Tiling Puzzle Solver

By: Darren Bolduc and Sean McElrath

**Overview:**

The algorithm we implemented is essentially a brute force, exhaustive algorithm. It works by trial and error of every possible combination of pieces that could fit with any orientation inside a given board. It is sped up only by eliminating large chunks of possibilities at once.

**Naïve Attempts:**

The first version of our algorithm started by taking any given piece and placing it in all possible spots in which it would fit. Then we would take another piece and fit that into all the remaining spots. And that process would continue. This was able to solve only the trivial board and the two test boards in a reasonable amount of time. For puzzles like the 6x10 pentomino, each of the 12 pieces had roughly 50 places in which it could fit, so we were looking at around 50^12, or roughly 10^20 total combinations.

Next we smartened up (a little bit) and sorted the pieces by which ones would have the least possible available places. This assuredly made the code faster, but still did not yield timely solutions for anything other than the most trivial cases. The next thing we realized is that we could have an 100x100 board and in the very first step, try placing a + piece in the corner, like so:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

If this were the case, our algorithm would attempt to fill the remaining ten thousand squares, despite the fact that no piece would be able to fit in the 1x1 square in the upper left corner.

So we then devised a way to detect whether or not the board was severed into multiple disjoint pieces, the details of which we will discuss in more depth later. We hoped to be able to then solve the disjoint pieces one at a time, which would hopefully lead to either a quick solution, or a quick termination. It was at this point that we realized that our original method of trying to fit a piece in all possible places would not be sufficient, as not every piece would be used in a given disjoint piece of the board, which we refer to in our code as an island. We also realized that although it was not the case that every piece would be used, it was the case that every square in the board must be filled. This led us to our final, current working version of the code.

**The Algorithm:**

Given that every square must be filled in order to yield a solution, we then decided to loop over every square in the board and try every combination of piece and orientation that would cover that square and also fit inside the board. At every point we placed a new piece into the board, we utilized our old code of checking to see whether the board had been split or not. If it had, the next square that we would decide to fill would be picked from the island that had the smallest total size. This way we would quickly be able to determine whether it could be filled by other possible pieces or not. In addition to this we made a minor optimization for cases when all pieces are the same size. This involved making sure that an island was divisible by the size of the pieces, and discarding the possible solution if that were not the case.

**Details:**

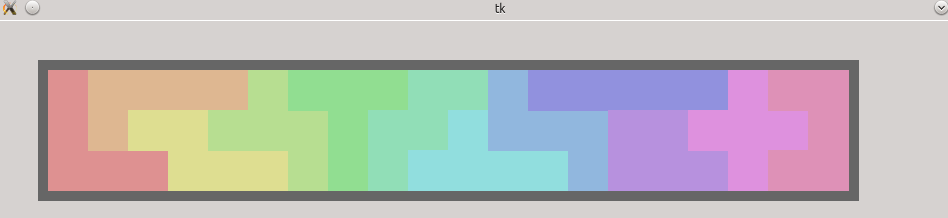
* Pieces were represented as a list of triplets [y location, x location, value], with several other values that could be determined from the list such as rotational/reflectional symmetry and size
* Pieces were constructed recursively, by calling buildPiece(). Given a value in the input file, it was removed and the [y,x,val] triplet was added to the piece being constructed. Then the same process was conducted on every [y2,x2,val2] adjacent to the first one, as long as val2 was not a blank space.
* A nearly identical recursive method was used in the method splits() to determine if a board, which is essentially a giant piece, had been split.
* Symmetries of pieces were checked by trying to place a rotated or reflected piece upon itself. If a piece was invariant to a 90 degree rotation it was given symmetry = 1, if it was invariant under a 180 degree rotation it was given symmetry = 2, otherwise it was given symmetry = 4. If a reflected version of a piece was the same as it or any of its rotations, then the piece was given reflection = 1. Otherwise, reflection = 2.
* The giant loop worked by having a list of triplets called bfirst that was [bag, used, [board]]. Bag was a list of all the unused pieces. Used was a list of all the used pieces, with orientation. The third component was a list of all the disjoint boards.
* In every step of the program the first triplet [bag, used, [board]] was popped off a list. For a given square it was determined every possible piece in the bag that could cover it. Then all of those possibilities were inserted into the very front of the list. Note that a list is probably not the optimal data structure for these queue-like operations, but it certainly has the least overhead and is easiest to work with. We were unsure if we would find a reasonable solution, so we were most concerned with just producing something that would work.

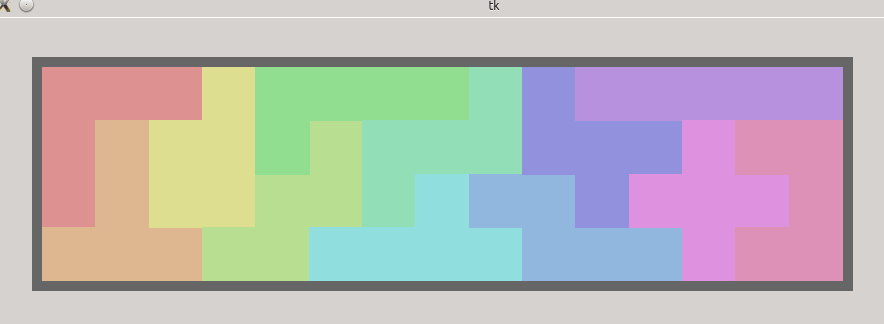
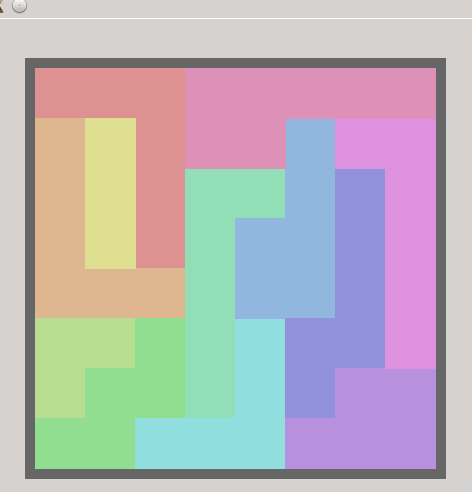
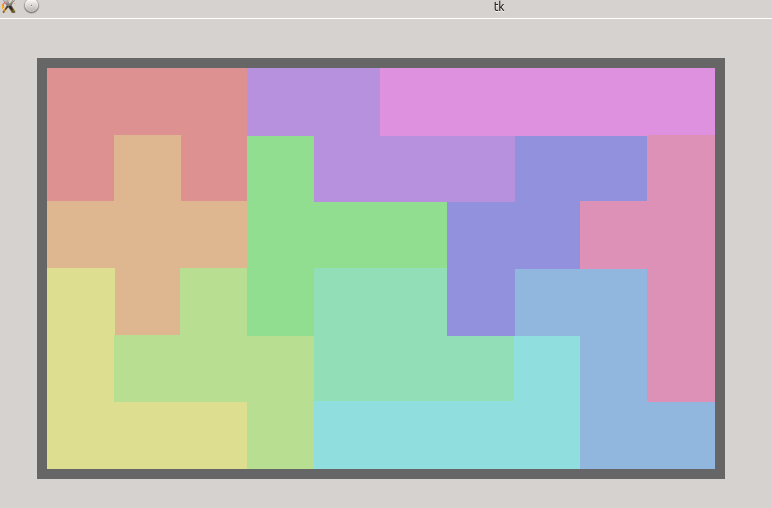
**Performance:**

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Number of Solutions | CPU time for 1 sol. (s) | CPU time for all (s) |
| Trivial | 1 | .000597 | .000597 |
| Pentominoes3x20 | 2 | 1.513 | 2.179 |
| Pentominoes4x15 | 8 | 17.31 | 26.98 |
| Pentominoes6x10 | 106 | 1.277 | 652.6 |
| Checkerboard | 22 | 62.51 | 347.4 |

\*Note: this data was collected without allowing reflections.

**Example Solutions Generated by Code:**





**Raw Code:**

from Tkinter import \*

import ttk

import time

import copy

import random

#txt = open('boards/trivial','r').read().split('\n')

txt = open('boards/trivial2','r').read().split('\n')

txt = open('boards/checkerboard','r').read().split('\n')

#txt = open('boards/IQ\_creator','r').read().split('\n')

#txt = open('boards/lucky13','r').read().split('\n')

txt = open('boards/thirteen\_holes','r').read().split('\n')

#txt = open('boards/test1','r').read().split('\n')

#txt = open('boards/test2','r').read().split('\n')

#txt = open('boards/pentominoes3x20','r').read().split('\n')

#txt = open('boards/pentominoes4x15','r').read().split('\n')

#txt = open('boards/pentominoes5x12','r').read().split('\n')

#txt = open('boards/pentominoes6x10','r').read().split('\n')

#txt = open('boards/pentominoes8x8\_middle\_missing','r').read().split('\n')

#txt = open('boards/new','r').read().split('\n')

class piece:

def \_\_init\_\_(self, char):

self.data = [[0,0,char]]

self.size = 1

self.symmetry = 4 #rotational symmetry. 1, 2, or 4

self.available = []

self.reflect = 1 #assume no reflections. 1 or 2

#draw a piece with a place, orientation, reflection on the tkinter canvas

def drawPiece(sol,color):

a = sol[0]

y = sol[1]

x = sol[2]

ori = sol[3]

ref = sol[4]

mult = 1-2\*ref

for i in a.data:

if ori==0:

rectlist.append(w.create\_rectangle(50+(x+mult\*i[1])\*scale,50+(y+i[0])\*scale,50+(1+x+mult\*i[1])\*scale,50+(1+y+i[0])\*scale,fill=color, outline=color))

if ori==1:

rectlist.append(w.create\_rectangle(50+(x+i[0])\*scale,50+(y-mult\*i[1])\*scale,50+(1+x+i[0])\*scale,50+(1+y-mult\*i[1])\*scale,fill=color, outline=color))

if ori==2:

rectlist.append(w.create\_rectangle(50+(x-mult\*i[1])\*scale,50+(y-i[0])\*scale,50+(1+x-mult\*i[1])\*scale,50+(1+y-i[0])\*scale,fill=color, outline=color))

if ori==3:

rectlist.append(w.create\_rectangle(50+(x-i[0])\*scale,50+(y+mult\*i[1])\*scale,50+(1+x-i[0])\*scale,50+(1+y+mult\*i[1])\*scale,fill=color, outline=color))

#draw empty board

def drawBoard(aboard):

rectlist.append(w.create\_rectangle(40,40,(1+xdiff)\*scale+60,(1+ydiff)\*scale+60,fill="#666666", outline="#666666"))

for i in aboard.data:

rectlist.append(w.create\_rectangle(50+i[1]\*scale,50+i[0]\*scale,50+(1+i[1])\*scale,50+(1+i[0])\*scale,fill="#ccffff", outline="#ccffff"))

#test rotational symmetry of a piece

def testSym(a):

sym = 4

for sq in a.data:

if(trypiece(a,sq[0],sq[1],1,0,a)): # a = i \* a

sym = min(sym,1)

if(trypiece(a,sq[0],sq[1],2,0,a)): # a = -a

sym = min(sym,2)

return sym

#test reflective symmetry of a piece

def testRef(a):

ref = 2

for sq in a.data:

if (trypiece(a,sq[0],sq[1],0,1,a)):

ref = 1

if (trypiece(a,sq[0],sq[1],1,1,a)):

ref = 1

if (trypiece(a,sq[0],sq[1],2,1,a)):

ref = 1

if (trypiece(a,sq[0],sq[1],3,1,a)):

ref = 1

return ref

#take a board and check to see if it is disjoint

def splits(aboard):

seen = []

islands = []

for i in aboard.data:

if i not in seen:

a = piece(i[2])

a.data = [i]

yoff = i[0]

xoff = i[1]

seen.append(i)

buildIsland(aboard,a,yoff,xoff,0,0,seen)

islands.append(a)

return islands

#called by splits(). return all the blocks connected to a given piece

def buildIsland(aboard,a,y,x,dy,dx,seen):

#print "called2: " + str(dy) + ", " + str(dx)

for i in aboard.data:

if i not in seen:

if i[0]==y+dy and i[1]==x+dx+1:

a.data.append(i)

a.size += 1

seen.append(i)

buildIsland(aboard,a,y,x,dy,dx+1,seen)

if i[0]==y+dy and i[1]==x+dx-1:

a.data.append(i)

a.size += 1

seen.append(i)

buildIsland(aboard,a,y,x,dy,dx-1,seen)

if i[0]==y+dy+1 and i[1]==x+dx:

a.data.append(i)

a.size += 1

seen.append(i)

buildIsland(aboard,a,y,x,dy+1,dx,seen)

if i[0]==y+dy-1 and i[1]==x+dx:

a.data.append(i)

a.size += 1

seen.append(i)

buildIsland(aboard,a,y,x,dy-1,dx,seen)

#construct a piece from the original file text

def buildPiece(a,y,x,dy,dx):

layout[y] = layout[y][:x] + " " + layout[y][x+1:]

if layout[y-1][x]!=" ":

a.data.append([dy-1,dx,layout[y-1][x]])

a.size = a.size+1

buildPiece(a,y-1,x,dy-1,dx)

if layout[y+1][x]!=" ":

a.data.append([dy+1,dx,layout[y+1][x]])

a.size = a.size+1

buildPiece(a,y+1,x,dy+1,dx)

if layout[y][x-1]!=" ":

a.data.append([dy,dx-1,layout[y][x-1]])

a.size = a.size+1

buildPiece(a,y,x-1,dy,dx-1)

if layout[y][x+1]!=" ":

a.data.append([dy,dx+1,layout[y][x+1]])

a.size = a.size+1

buildPiece(a,y,x+1,dy,dx+1)

#make sure there are enough pieces to fill up the entire board

def sizeCheck():

totalsize = 0

for i in bag:

totalsize += i.size

if totalsize >= boardsize:

print "VALID PUZZLE (SO FAR)"

else:

print "STOP RIGHT THERE. NOT ENOUGH PIECES TO COVER BOARD"

#attempt to place piece a in aboard with location dy,dx with orientation and reflection

#returns boolean value

def trypiece(a,dy,dx,orientation,reflection,aboard):

flag = 1

for i in a.data:

mult = 1-2\*reflection #0 -> 1, 1 -> -1

if orientation==0: #as is

if [i[0]+dy,mult\*i[1]+dx,i[2]] not in aboard.data:

flag = 0

break

if orientation==1: #90 degrees counterclockwise

if [-mult\*i[1]+dy,i[0]+dx,i[2]] not in aboard.data:

flag = 0

break

if orientation==2: #180 degree rotation

if [-i[0]+dy,-mult\*i[1]+dx,i[2]] not in aboard.data:

flag = 0

break

if orientation==3: #270 degree counterclockwise

if [mult\*i[1]+dy,-i[0]+dx,i[2]] not in aboard.data:

flag = 0

break

return flag

#one step in solving the puzzle. take the first available square and try all the pieces in it

def boardfirstSolve():

if len(bfirst)==0:

print "Hmmmm..."

return

curstate = bfirst.pop(0)

abag = curstate[0]

aused = curstate[1]

aboards = curstate[2]

if len(aboards)==0:

print "we did it in " + str(time.time()-t) + " seconds"

solutions2.append(aused)

return

aboards.sort(key=lambda x:x.size) #start with smallest remaining board

aboard = aboards[0]

if (samesize): #we can eliminate possibilities all pieces are same size, and that size does not divide the size of an island

if aboard.size%samesize != 0:

return

sq = aboard.data[0]

for config in piecefit(aboard,abag,sq[0],sq[1]):

#print config[0].size, config[1], config[2]

a = config[0]

center = config[1]

ori = config[2]

ref = config[3]

tempboards = aboards[:]

tempboards[0] = changeBoard2(a,sq[0],sq[1],center,ori,ref,aboard)

if tempboards[0].size==0:

tempboards.pop(0)

else:

for i in splits(tempboards.pop(0)):

tempboards.append(i)

tempused = aused[:]

if ori==0:

newy = sq[0]-center[0]

newx = sq[1]-center[1]

if ori==1:

newy = sq[0]+center[1]

newx = sq[1]-center[0]

if ori==2:

newy = sq[0]+center[0]

newx = sq[1]+center[1]

if ori==3:

newy = sq[0]-center[1]

newx = sq[1]+center[0]

tempused.append([a,newy,newx,ori,ref])

#if ref==1:

# drawPiece([a,newy,newx,ori,ref],"#00ff00")

# print a.data

# break

tempbag = abag[:]

tempbag.remove(a)

bfirst.insert(0,[tempbag,tempused,tempboards])

#return all of the pieces that can fit a given square

def piecefit(aboard,abag,y,x):

configs = []

for a in abag:

for i in a.data:

for ori in range(a.symmetry):

for ref in range(a.reflect):

if (trypiece2(a,y,x,i,ori,ref,aboard)):

configs.append([a,i,ori,ref])

return configs

#more general version of trypiece that adjusts for a board that doesnt contain 0,0

def trypiece2(a,y,x,center,ori,ref,aboard):

if ori==0:

return trypiece(a,y-center[0],x-center[1],ori,ref,aboard)

if ori==1:

return trypiece(a,y+center[1],x-center[0],ori,ref,aboard)

if ori==2:

return trypiece(a,y+center[0],x+center[1],ori,ref,aboard)

if ori==3:

return trypiece(a,y-center[1],x+center[0],ori,ref,aboard)

#remove a piece a from the board

def changeBoard2(a,dy,dx,center,orientation,reflection,bboard):

## only call this if its been tested with trypiece first!!!

cboard = copy.deepcopy(bboard)

for i in a.data:

mult = 1 - 2\*reflection

cboard.size -= 1

if orientation==0: #as is

cboard.data.remove([i[0]-center[0]+dy,mult\*i[1]-center[1]+dx,i[2]])

if orientation==1: #90 degrees counterclockwise

cboard.data.remove([-mult\*i[1]+center[1]+dy,i[0]-center[0]+dx,i[2]])

if orientation==2: #180 degree rotation

cboard.data.remove([-i[0]+center[0]+dy,-mult\*i[1]+center[1]+dx,i[2]])

if orientation==3: #270 degree counterclockwise

cboard.data.remove([mult\*i[1]-center[1]+dy,-i[0]+center[0]+dx,i[2]])

return cboard

#################################################################################################

master = Tk()

h = 500

w = Canvas(master,width=2\*h,height=h)

w.pack()

rectlist = []

allowReflections = 0

#reading input

m = 0

for i in txt:

m = max(m,len(i))

layout = []

bag = []

used = []

#buffer around input file

height = 2

blankline = " "

for i in range(m):

blankline+=" "

layout.append(blankline)

for i in txt:

s = " "

for j in range(m):

try:

s+=i[j]

except IndexError:

s+=" "

s+=" "

layout.append(s)

height+=1

layout.append(blankline)

boardsize = 0

for i in range(height): #row

for j in range(m+2):

if (layout[i][j]!=" "):

a = piece(layout[i][j])

buildPiece(a,i,j,0,0)

a.symmetry = testSym(a)

if (allowReflections):

a.reflect = testRef(a)

bag.append(a)

if a.size > boardsize:

boardsize = a.size

#remove the board from bag

for i in bag:

if i.size == boardsize:

board = i

bag.remove(i)

break

for i in bag: #fix one piece with no symmetries if the board is symmetrical/semisymmetrical

if i.reflect==2 and i.symmetry==4:

i.symmetry = board.symmetry

i.reflect = board.reflect

break

if (allowReflections==0) and i.symmetry==4:

i.symmetry = board.symmetry

break

samesize = bag[0].size

for i in bag:

if i.size != samesize:

samesize = 0

break

sizeCheck()

solutions2 = []

bfirst = [[bag,used,[board]]]

#the following stuff is for the GUI

xlow = 0

xhigh= 0

ylow = 0

yhigh= 0

for i in board.data:

if i[0] < ylow:

ylow = i[0]

if i[0] > yhigh:

yhigh = i[0]

if i[1] < xlow:

xlow = i[1]

if i[1] > xhigh:

xhigh = i[1]

xdiff = xhigh - xlow

ydiff = yhigh - ylow

scale = min(400.0/(1+ydiff), 800.0/(1+xdiff))

drawBoard(board)

colors = []

colors.append("#de9191")

colors.append("#deb791")

colors.append("#dede91")

colors.append("#b7de91")

colors.append("#91de91")

colors.append("#91deb7")

colors.append("#91dede")

colors.append("#91b7de")

colors.append("#9191de")

colors.append("#b791de")

colors.append("#de91de")

colors.append("#de91b7")

#those last line are worth it, because these are some nice looking colors

ccount = 0

for i in range(10,len(bag)): #sloppy. could say 12, len(bag)

color = "#%06x" % random.randint(0,0xffffff)

colors.append(color)

#drawPiece([bag[0],1,1,1,0],colors[0])

#drawPiece([bag[0],4,1,3,1],colors[1])

#print trypiece(bag[0],1,1,1,0,board)

#print trypiece(bag[0],1,1,3,1,board)

t=time.time()

#this is where the puzzle actually gets solved

while len(bfirst)>0:

boardfirstSolve()

if len(solutions2) == 0:

print "NO SOLUTIONS PAL"

for j in range(len(solutions2)):

print "SOLUTION #" + str(j+1) + ":"

for i in solutions2[j]:

print "piece: " + str(i[0].data) + ", location: (" + str(i[1]) + ", " + str(i[2]) + "), orientation: " + str(i[3]) + ", reflection: " + str(i[4])

print "================\n"

print "total time solving: " + str(time.time()-t)

if len(solutions2)>0:

for i in solutions2[0]:

drawPiece(i,colors[ccount])

ccount += 1

mainloop()