

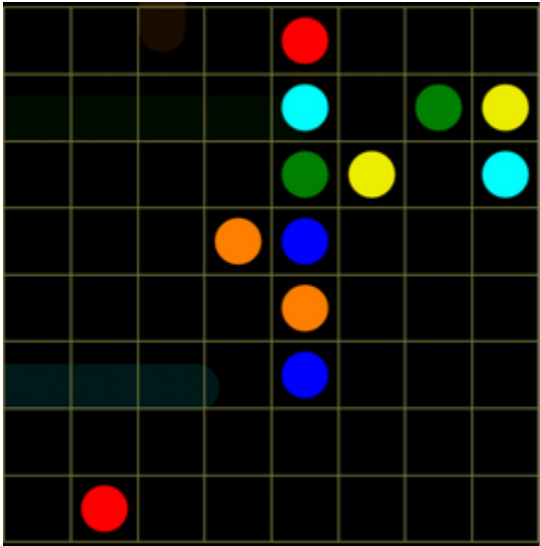
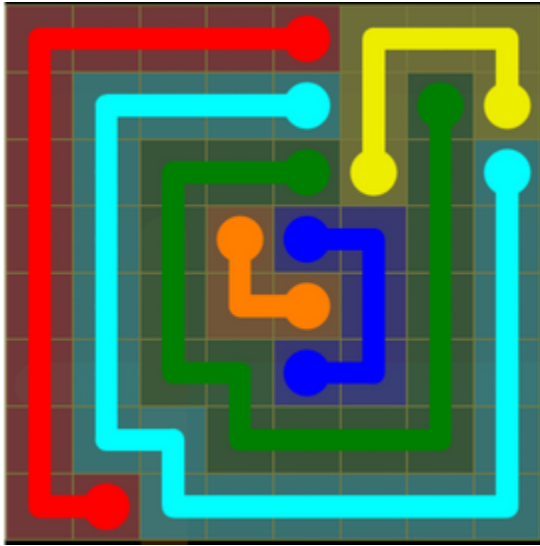
Flow Free Solver

Deji Oyerinde(oko2107) and Kidus Mulu(km3533)

The Problem

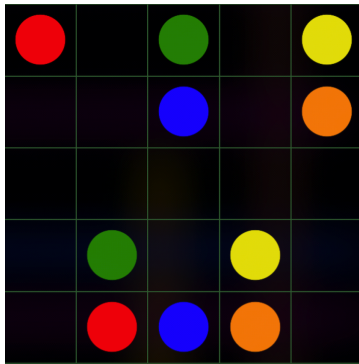
Flow free is a video game for mobile that is based on the [Number Link](#) puzzle. To begin, there is a grid with pairs of different colors in different boxes. The objective of the game is to connect all the pairs to each other in flows. A flow is an uninterrupted path from one colored dot to the other of the same color. A flow can move in any of the four directions within the grid but it cannot cross another flow and it cannot go into a box occupied by another dot of a different color.

For example, one starting grid is shown in the following table along with a solution:

Start grid	Solution
	

Our program, flow-solver, takes the initial grid in the form of a text file and produces a solution to the flow-free game represented by that text file.

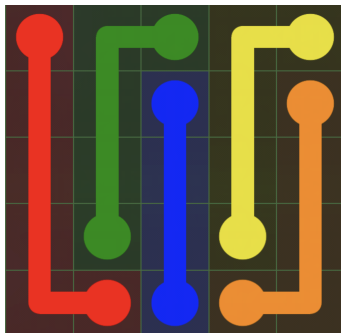
Problem representation



```
> cat regular 5x5 01.txt  
r0g0y  
00b0o  
00000  
0g0y0  
0rbo0
```

We represented the problem board as a text file with a matrix of lower case letters and zeros. Each empty cell would be represented by a '0' and each color dot would be represented by a lower-case letter.

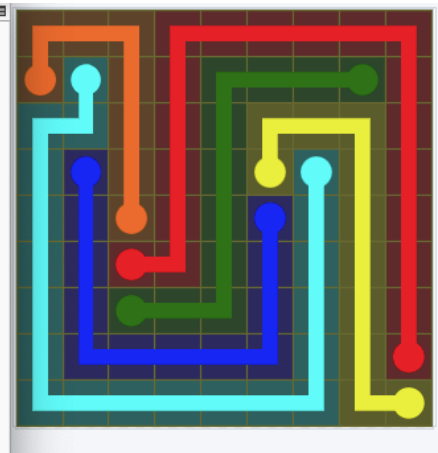
In the solved board, the flows are represented by a continuous flow of uppercase letters:



```
> stack run puzzles/regular 5x5 01.txt  
RGGYY  
RGBYO  
RGBYO  
RGBYO  
RRB00
```

Another run of the algorithm on a 9x9 board.

```
> stack run puzzles/extreme 9x9 30.txt par  
00ORRRRRR  
OCORGGGGR  
CCORGYYR  
CBORGICYR  
CBORGBCYR  
CBRRGBCYR  
CBGGGBCYR  
CBBBBBCYR  
CCCCCCYY
```



Other Possible Solutions

Before we produced our solution, there were multiple approaches that we considered for solving the game. The first was to use an A* graph search algorithm. Given a graph and start node, The A* algorithm determines the next possible node to search by trying to minimize a function $g(n) + h(n)$, where $g(n)$ is the cost to get to node n and $h(n)$ is the heuristic function used to determine the best possible selection of the next node. In the case of Flow Free, you could determine $h(n)$ to be the number of remaining unoccupied cells on the board and each possible state would be a possible node of the graph. We decided not to use A* search even though it is asymptotically faster than our solution because it did not have an embarrassingly parallel aspect to it.

The second approach we considered was to use a satisfiability solver (SAT solver) by representing the Flow Free starting board as a Conjunctive Normal Form (CNF) formula. After this conversion, we would input that formula into a SAT solver and if solvable, receive a solution that satisfies all constraints. However, we decided against this approach after learning that it would require 1000+ clauses and 100+ variables just to represent the simplest 5x5 board.

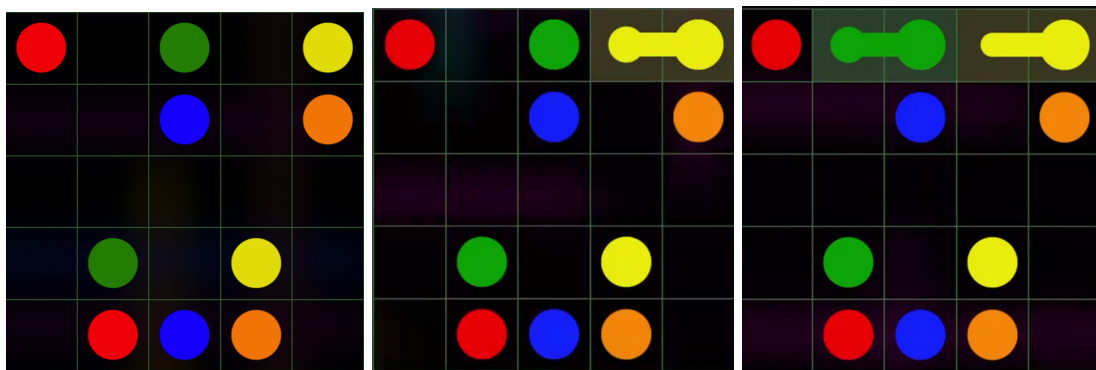
Our Solution

Ultimately, we decided to use a **heuristic based Best First Search algorithm** to solve the board. The heuristic used is to move the color (or dot) that has the smallest possible moves at each iteration.

The algorithm generally works as follows:

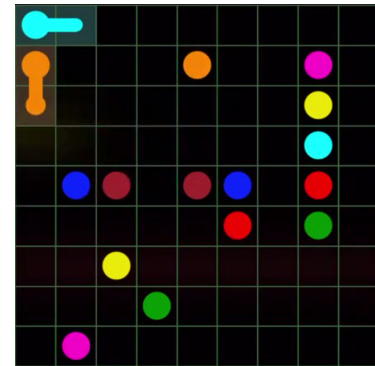
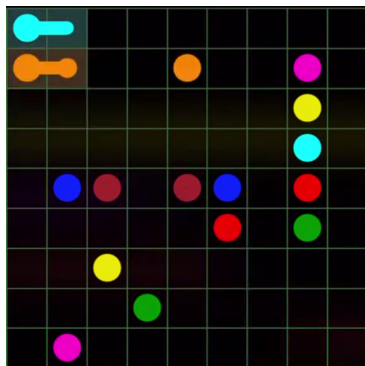
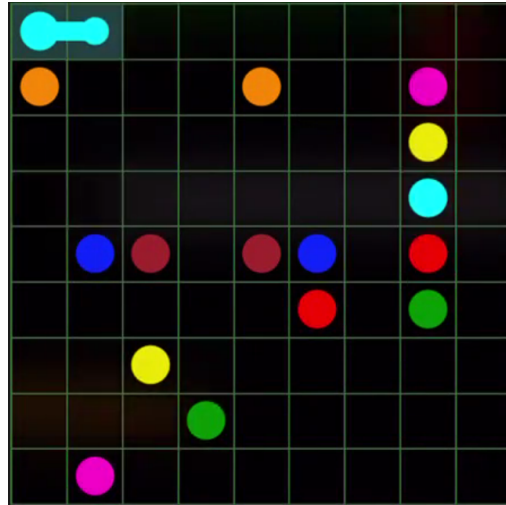
1. On the problem board, find the color with the smallest number of possible moves
2. Make produce a board with each of those possible moves and solve the generated boards recursively

As an example, consider the following progressions of the algorithm:



In the first board, there are a few dots with only one move to make. We choose the yellow dot in the top right corner and produce the board in the middle. The algorithm then goes again and scans the board, finding that the green at top-center only has one move to make. It makes that move and continues like that until the entire board is solved.

In the 5x5 example above, at each iteration there will always be a color with only one move to make, so the algorithm proceeds fairly sequentially. However, consider the following board after the first move:



In the top board, each dot has at least two valid moves. We select the orange in the top right with only two possible moves as our next color to advance. Then, we produce two children boards as shown by advancing the orange dot in the two possible ways. After producing the two sub-problems, we call the solving routine recursively on both. The algorithm continues like this, at each step checking if the board is solved or not. Once it discovers that a board is solved, it returns the solution to the caller.

Parallel Strategies

Each call has to look at its board to figure out if it is solved and, if it isn't, figure out which subproblems it needs to solve. This step in the algorithm is not easily parallelizable since the tasks are linked in ways that would make a parallel algorithm too complicated. After these checks, though, a call has to solve its children recursively. Since the children are completely independent problems, the calls are embarrassingly parallel and can each be computed simultaneously.

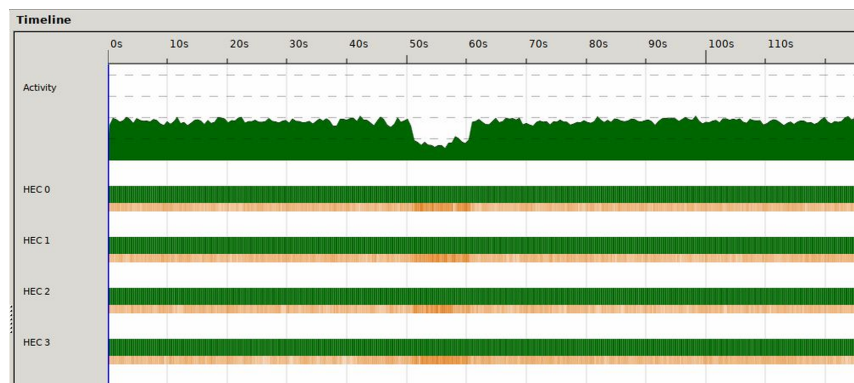
We explored 3 strategies of how a parent call can evaluate its children as we discuss below.

The Naive Parent

The first and simplest solution that we had is the naive approach: parallelize everything. At every stage in the recursion tree, any parent which has to execute two or more children will spark computation for them in parallel.

Theoretically, if there was no overhead for the management of sparks, this is the best solution since the processors will be completely busy until all the work is finished. In practice, however, this strategy is wasteful and performs poorly.

Results



Time	Heap	GC	Spark stats	Spark sizes	Process info	Raw events	
HEC	Total		Converted	Overflowed	Dud	GC'd	Fizzled
Total	21964110	192	0	0	20784119	1179799	
HEC 0	5413450	74	0	0	5118645	294701	
HEC 1	5457810	59	0	0	5161002	296751	
HEC 2	5683952	38	0	0	5394274	289679	

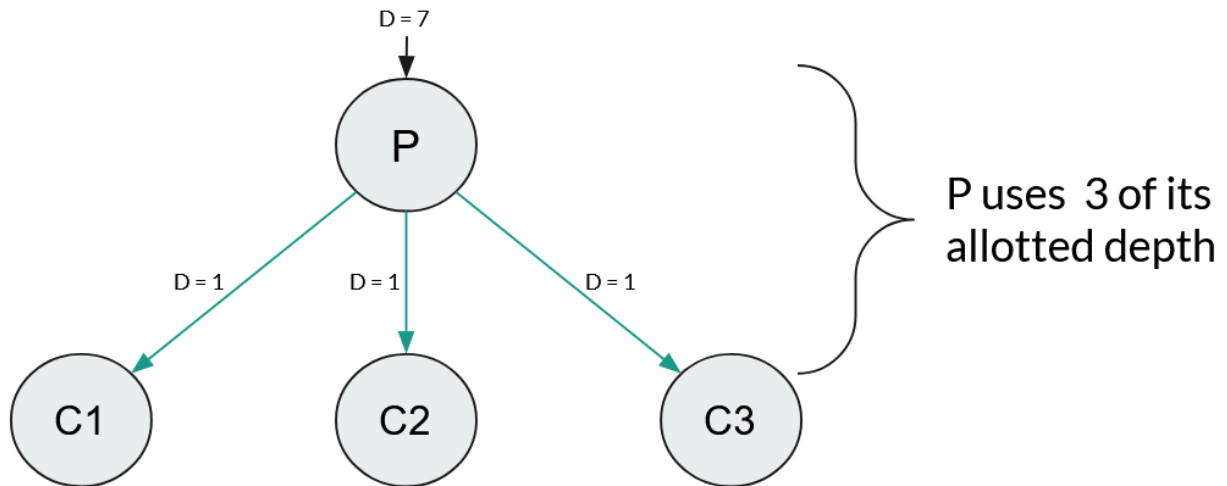
Type	Run Time	Speedup($\frac{Seq}{Par}$)
Seq	2m:49s	1.34x
Par (-N4)	2m:06s	

When tested on a 9x9 game board, as the results above indicate, out of almost 22 million sparks created, only 192 end up being converted. The CPU utilization is good however, since no CPU will be idle at any time as long as there are multiple problems being solved. In the end, however, the overhead dominated the runtime and the algorithm achieved only a 1.34x speedup from the sequential solution.

The Strictly Fair Parent

The problem with the naive strategy is that there are more sparks created that could ever be evaluated by the processors. To remedy this, we introduced the concept of *depth*. The depth is a parameter to a *solve* function call that indicates the total number of sparks that may be created by that call and all its children. If a call is given a positive depth, then it may use that depth and pass it to its children as it wishes. If a call is not given a positive depth, then it must execute itself and all its children sequentially.

The strictly fair parent strategy uses the depth it is given to spark computation for all of its children, and then equally divides what's left among the children it sparked. As an illustration, take the following scenario:

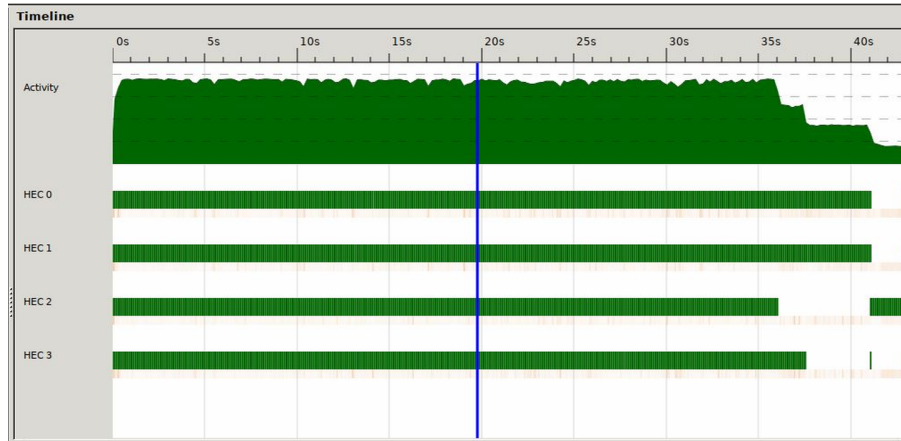


The parent is given a depth allotment of 7. Since it has 3 children, it sparks computation of the three children (using 3 of its 7 depth) and divides the remaining 4 equally among the children, which continue to do the same thing. The grandchildren, who will not be given any depth, must execute completely sequentially. Notice how the total number of sparks created by the sub-tree rooted at the parent is limited by the depth (with some wiggle-room). By controlling the depth passed to the first call, the total number of sparks created in the entire program can be controlled.

In our experiments and testing on the strictly fair parent strategy, we found one issue to be that the sub problems created by a parent may not need all the parallelizing power equally. For instance, in the above example, if C1 is immediately found to be unsolvable, it will never use the 1 depth value that it is passed, which C2 and C3 might have needed more. Because this strategy does not distinguish between the sub-problems, there can be imbalances created.

On average though, we found this strategy to be adequately performant and certainly an improvement on the naive approach.

Results



Time	Heap	GC	Spark stats	Spark sizes	Process info	Raw events
HEC	Total	Converted	Overflowed	Dud	GC'd	Fizzled
Total	124	13	0	0	2	109
HEC 0	30	5	0	0	1	25
HEC 1	20	4	0	0	0	17
HEC 2	36	2	0	0	1	34

Type	Run Time	Speedup($\frac{Seq}{Par}$)
Seq	1m:49s	2.53x
Par (-N4)	0m:43	

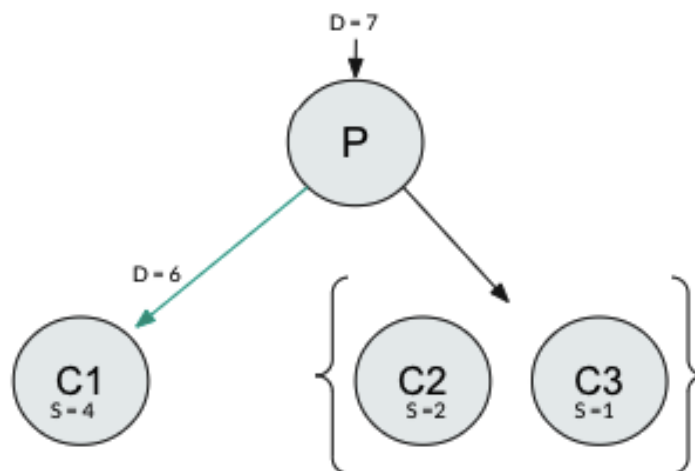
As seen in the results, there was a much greater utilization of sparks due to the depth limit, and the overall performance is also better.

The Forward-Thinking Parent

The problem with the strictly fair strategy is that the parent doesn't know which child will need the finite sparking power more than the others. Thus, the forward-thinking parent strategy tries to remedy this by computing the number of next possible moves that each child can make and

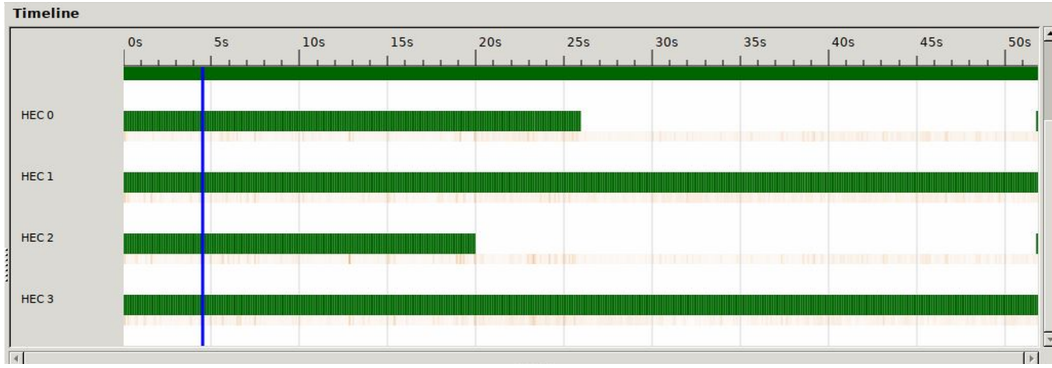
assigns the most sparking power (parent's sparking power - 1) to the child with the most possible next moves. The rest of the children will then evaluate sequentially. This selected child will always be the same across executions because our algorithm is deterministic given the same board.

The following graphic shows a parent P with a depth allotment of 7. The parent P calculates the number of next possible moves (represented by S) that each of its children can make. C1 can make 4 possible moves, C2 can make 2 possible moves and C3 can only make one possible move. C1 has the higher number of next possible moves out of all the children so P sparks computation of C1 and gives it depth 6. The remaining children C2 and C3 are not given anything and must evaluate sequentially.



In our experiments and testing on the forward-thinking parent strategy, it was found to be heavily board-dependent. For example, if C1 takes all the sparking power and ends up unsolvable a few moves later, then we've wasted all the parallel computation and the overall computation becomes sequential. This is even worse if we make a wrong selection early in the execution, which will make our program devolve to sequential execution.

Results



Time	Heap	GC	Spark stats	Spark sizes	Process info	Raw events
HEC	Total	Converted	Overflowed	Dud	GC'd	Fizzled
Total	14	4	0	0	1	9
HEC 0	11	2	0	0	1	9
HEC 1	2	1	0	0	0	0
HEC 2	1	0	0	0	0	0

Type	Run Time	Speedup($\frac{Seq}{Par}$)
Seq	1m:51s	2.27x
Par (-N4)	0m:49s	

Future Work

There are several ways that our current implementation can be further improved:

- We can return early if any subproblem finishes without waiting for the rest since one solution is all that is required.
- We could strengthen our searching algorithm by implementing more restrictive heuristics to narrow down the possible sub-moves at each position.
- We can add a more robust testing architecture both for correctness and easy comparisons for performance.
- The current lookahead strategy (the forward-thinking parent) does duplicate work because the parent calculates the next move for the children only for the children to do the same calculations so they could make their moves. This can be eliminated by the parent passing the work it did to the children so they can use that instead.
- We can devise a more balanced strategy that calculates the children's possibilities but then splits the depth among the children weighted on how much they are expected to need it

Resources

- Initial proposal:
<http://www.cs.columbia.edu/~sedwards/classes/2021/4995-fall/proposals/ParallelFlow.pdf>
- Github: <https://github.com/deji725/PFP-FlowSolver>
- Test Cases from https://github.com/mzucker/flow_solver/tree/master/puzzles

1 Main.hs

```
module Main where

import Lib

import System.Environment(getArgs)
import System.Exit(die)
import qualified Data.Vector as V
import qualified Data.Map.Strict as M
import qualified Data.Set as S

main :: IO ()
main = do
  args <- getArgs
  (filename, parallelize, depth) <-
    case args of
      [fn, p] -> return (fn,p,10) -- default depth of 10
      [fn, p, d] -> return (fn,p, read d :: Int)
      _ -> die "Usage: flow-solver [filename] [par|seq]"
  let solver = if parallelize == "par" then par_solver depth else seq_solver

  contents <- readFile filename
  let ls = lines contents
      matrix = V.fromList $ map V.fromList ls
      colors = S.delete '0' $ S.fromList $ concat ls
      ends = M.delete '0' $ getEnds matrix
      sol = solver matrix colors ends
  case sol of
    Nothing -> putStrLn "The board does not have a solution"
    Just s -> putStrLn $ myShow s
```

2 Naive Parent Strategy

```
module Lib
  where

import System.Environment(getArgs)
import System.Exit(die)
import Data.Maybe
import Data.List
import Data.Char(isUpper, toUpper)
import Data.Vector((!?),(!), (//))
import Control.Parallel.Strategies(using, parList, rseq)
import qualified Data.Vector as V
import qualified Data.Map.Strict as M
import qualified Data.Set as S

type Board = (V.Vector (V.Vector Char))
type Fronts = M.Map Char [Pos]
type Ends   = Fronts
type Pos    = (Int,Int)

myShow :: Board -> String
myShow board = concat $ intersperse "\n" $ V.toList $ V.map (V.toList) board

seq_solver :: Board -> S.Set Char -> M.Map Char [Pos] -> Maybe Board
seq_solver board colors ends = helper board ends
  where
    helper cur_board fronts
      | isSolved cur_board colors ends = Just cur_board
      | M.size nextMoves == 0 = Nothing
      | length moves == 0 = helper (makeMove cur_board best_pos best_pos)
        (M.delete best_char fronts)
      | otherwise = case filter isJust sub_sols of
          [] -> Nothing
          (s:_) -> s
    where
      nextMoves = getNextMoves cur_board fronts
      (best_pos@(i,j), moves) = getShortestMove nextMoves
      best_char = cur_board ! i ! j
      sub_problems = map (\nxt -> ((makeMove cur_board best_pos nxt),
        (advanceFront fronts best_char
          best_pos nxt))
        ) moves
      sub_sols = map (\(nxt_move, nxt_fronts) ->
        helper nxt_move nxt_fronts) sub_problems
```

```

par_solver :: Board -> S.Set Char -> M.Map Char [Pos] -> Maybe Board
par_solver board colors ends = helper board ends
  where
    helper cur_board fronts
      | isSolved cur_board colors ends = Just cur_board
      | M.size nextMoves == 0 = Nothing
      | length moves == 0 = helper (makeMove cur_board best_pos best_pos)
                            (M.delete best_char fronts)
      | otherwise = case filter isJust sub_sols of
                    [] -> Nothing
                    (s:_) -> s
    where
      nextMoves = getNextMoves cur_board fronts
      (best_pos@(i,j), moves) = getShortestMove nextMoves
      best_char = cur_board ! i ! j
      sub_problems = map (\nxt -> ((makeMove cur_board best_pos nxt),
                                (advanceFront fronts best_char
                                 best_pos nxt)))
                    moves
      sub_sols = map (\(nxt_move, nxt_fronts) ->
                    helper nxt_move nxt_fronts)
                sub_problems `using` parList rseq

advanceFront :: Fronts -> Char -> Pos -> Pos -> Fronts
advanceFront fronts c old new =
  M.insert c (new:(filter ((/=) old) $ fronts M.! c)) fronts

makeMove :: Board -> Pos -> Pos -> Board
makeMove board (b1,b2) (a1,a2) = replaceBoard (b1,b2) tmp (toUpper cur_char)
  where cur_char = board ! b1 ! b2
        replaceBoard (i,j) brd value = brd // [(i, brd ! i // [(j, value)]]
        tmp = replaceBoard (a1,a2) board cur_char

getShortestMove :: M.Map Pos [Pos] -> (Pos, [Pos])
getShortestMove all_moves =
  M.foldlWithKey getMinMoves ((0,0), replicate 5 (-1,-1)) all_moves
  where getMinMoves cur_min@(_, min_moves) cur_pos cur_moves =
        if length min_moves > length cur_moves then (cur_pos, cur_moves)
        else cur_min

-- Gets all the possible moves on the board

```



```

getNextMoves :: Board -> Fronts -> M.Map Pos [Pos]
getNextMoves board fronts =
  M.foldl helper M.empty fronts
  where getMoves (i,j) = map fst $ --getMoves :: Pos -> [(Pos,Char)]
    filter (\(_, ch) -> (ch == '0' || ch == cur_char) )
      $ neighbors_idxxs (i,j) board
    where cur_char = board ! i ! j
  helper m l = if all ((==) (head l)) l then M.insert (head l) [] m
    else foldl (\m' pos -> M.insert pos (getMoves pos) m') m l

isSolved :: Board -> S.Set Char -> M.Map Char [Pos] -> Bool
isSolved board colors ends
  | V.any (\v -> V.any (not . isUpper) v) board = False -- all places filled
  | otherwise = all color_has_path colors
  where color_has_path c = validPath board strt end
    where (strt:end:_) = ends M.! c

getEnds :: Board -> M.Map Char [Pos]
getEnds board = foldl (helper) M.empty [0.. (V.length $ board)-1]
  where helper boardMap i = foldl (helper2) boardMap
    [0.. V.length (board ! i) - 1]
    where helper2 m j =
      if is_end then
        M.insertWith (++) (board ! i ! j) [(i,j)] m
      else
        m
    where
      is_end = (length $ filter ((==) (board ! i !? j))
        (neighbors (i,j) board)) <= 1

validPath :: Board -> Pos -> Pos -> Bool
validPath board (i,j) end = helper (fst $ head first_step) (i,j)
  where
    cur_char = board ! i ! j
    first_step = filter (\(_,c) -> c == cur_char)
      (neighbors_idxxs (i,j) board)
    helper cur_pos p
      | cur_pos == end = True
      | length nextStep /= 1 = False
      | otherwise = helper (fst $ head nextStep) cur_pos
    where nextStep =
      filter (\(idx, c) -> (c == cur_char) && idx /= p)
        (neighbors_idxxs cur_pos board)

```

```

neighbors_idx :: Pos -> Board -> [(Pos, Char)]
neighbors_idx (i,j) board = map (\(p,m) -> (p, fromJust m)) $
    filter (\a -> isJust (snd a)) tmp
  where tmp = zip ([ (i, j-1), (i-1, j), (i,j+1), (i+1,j) ])
    (neighbors (i,j) board)

-- returns the neighbors of the color at (i,j)
neighbors :: Pos -> Board -> [Maybe Char]
neighbors (i,j) board = [left, up, right, down]
  where left = board ! i !? (j-1)
        right = board ! i !? (j+1)
        up = case board !? (i-1) of
            Nothing -> Nothing
            Just row -> row !? j
        down = case board !? (i+1) of
            Nothing -> Nothing
            Just row -> row !? j

```

3 Strictly Fair Parent Strategy

```
module Lib where

import System.Environment(getArgs)
import System.Exit(die)
import Data.Maybe
import Data.List
import Data.Char(isUpper, toUpper)
import Data.Vector((!?),(!), (//))
import Control.Parallel.Strategies(using, parList, rseq)
import qualified Data.Vector as V
import qualified Data.Map.Strict as M
import qualified Data.Set as S

type Board = (V.Vector (V.Vector Char))
type Fronts = M.Map Char [Pos]
type Ends   = Fronts
type Pos    = (Int,Int)

myShow :: Board -> String
myShow board = concat $ intersperse "\n" $ V.toList $ V.map (V.toList) board

seq_solver :: Board -> S.Set Char -> M.Map Char [Pos] -> Maybe Board
seq_solver board colors ends = helper board ends
  where
    helper cur_board fronts
      | isSolved cur_board colors ends = Just cur_board
      | M.size nextMoves == 0 = Nothing
      | length moves == 0 = helper (makeMove cur_board best_pos best_pos)
                              (M.delete best_char fronts)
      | otherwise = case filter isJust sub_sols of
          [] -> Nothing
          (s:_) -> s
    where
      nextMoves = getNextMoves cur_board fronts
      (best_pos@(i,j), moves) = getShortestMove nextMoves
      best_char = cur_board ! i ! j
      sub_problems = map (\nxt -> ((makeMove cur_board best_pos nxt),
                                (advanceFront fronts
                                 best_char best_pos nxt))
                        ) moves
      sub_sols = map (\(nxt_move, nxt_fronts) ->
                    helper nxt_move nxt_fronts) sub_problems

par_solver :: Int -> Board -> S.Set Char -> M.Map Char [Pos] -> Maybe Board
```

```

par_solver depth board colors ends = helper board ends depth
  where
    helper cur_board fronts dpth
      | isSolved cur_board colors ends = Just cur_board
      | length sub_problems == 0 = Nothing
      | length sub_problems == 1 = let (b,f) = head sub_problems
                                   in helper b f dpth
      | otherwise = case filter isJust sub_sols of
                    [] -> Nothing
                    (s:_) -> s
    where
      sub_problems = getSubProblems cur_board fronts
      sub_sols =
        if dpth > 0 then
          map helper_recurse sub_problems `using` parList rseq
        else
          map helper_recurse sub_problems
      helper_recurse (nxt_move,nxt_fronts) =
        helper nxt_move nxt_fronts (dpth `quot` (length sub_problems))

-- maybe give back the length of all possible moves in the board if we make this
-- move
getSubSubProblemsSize :: [(Board, Fronts)] -> [Int]
getSubSubProblemsSize sub_problems =
  map (\(b,f) -> foldl (\s l -> s + (length l)) 0 (getNextMoves b f) )
    sub_problems
  -- map (\(b,f) -> length £ getSubProblems b f) sub_problems

getSubProblems :: Board -> Fronts -> [(Board, Fronts)]
getSubProblems board fronts
  | length nextMoves == 0 = []
  | length moves == 0     = [(makeMove board best_pos best_pos,
                              M.delete best_char fronts)]
  | otherwise = map (\nxt -> ((makeMove board best_pos nxt),
                              (advanceFront fronts best_char best_pos nxt)) )
    moves
  where nextMoves = getNextMoves board fronts
        (best_pos@(i,j), moves) = getShortestMove nextMoves
        best_char = board ! i ! j

advanceFront :: Fronts -> Char -> Pos -> Pos -> Fronts
advanceFront fronts c old new =
  M.insert c (new:(filter ((/=) old) $ fronts M.! c)) fronts

```

```

makeMove :: Board -> Pos -> Pos -> Board
makeMove board (b1,b2) (a1,a2) = replaceBoard (b1,b2) tmp (toUpper cur_char)
  where cur_char = board ! b1 ! b2
        replaceBoard (i,j) brd value = brd // [(i, brd ! i // [(j, value)]]]
        tmp = replaceBoard (a1,a2) board cur_char

getShortestMove :: M.Map Pos [Pos] -> (Pos, [Pos])
getShortestMove all_moves =
  M.foldlWithKey getMinMoves ((0,0), replicate 5 (-1,-1)) all_moves
  where getMinMoves cur_min@(_, min_moves) cur_pos cur_moves =
        if length min_moves > length cur_moves then (cur_pos, cur_moves)
        else cur_min

-- Gets all the possible moves on the board
getNextMoves :: Board -> Fronts -> M.Map Pos [Pos]
getNextMoves board fronts =
  M.foldl helper M.empty fronts
  where getMoves (i,j) = map fst $ --getMoves :: Pos -> [(Pos,Char)]
        filter (\(_, ch) -> (ch == '0' || ch == cur_char) )
          $ neighbors_idxs (i,j) board
        where cur_char = board ! i ! j
        helper m l = if all ((==) (head l)) l then M.insert (head l) [] m
          else foldl (\m' pos -> M.insert pos (getMoves pos) m') m l

isSolved :: Board -> S.Set Char -> M.Map Char [Pos] -> Bool
isSolved board colors ends
  | V.any (\v -> V.any (not . isUpper) v) board = False -- all places filled
  | otherwise = all color_has_path colors
  where color_has_path c = validPath board strt end
        where (strt:end:_) = ends M.! c

getEnds :: Board -> M.Map Char [Pos]
getEnds board = foldl (helper) M.empty [0.. (V.length $ board)-1]
  where helper boardMap i = foldl (helper2) boardMap
        [0.. V.length (board ! i) - 1]
        where helper2 m j =
              if is_end then
                M.insertWith (++) (board ! i ! j) [(i,j)] m
              else
                m
        where
          is_end = (length $ filter ((==) (board ! i ! j))
                    (neighbors (i,j) board)) <= 1

```

```

validPath :: Board -> Pos -> Pos -> Bool
validPath board (i,j) end = helper (fst $ head first_step) (i,j)
  where
    cur_char = board ! i ! j
    first_step = filter (\(_,c) -> c == cur_char)
      (neighbors_idxes (i,j) board)
    helper cur_pos p
      | cur_pos == end = True
      | length nextStep /= 1 = False
      | otherwise = helper (fst $ head nextStep) cur_pos
    where nextStep =
      filter (\(idx, c) -> (c == cur_char) && idx /= p)
        (neighbors_idxes cur_pos board)

neighbors_idxes :: Pos -> Board -> [(Pos, Char)]
neighbors_idxes (i,j) board = map (\(p,m) -> (p, fromJust m)) $
  filter (\a -> isJust (snd a)) tmp
  where tmp =
    zip ([ (i, j-1), (i-1, j), (i,j+1), (i+1,j) ]) (neighbors (i,j) board)

-- returns the neighbors of the color at (i,j)
neighbors :: Pos -> Board -> [Maybe Char]
neighbors (i,j) board = [left, up, right, down]
  where left = board ! i !? (j-1)
        right = board ! i !? (j+1)
        up = case board !? (i-1) of
          Nothing -> Nothing
          Just row -> row !? j
        down = case board !? (i+1) of
          Nothing -> Nothing
          Just row -> row !? j

```

4 Forward Think Parent Strategy

```
module Lib where

import System.Environment(getArgs)
import System.Exit(die)
import Data.Maybe
import Data.List
import Data.Char(isUpper, toUpper)
import Data.Vector((!?),(!), (//))
import Control.Parallel.Strategies(using, parListNth, rseq)
import Data.Function(on)
import qualified Data.Vector as V
import qualified Data.Map.Strict as M
import qualified Data.Set as S

type Board = (V.Vector (V.Vector Char))
type Fronts = M.Map Char [Pos]
type Ends   = Fronts
type Pos    = (Int,Int)

myShow :: Board -> String
myShow board = concat $ intersperse "\n" $ V.toList $ V.map (V.toList) board

seq_solver :: Board -> S.Set Char -> M.Map Char [Pos] -> Maybe Board
seq_solver board colors ends = helper board ends
  where
    helper cur_board fronts
      | isSolved cur_board colors ends = Just cur_board
      | M.size nextMoves == 0 = Nothing
      | length moves == 0 = helper (makeMove cur_board best_pos best_pos)
                               (M.delete best_char fronts)
      | otherwise = case filter isJust sub_sols of
                    [] -> Nothing
                    (s:_) -> s
    where
      nextMoves = getNextMoves cur_board fronts
      (best_pos@(i,j), moves) = getShortestMove nextMoves
      best_char = cur_board ! i ! j
      sub_problems = map (\nxt -> ((makeMove cur_board best_pos nxt),
                                (advanceFront fronts best_char
                                 best_pos nxt))
                        ) moves
      sub_sols = map (\(nxt_move, nxt_fronts) ->
                    helper nxt_move nxt_fronts) sub_problems
```

```

par_solver :: Int -> Board -> S.Set Char -> M.Map Char [Pos] -> Maybe Board
par_solver depth board colors ends = helper board ends depth
  where
    helper cur_board fronts dpth
      | isSolved cur_board colors ends = Just cur_board
      | length sub_problems == 0 = Nothing
      | length sub_problems == 1 = let (b,f) = head sub_problems
                                   in helper b f dpth
      | otherwise = case filter isJust sub_sols of
                    [] -> Nothing
                    (s:_) -> s
    where
      sub_problems = getSubProblems cur_board fronts
      subSubProblemsSizes = getSubSubProblemsSize sub_problems
      -- numOfChildren :: [(Board,Front), Int, n]
      numOfChildren = sortBy (compare `on` (\(_,a,_)->a)) $
        zip3 sub_problems subSubProblemsSizes [1..]
      sub_sols =
        if dpth > 0 then
          map helper_recurse numOfChildren
            `using` parListNth ((length numOfChildren) - 1) rseq
        else
          map helper_recurse numOfChildren
      helper_recurse ((nxt_move,nxt_fronts),_,n) = helper nxt_move
        nxt_fronts
        (if n == length numOfChildren then dpth-1 else 0)

-- maybe give back the length of all possible moves in the board if we make this
-- move
getSubSubProblemsSize :: [(Board, Fronts)] -> [Int]
getSubSubProblemsSize sub_problems =
  map (\(b,f) -> foldl (\s l -> s + (length l)) 0 (getNextMoves b f) )
    sub_problems
  -- map (\(b,f) -> length £ getSubProblems b f) sub_problems

getSubProblems :: Board -> Fronts -> [(Board, Fronts)]
getSubProblems board fronts
  | length nextMoves == 0 = []
  | length moves == 0     = [(makeMove board best_pos best_pos,
                              M.delete best_char fronts)]
  | otherwise = map (\nxt -> ((makeMove board best_pos nxt),
                              (advanceFront fronts best_char best_pos nxt)) )
    moves
  where nextMoves = getNextMoves board fronts
        (best_pos@(i,j), moves) = getShortestMove nextMoves

```



```

best_char = board ! i ! j

advanceFront :: Fronts -> Char -> Pos -> Pos -> Fronts
advanceFront fronts c old new =
  M.insert c (new:(filter ((/=) old) $ fronts M.! c)) fronts

makeMove :: Board -> Pos -> Pos -> Board
makeMove board (b1,b2) (a1,a2) = replaceBoard (b1,b2) tmp (toUpper cur_char)
  where cur_char = board ! b1 ! b2
        replaceBoard (i,j) brd value = brd // [(i, brd ! i // [(j, value)]]
        tmp = replaceBoard (a1,a2) board cur_char

getShortestMove :: M.Map Pos [Pos] -> (Pos, [Pos])
getShortestMove all_moves =
  M.foldlWithKey getMinMoves ((0,0), replicate 5 (-1,-1)) all_moves
  where getMinMoves cur_min@(_, min_moves) cur_pos cur_moves =
        if length min_moves > length cur_moves then (cur_pos, cur_moves)
        else cur_min

-- Gets all the possible moves on the board
getNextMoves :: Board -> Fronts -> M.Map Pos [Pos]
getNextMoves board fronts =
  M.foldl helper M.empty fronts
  where getMoves (i,j) = map fst $ --getMoves :: Pos -> [(Pos,Char)]
        filter (\(_, ch) -> (ch == '0' || ch == cur_char) )
        $ neighbors_idx (i,j) board
        where cur_char = board ! i ! j
        helper m l = if all ((==) (head l)) l then M.insert (head l) [] m
        else foldl (\m' pos -> M.insert pos (getMoves pos) m') m l

isSolved :: Board -> S.Set Char -> M.Map Char [Pos] -> Bool
isSolved board colors ends
  | V.any (\v -> V.any (not . isUpper) v) board = False -- all places filled
  | otherwise = all color_has_path colors
  where color_has_path c = validPath board strt end
        where (strt:end:_) = ends M.! c

getEnds :: Board -> M.Map Char [Pos]
getEnds board = foldl (helper) M.empty [0.. (V.length $ board)-1]
  where helper boardMap i = foldl (helper2) boardMap
        [0.. V.length (board ! i) - 1]
        where helper2 m j =
              if is_end then

```

```

        M.insertWith (++) (board ! i ! j) [(i,j)] m
    else
        m
    where
        is_end = (length $ filter ((==) (board ! i !? j))
                    (neighbors (i,j) board)) <= 1

validPath :: Board -> Pos -> Pos -> Bool
validPath board (i,j) end = helper (fst $ head first_step) (i,j)
    where
        cur_char = board ! i ! j
        first_step = filter (\(_,c) -> c == cur_char)
                        (neighbors_idx (i,j) board)
        helper cur_pos p
            | cur_pos == end = True
            | length nextStep /= 1 = False
            | otherwise = helper (fst $ head nextStep) cur_pos
        where nextStep =
            filter (\(idx, c) -> (c == cur_char) && idx /= p)
                (neighbors_idx cur_pos board)

neighbors_idx :: Pos -> Board -> [(Pos, Char)]
neighbors_idx (i,j) board = map (\(p,m) -> (p, fromJust m)) $
    filter (\a -> isJust (snd a)) tmp
    where tmp = zip ([ (i, j-1), (i-1, j), (i,j+1), (i+1,j) ])
                    (neighbors (i,j) board)

-- returns the neighbors of the color at (i,j)
neighbors :: Pos -> Board -> [Maybe Char]
neighbors (i,j) board = [left, up, right, down]
    where left = board ! i !? (j-1)
          right = board ! i !? (j+1)
          up = case board !? (i-1) of
                Nothing -> Nothing
                Just row -> row !? j
          down = case board !? (i+1) of
                Nothing -> Nothing
                Just row -> row !? j

```