Ruslan Khait

Project 2

Polynominos are shaped figures that represent squares that touch adjacent to other squares. They are also called n-ominos, the n represents the number of squares that creates that shape. These shapes can rotate and reflect itself. You might have seen them in tetris before. But, can we create an algorithm that can show as all the reflects and rotations of any polynomino?

I used ruby to design this algorithm. In the main function, the function create\_poly(n) is called. It reads an integer and calls a different function named rank inside of itself with that value. Rank is a recursion function where it has two base case. If the value is 0 then the function will create an empty array. If the value is 1 then it creates an array inside of an array that can represent a two-dimension grid. It would represent the first square. Else the recursion happens, and a function called new\_function reads the rank function but the parameter value in the rank function is subtracted by one. We do this to see if the next number can work on the base case. This keeps repeating until it meets the base case where n equals to one. The new\_function method has a parameter that would read the array that we got from rank. We loop through a global variable called $answer, that contains an empty array. In the loop we created an i variable that goes thought the array, $answer. We use the method function\_x to see all the possibilities of squares that can touch the square in [0,0]. We look for the square’s neighbors, left, right, up, and down. The neighbors function helps create the square’s neighbors. If will create a neighbor to the left by subtracting one to the x in the (x,y)coordinate For the right, by adding one to the x from the coordinate and add that coordinate For the top neighbor square, it would add the y coordinate by 1. Finally, the bottom neighbor square would be created by subtracting one of the y coordinates. It would do this for all the coordinates that were already inside the array ($answer). All the coordinates are going to be added into $answer. Going back to new\_function method. It will look through all the coordinates in the array, if they aren’t repeat. This would repeat for rank (2), all the way to rank(n).

This then leads us back to the method create\_poly(n) after rank was called. We declared a new array that’s empty that we called answer2. We created an if statement that would check if the length of the shape that we are creating is equal to n. n is the number of squares that are going to be used to create our shape. From all the coordinates we got, the compiler will check if they equal to the length of the shape we want, and if they do, add them into answer2. This helps the computer create the shapes.

This algorithm is an example of backtracking, where we look to see if the computer needs to add an extra square to create a shape that can be a polyomino with that number of squares or not. If it does keep going to see if the next square needs to get added. If not, it ignores this shape and works on the next one. Since we either add a square or not into the shape, this will create 2 in the power of n ways to add squares for the shape to become a polyominos. . So, the time complexity would be big-O of . I understand that the time complexity for this assignment was to create an algorithm that goes O, but the best one I can come up with was this one.

# used ruby

$answer = []

def rank(n)

case n

when 0 then [[]]

when 1 then [[[0,0]]]

else new\_function(rank(n-1))

end

end

def new\_function(answer\_array)

answer\_array.each{|i| function\_x(i) }

$answer.uniq

end

def function\_x(shape)

shape.each{|x,y|

neighbors(x,y).each{|new\_square|

if !shape.include?(new\_square)

$answer << shape + [new\_square]

end

}

}

end

# the neighbors' squares that are touching the square that was choosen

def neighbors(x,y)

[[x - 1, y], [x + 1, y], [x, y - 1], [x, y + 1]]

end

def create\_poly(n)

rank(n)

answer2 = []

$answer.each{|shape|

if shape.length == n

answer2 << shape

end

}

remove\_copy(answer2).sort.each{|poly| puts text\_representation(poly),""}

end

def text\_representation(poly)

table = Hash.new(' ')

poly.each{|x,y| table[[x,y]] = '#'}

maxx = poly.map(&:first).max

maxy = poly.map(&:last).max

minx = poly.map(&:first).min

miny = poly.map(&:last).min

(minx..maxx).map{|x| (miny..maxy).map{|y| table[[x,y]]}.join}

end

def remove\_copy(ans)

answer3 = []

answer4 = []

ans.each{|shape|

shape.each{|x,y| answer4 << x }

}

min\_x = answer4.min

ans.each{|shape|

new\_shape = []

minx2 = shape.map(&:first).min

shape.each{|x,y|

new\_shape << [x - (minx2 - min\_x), y] }

answer3 << new\_shape

}

#y

answer7 = []

answer8 = []

ans.each{|shape|

shape.each{|x,y| answer8 << y }

}

min\_y = answer8.min

answer3.each{|shape|

new\_shape = []

miny2 = shape.map(&:last).min

shape.each{|x,y|

new\_shape << [x , y - (miny2 - min\_y)] }

answer7 << new\_shape

}

answer6 = answer7.map{|shape| shape.sort}

answer6 = answer6.uniq

puts "length #{answer6.length}"

return answer6

end

create\_poly(2)