CSC148 winter 2015

linked structures

week 7

Danny Heap
heap@cs.toronto.edu
BA4270 (behind elevators)
http://www.cdf.toronto.edu/~csc148h/winter/
416-978-5899

February 25, 2015

Outline

Assignment 2

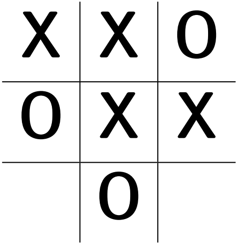
binary trees

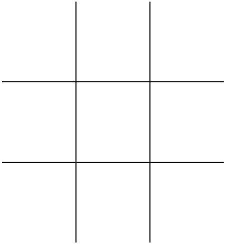
traversals

binary *search* trees

tippy and minimax

This continues the game-playing framework of Assignment 1, adding a new game and a new strategy:



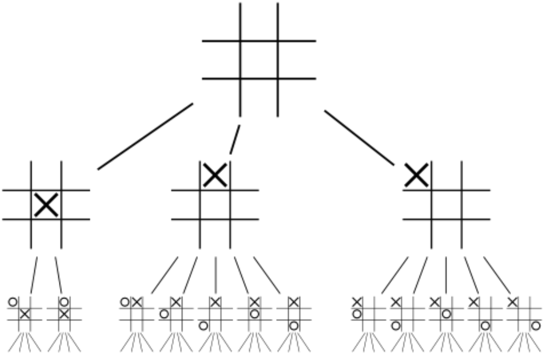


tippy game state

Figure out what a tippy game state needs to be able to represent and do. It helps to peek at both subtract_square_state.py and game_state.py.

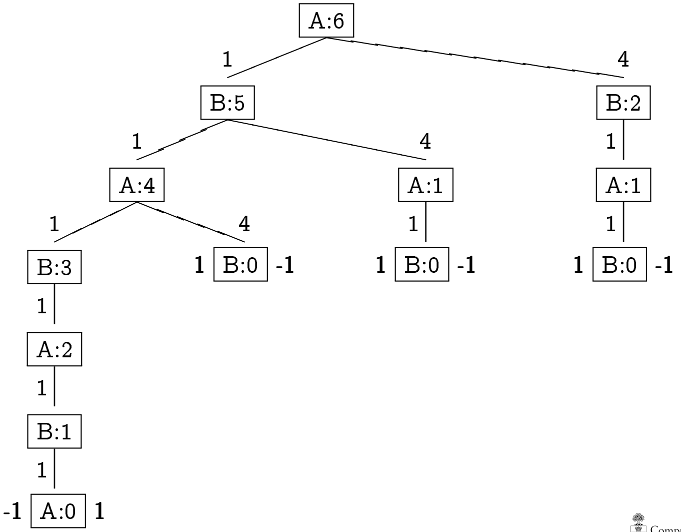
minimax AKA negamax

This is a strong strategy that assumes both players are completely rational and have full information. They look at the consequences of each possible move and select the one with the best outcome for them. A very simplified diagram:



minimax on subtract square

Subtract square doesn't spread out so fast:



what about rough_outcome?

Make an educated guess at the score without looking ahead any moves. In subtract square, this might be: “Can I win this move?” versus “Will I guarantee my opponent can win next move?” versus “Neither of the above”.

```
if is_pos_square(self.current_total):
    return SubtractSquareState.WIN
elif all([is_pos_square(self.current_total - n**2)
          for n in range(1, self.current_total + 1)
          if n**2 < self.current_total]):
    return SubtractSquareState.LOSE
else:
    return SubtractSquareState.DRAW
```

BTNode

Change our generic **Tree** design so that we have two named children, **left** and **right**, and can represent an empty tree with **None**

```
class BTNode:
    '''Binary Tree node.'''

    def __init__(self, data, left=None, right=None):
        ''' (BTNode, object, BTNode, BTNode) -> NoneType

        Create BTNode (self) with data
        and children left and right.
        '''
        self.data, self.left, self.right = data, left, right
```

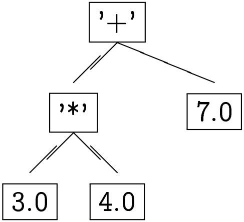
special methods...

We'll want the standard special methods:

- `__eq__`
- `__str__`
- `__repr__`

arithmetic expression trees

Binary arithmetic expressions can be represented as binary trees:



evaluating a binary expression tree

- there are no empty expressions
- if it's a leaf, just return the value
- otherwise...
  - evaluate the left tree
  - evaluate the right tree
  - combine left and right with the binary operator

Python built-in eval might be handy.

inorder

A **recursive** definition:

- visit the left subtree **inorder**
- visit this node itself
- visit the right subtree **inorder**

The code is almost identical to the definition.

preorder

- visit this node itself
- visit the left subtree in **preorder**
- visit the right subtree in **preorder**

postorder

- visit the left subtree in **postorder**
- visit the rightsubtree in **postorder**
- visit this node itself

definition

Add ordering conditions to a binary tree:

- data are comparable
- data in left subtree are less than node.data
- data in right subtree are more than node.data

why binary search trees?

Searchs that are directed along a single path are efficient:

- a BST with 1 one has height 1
- a BST with 3 nodes may have height 2
- a BST with 7 nodes may have height 3
- a BST with 15 nodes may have height 4
- a BST with n nodes may have height $\lceil \lg n \rceil$.

bst_contains

If node is the root of a “balanced” BST, then we can check whether an element is present in about $\lg n$ node accesses.

```
def bst_contains(node, value):
    ''' (BTNode, object) -> value

    Return whether tree rooted at node contains value.

    Assume: node is the root of a BST.

    >>> bst_contains(None, 5)
    False
    >>> bst_contains(BTNode(7, BTNode(5), BTNode(9)), 5)
    True
    '''
    # Use BST property to avoid
    # examining unnecessary nodes.
```