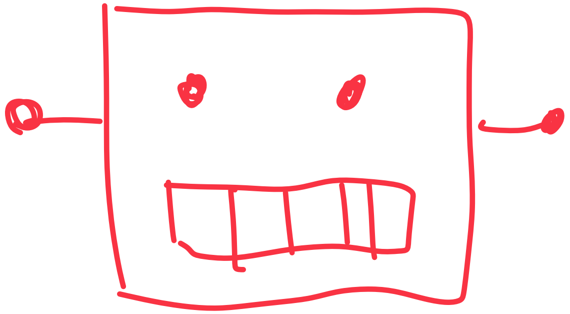
Algorithms and Data Structures for Data Science
lab_ml

CS 277
Brad Solomon

April 5, 2024



UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN
Department of Computer Science

Learning Objectives

Using a graph as a state space

An introduction to reinforcement learning

Honeycomb Havoc

Game of Nim

- Each game starts with k tokens on the table
- Starting with Player 1, players alternate turns:
  - Each turn, a player may pick up 1 or 2 tokens
  - The player who picks up the last token(s) wins

Find a partner and play couple of games!

Or you can play with the computer
https://education.jlab.org/nim/

Solving Nim?

Claim: You can figure out how to play Nim perfectly by looping and remembering things in a dict/array

Coins on the board to start with:	1	2	3	4	>4
If both players play perfectly, the first player always:	Wins!	Wins!	Loses :|	Wins?	There's a pattern ...

How general?

So, depending on the number of tokens, either player 1 or player 2 can *always* win

This can be generalized:

Zermelo's theorem (game theory)

From Wikipedia, the free encyclopedia

For Zermelo's theorem in set theory, see well-ordering theorem.

In game theory, **Zermelo's theorem** is a theorem about finite two-person games of perfect information in which the players move alternately and in which chance does not affect the decision making process. It says that if the game cannot end in a draw, then one of the two players must have a winning strategy (i.e. force a win). An alternate statement is that for a game meeting all of these conditions except the condition that a draw is not possible, then either the first-player can force a win, or the second-player can force a win, or both players can force a draw.[1] The theorem is named after Ernst Zermelo.

Other Perfect Information Games

Nim (**solved**, and you can figure out the general rule)

Tic-Tac-Toe (**solved**, and coding it up is tricky but I bet many of you cant lose)

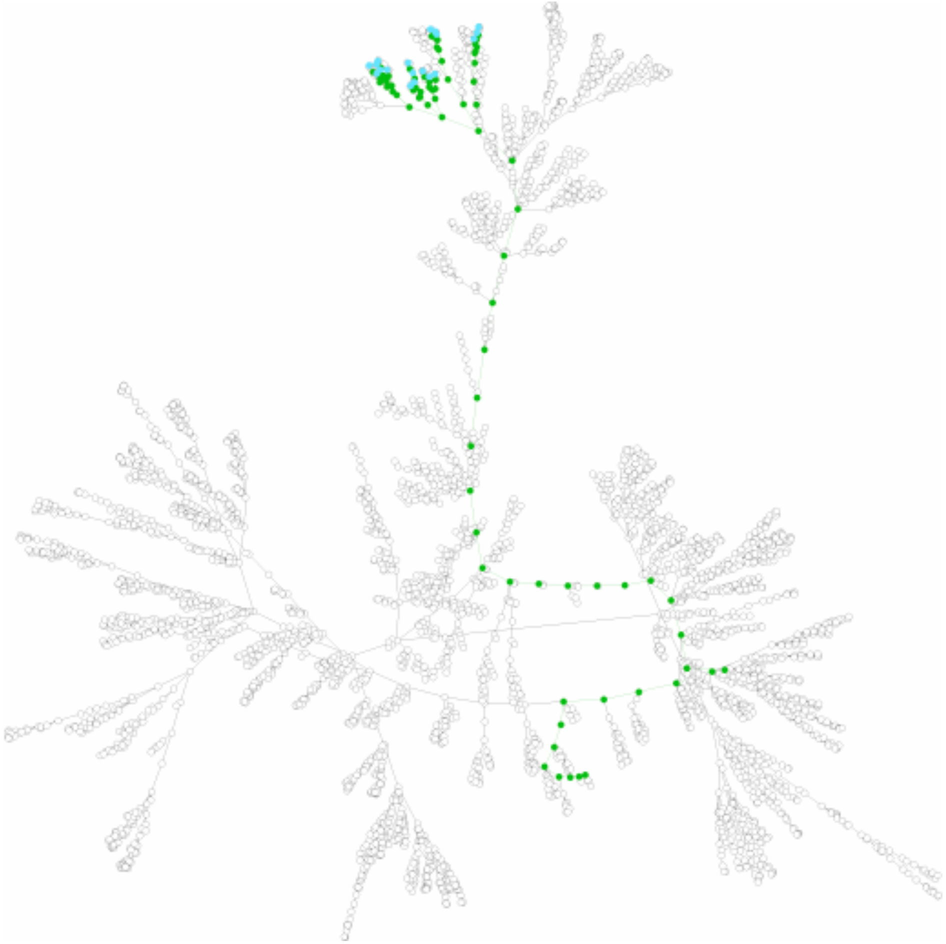
Connect-4 (**solved**, player that starts can always win by dropping a counter in the middle. However there are ~4 trillion configurations)

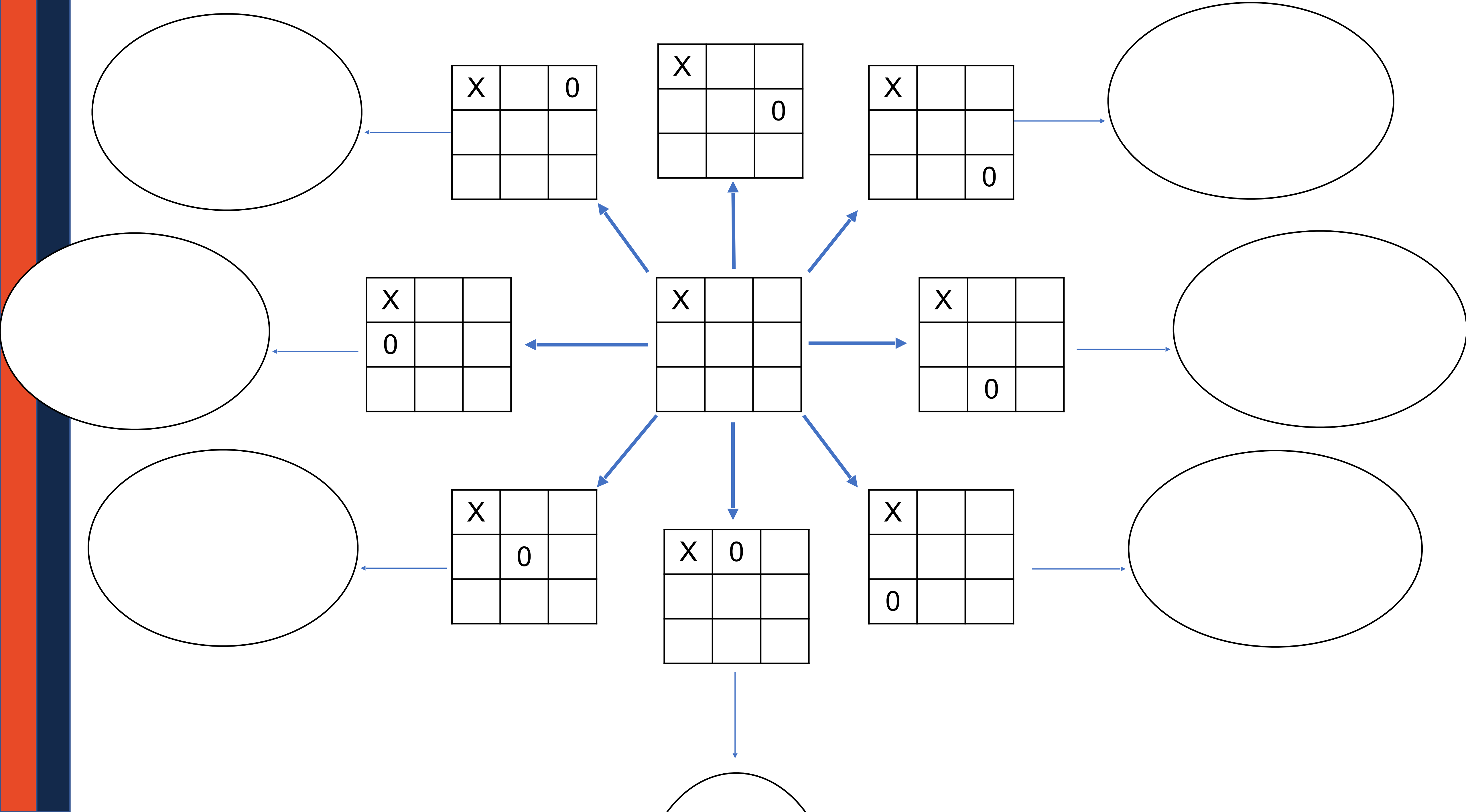
Checkers (**solved** (“weakly”, technical term) - both players can always guarantee a draw)

Chess! (unsolved, more configurations than atoms in the universe …)

State Spaces

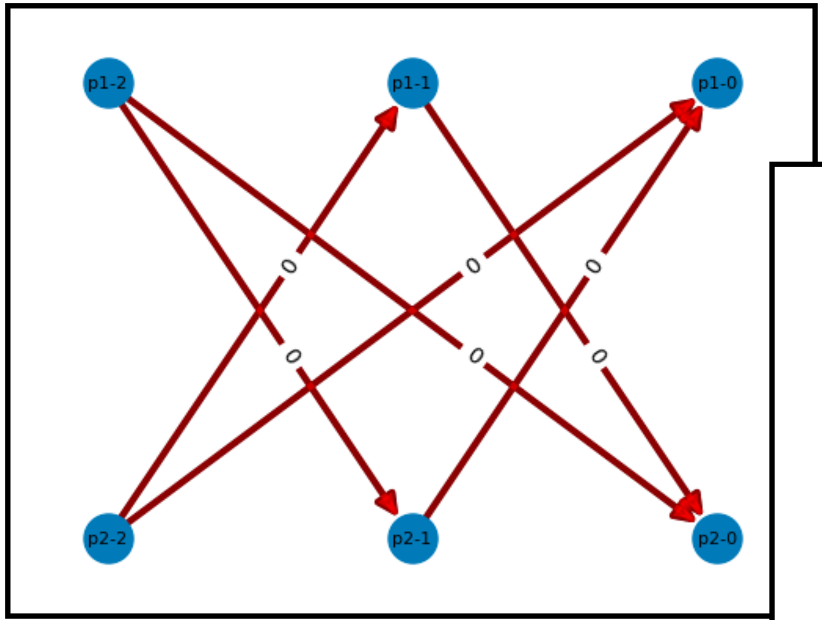
A **state space** is a mathematical representation of the state of a physical system.



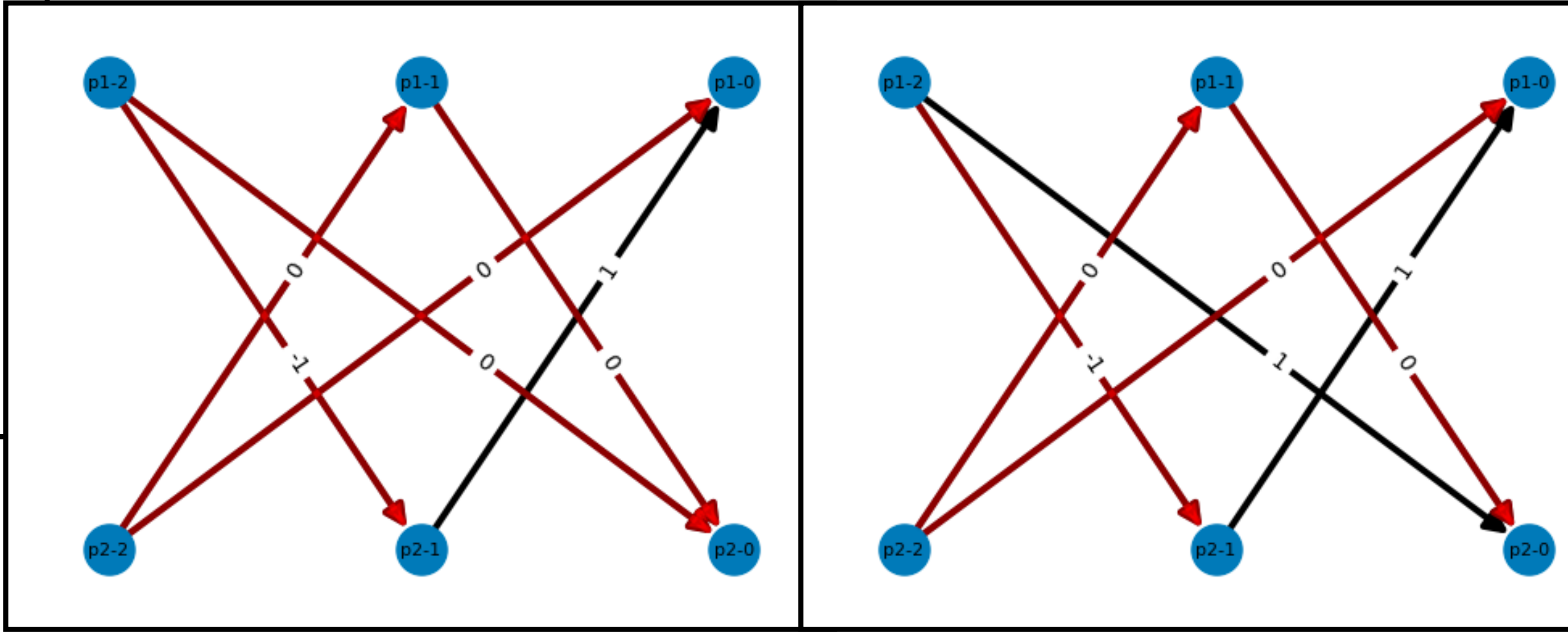

X
X 0
X 0
X 0
X 0
X 0
X 0
X 0
X 0

Nim Reinforcement Learning

1. Build a Nim graph

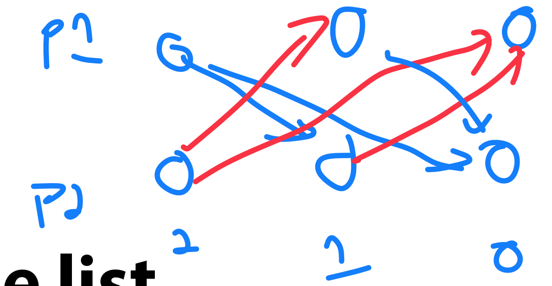


2. Run a random Nim game



3. Update the weights based on the results — and repeat!

make_edge_list()



Given a count of tokens, create a **directed edge list**

1. Each vertex label is a string of the form **`p<#>-<tokens>`**

```
Ex: 'p1-10', 'p2-3'
```

2. Each edge is a list of form **`[start , end]`**

```
Ex: ['p1-10','p2-9']
```

3. Edges are directed and exist only for valid moves.

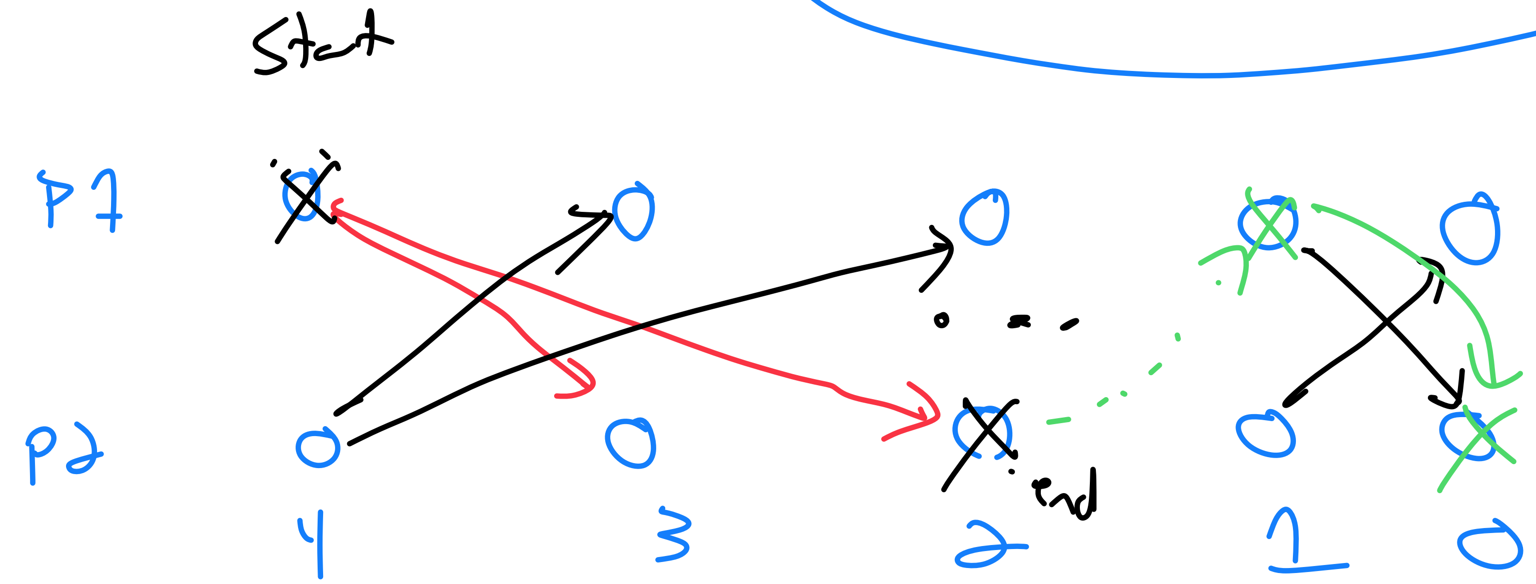
```
['p1-10','p2-7'] valid?  ['p1-10','p1-9'] valid?
```

build_graph()

Given a directed edge list, create a NetworkX graph

1. The graph must be a directed graph!

`G = nx.Graph()` → `G = nx.DiGraph()`



All operations still work — but we now assume edges are one direction.

build_graph()

Given a directed edge list, create a NetworkX graph

2. The graph must be weighted!

```
G = nx.Graph()                G = nx.DiGraph()

        G = nx.add_edge(A, B, weight=5)

        G[A][B]['weight'] # Has value 5
```

All operations still work — but we now assume edges are one direction.

NetworkX Graph ADT

Find

getVertices() —> list(G.nodes())

getEdges(v) —> G[v]

areAdjacent(u, v) —> G.has_edge(u, v)

Insert

insertVertex(v) —> G.add_node(v)

insertEdge(u, v) —> G.add_edge(u, v)

Remove

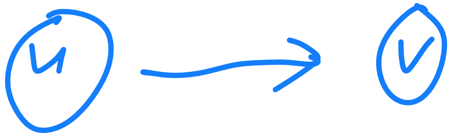
removeVertex(v) —> G.remove_node(v)

removeEdge(u, v) —> G.remove_edge(u, v)

dictionary

list(G[v])

If DiGraph
only start -> end



play_random_game()

G , start

Given a NetworkX graph and a start vertex, return a path through the game

1. You must use random.choice() on the list of adjacent nodes

How can I get a list of keys from a dictionary?

list of options for 'end'

Dictionary.keys()

2. You must save the path as a list of edges

Edges must be of the form [start, end]

update start to be end

update_edge_weights()

Given a path through the Nim graph, update weights for winner / loser

1. Every move made by the winner gets +1 to its edge weight

X += 1

2. Every move made by loser gets -1 to its edge weight

Access a specific edge with: `G[start][stop][‘weight’]`

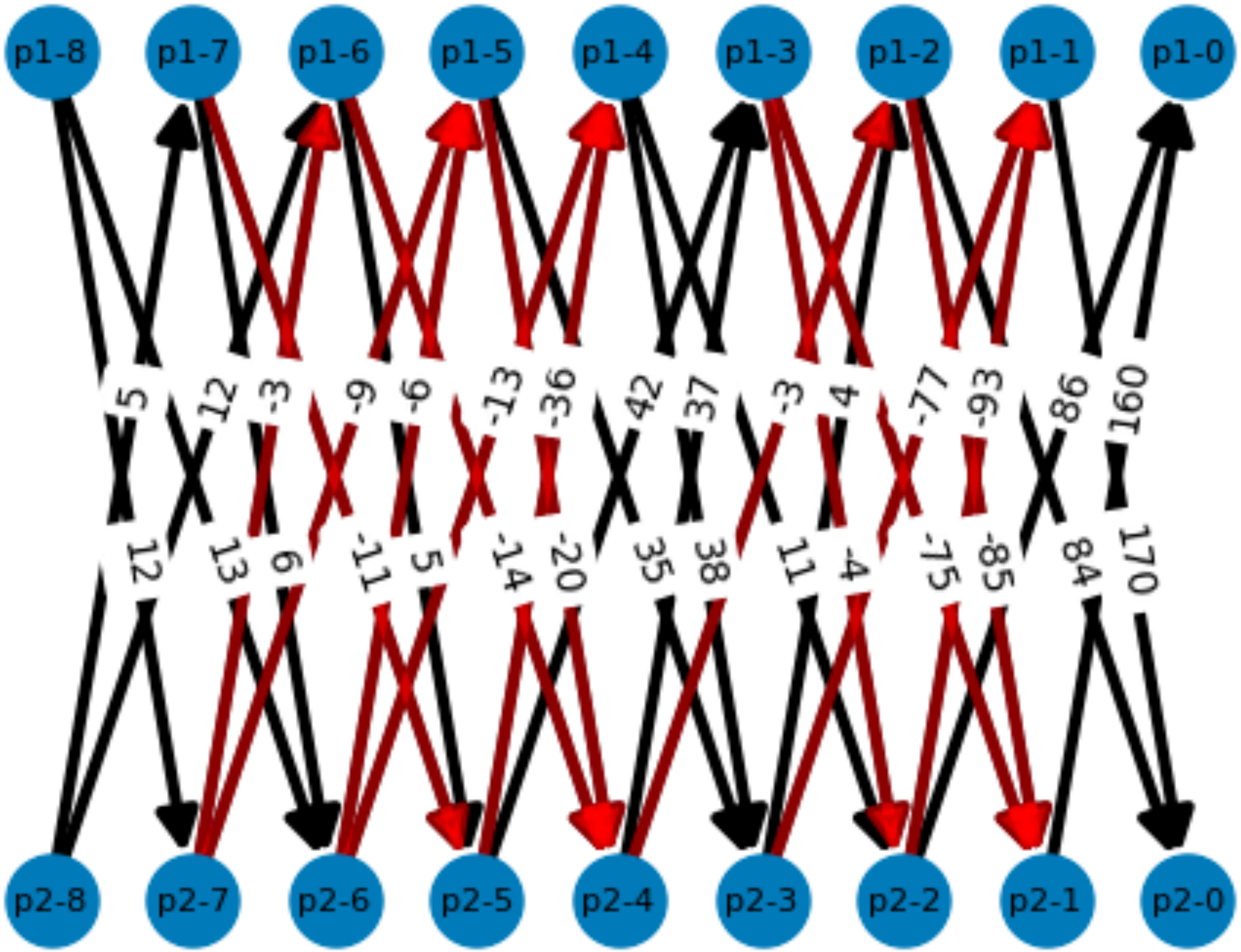
3. How do I know the winner / loser given a path? (Who won:)

```
[('p1-2', 'p2-1'), ('p2-1', 'p1-0')]
```

P2 took last turn

P1 lost b/c its their turn and they have no moves

```
[('p1-2', 'p2-0')]
```


p1-8
p1-7
p1-6
p1-5
p1-4
p1-3
p1-2
p1-1
p1-0
5
12
-3
-9
-6
-13
-36
42
37
-3
4
-77
-93
86
160
12
13
6
-11
5
-14
-20
35
38
11
-4
-75
-85
84
170
p2-8
p2-7
p2-6
p2-5
p2-4
p2-3
p2-2
p2-1
p2-0