Extended Introduction to Computer Science CS1001.py

Lecture 13b: Munch!

Instructors: Benny Chor, Amir Rubinstein

Teaching Assistants: Amir Gilad, Michal Kleinbort

School of Computer Science
Tel-Aviv University

Spring Semester, 2017
http://tau-cs1001-py.wikidot.com

Lecture 12b, Plan

- The game of Munch!
- Two person games and winning strategies.
- A recursive program (in Python, of course).
- A warning regarding running time.
- An existential proof that the first player has a winning strategy.

And Now to Something Completely Different: Munch!

"The time has come," the Walrus said,
"To talk of many things:
Of shoes--and ships--and sealing-wax--
Of cabbages--and kings--
And why the sea is boiling hot--
And whether pigs have wings."

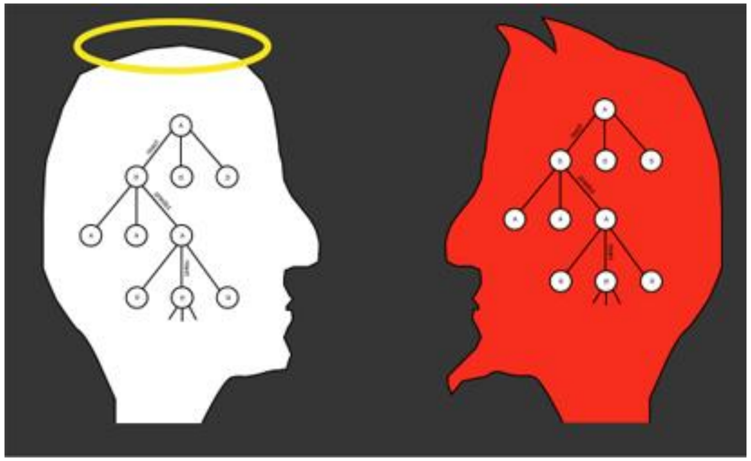
"But wait a bit," the Oysters cried,
"Before we have our chat;
For some of us are out of breath,
And all of us are fat!"
"No hurry!" said the Carpenter.
They thanked him much for that.

Through the Looking-Glass and What Alice Found There: Lewis Carroll, 1871.

Two Player Full Information Games

What is a winning strategy for a player, starting from some configuration of such game?

And what is a winning strategy for a player in the whole game?

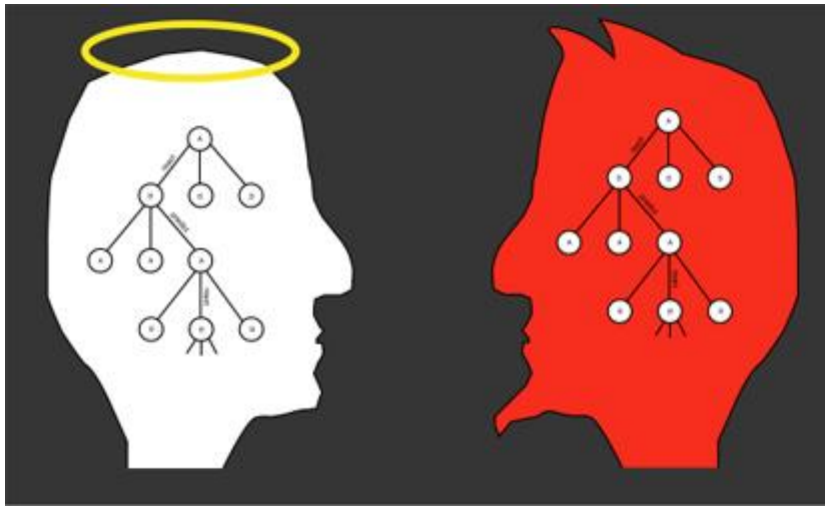


http://about.quickienomics.com/wp-content/uploads/2011/10/game-theory.jpg

Two Player Full Information Games

A theorem from game theory states that in a finite, full information, two player, zero one, deterministic game, either the first player or the second player has a winning strategy.

Unfortunately, finding such winning strategy is often computationally infeasible.

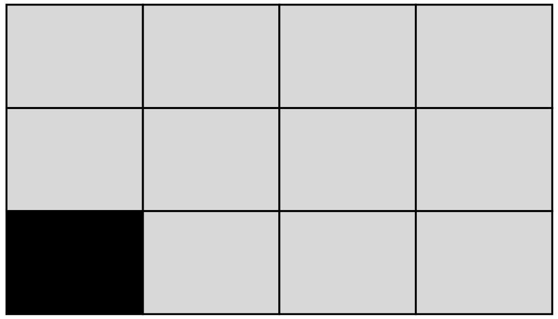


http://about.quickienomics.com/wp-content/uploads/2011/10/game-theory.jpg

Munch!

Munch! is a two player, full information game. The game starts with a chocolate bar with n rows and m columns. Players alternate taking moves, where they chose a chocolate square that was not eaten yet, and munch all existing squares to the right and above the chosen square (including the chosen square).

The game ends when one of the players chooses and munches the lower left square. It so happens that the lower left corner is poisoned, so the player who made that move dies immediately, and consequently loses the game.

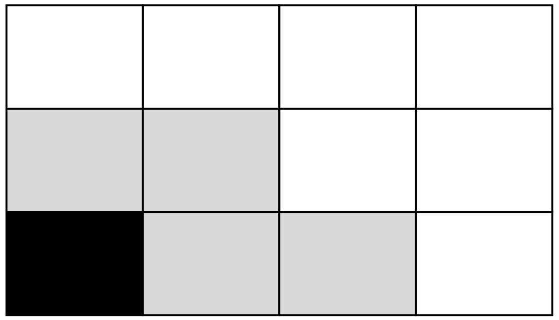


An image of a 3-by-4 chocolate bar (n=3, m=4). This configuration is compactly described by the list of heights [3,3,3,3]

Munch! (example cont.)

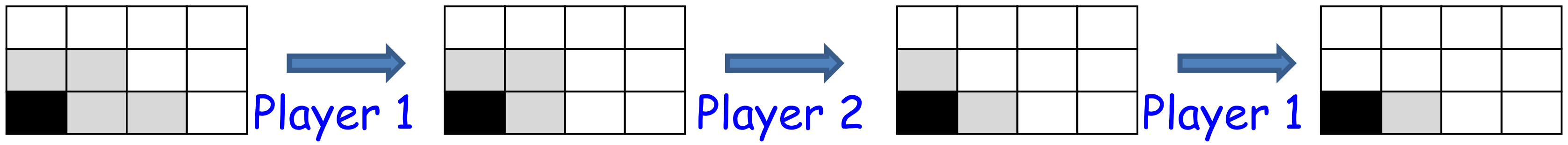
Munch! is a two player, full information game. The game starts with a chocolate bar with n rows and m columns. Players alternate taking moves, where they chose a chocolate square that was not eaten yet, and munch all existing squares to the right and above the chosen square (including the chosen square).

The game ends when one of the players chooses and munches the lower left square. It so happens that the lower left corner is poisoned, so the player who made that move dies immediately, and consequently loses the game.



An image of a possible configurations in the game. The white squares were already eaten. The configuration is described by the list of heights [2,2,1,0].

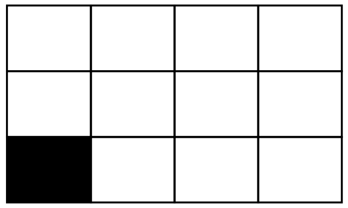
A possible Run of Munch!



[2,2,1,0] [2,2,0,0] [2,1,0,0] [1,1,0,0]

Suppose the game has reached the configuration on the left, [2,2,1,0], and it is now the turn of player 1 to move.
Player 1 munches the rightmost existing square, so the configuration becomes [2,2,0,0].

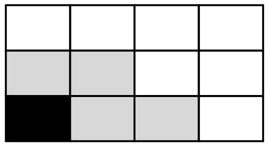
Player 2 munches the top rightmost existing square, so the configuration becomes [2,1,0,0].

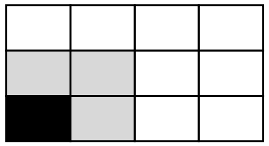

[1,0,0,0]

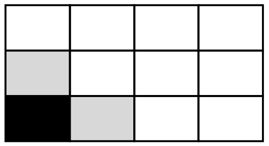
Player 1 move leads to [1,1,0,0].
Player 2 move leads to [1,0,0,0].

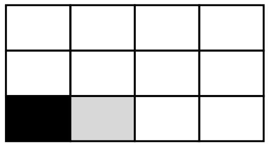
Player 1 must now munch the poisoned lower left corner, and consequently loses the game (in great pain and torment).

Munch!: Winning and Losing Configurations


[2,2,1,0]


[2,2,0,0]


[2,1,0,0]


[1,1,0,0]

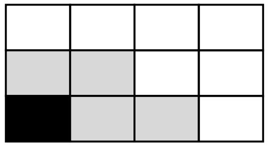
Every configuration has fewer than n times m legal continuing configurations.

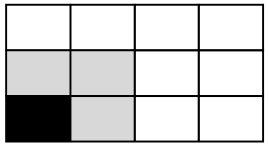
A given configuration is winning if it has some legal losing continuation.

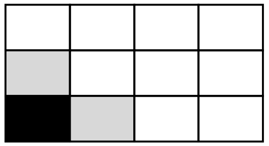
A given configuration is losing if all its legal continuations are winning.

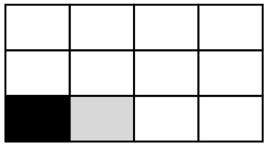
This defines a recursion, whose base case is the winning configuration [0,0,...,0].

Winning and Losing Configurations, cont.


[2,2,1,0]


[2,2,0,0]


[2,1,0,0]


[1,1,0,0]

A given configuration, *C*, is winning if it has some legal losing continuation, *C′*. The player whose turn it is in *C* will choose *C′* for its continuation, putting the opponent in a losing position.

A given configuration, *C*, is losing if all its legal continuations are winning. No matter what the player whose turn it is in *C* will choose, the continuation *C′* puts the opponent in a winning position.

The Initial Munch! Configuration is Winning

We will show (on the board) that the initial configuration [n,n,…,n] of an n-by-m chocolate bar is a winning configuration for all n-by-m size chocolate bars (provided the bar has at least 2 squares).

This implies that player 1 has a winning strategy.

Interestingly, our proof is purely existential. We show such winning strategy exists, but do not have a clue on what it is (e.g. what should player 1 munch so that the second configuration will be a losing one?).

Implementing Munch! in Python

An initial recursive implementation will be able to handle only very small values of n,m (in, say, one minute). It will be discussed in class shortly.

In HW5 you will be asked to write a Python program that will determine if a given configuration is a winning or a losing one, using recursion and memoization. It should able to handle somewhat larger values of n,m in the same amount of time.

A good sanity check for your code is verifying that [n,n,…,n] is indeed a winning configuration. Another sanity check is that in an n-by-n bar, the configuration [n,1,…,1] is a losing configuration.

Munch! Code (recursive)

```python
def win(n, m, hlst, show=False):
    ''' determines if in a given configuration, represented by hlst,
    in an n-by-m board, the player who makes the current move has a
    winning strategy. If show is True and the configuration is a win,
    the chosen new configuration is printed.'''
    assert n>0 and m>0 and min(hlst)>=0 and max(hlst)<=n and \
               len(hlst)==m
    if sum(hlst)==0:  # base case: winning configuration
        return True
    for i in range(m):  # for every column, i
        for j in range(hlst[i]): # for every possible move, (i,j)
            move_hlst = [n]*i+[j]*(m-i)
            # full height up to i, height j onwards
            new_hlst = [min(hlst[i],move_hlst[i]) for i in range(m)]
            # munching
            if not win(n,m,new_hlst):
                if show:
                    print(new_hlst)
                return True
    return False
```

Running the Munch! code

```
>>> win(5,3,[5,5,5],show=True)
[5, 5, 3]
True
>>> win(5,3,[5,5,3],show=True)
False
>>> win(5,3,[5,5,2],show=True)
[5, 3, 2]
True
>>> win(5,3,[5,5,1],show=True)
[2, 2, 1]
True
>>> win(5,5,[5,5,5,5,5],True)
[5, 1, 1, 1, 1]
True
>>> win(6,6,[6,1,1,1,1,1],show=True)
False
```