Crossword Generator and Solver

Gustaf Ahdritz (gwa2107) and Lucie le Blanc (ll3163)

23 December 2020

1 Introduction and Overview

This project creates crossword grids and solves them using a user-specified dictionary. There are no clues or pre-filled letters involved: the solver produces all valid solutions for the crossword based on its dimensions and layout.

2 Crossword generation

The crossword grid generation loosely follows the New York Times’ crossword construction constraints:

1. Crosswords must have black square symmetry, which typically comes in the form of 180-degree rotational symmetry;
2. Crosswords must have all-over interlock;
3. Crosswords must not have unchecked squares (i.e., all letters must be found in both Across and Down answers);
4. All answers must be at least 3 letters long;
5. Black squares should be used in moderation.

Constraints 1 and 3 are fully implemented: our crosswords have 180-degree rotational symmetry, and all white squares are part of both horizontal and vertical entries. We implemented constraints 4 and 5 by allowing our crossword generator to take in a user-supplied minimum word length and target black square density. The rotational symmetry constraint made it very difficult to generate crosswords with odd dimensions that didn’t violate the minimum word length, so we set our crossword generator to only generate grids with even dimensions. Our strategy also means that it is possible to generate grids with disconnected regions, which breaks constraint 2. See below for one of the example grids we generated, using a size of 10 and a black square density of 0.2.

1https://www.nytimes.com/puzzles/submissions/crossword
3 Solving strategy

The solving strategy for the crossword is straightforward. Using a user-supplied dictionary, solution candidates are generated by filling each row of the grid with possible word combinations. The validity of the candidate is checked column-wise. Valid solutions are counted and printed.

Internally, grids are represented as two-dimensional lists of Square. Filling the crossword involves using list comprehensions to generate all possible solutions for each word, row, and grid. Checking the crossword is done by transposing each solution candidate and checking each column-turned-row to make sure it only contains sequences of letters found in the dictionary.

4 Dictionary selection

In our initial attempts, the dictionary was stored as a Map Int (Set String), where integer keys representing word length mapped to sets of words with that number of letters. This was later changed to Map Int (Set [Square]), for reasons discussed later in this report. The Map/Set combination was chosen for efficient lookup.

The standard dictionary, located in /usr/share/dict/words, contains 102774 words. 63357 of these contain no special characters or captial letters, making them usable in our crosswords. However, testing with a dictionary this large proved to be too difficult – every machine we tested on, including a CLIC machine called Denhaag that boasts 125G of memory, failed to get past even the first layer of candidates for a 4x4 grid before getting killed. The solutions it did generate were often quite interesting (see Figure 1).
We instead chose to test smaller dictionaries of varying sizes. We started with a dictionary of 10,000 most common English words, and trimmed the list by taking only the first 1000, 500, 200 or 100 words for different tests.

In terms of crossword filling, this does not yield very interesting solutions. Words like “the” and “that” appear often, but would not be not easy to write creative clues for in a real puzzle. In the 1000-word case, filling a 4x4 grid generated 39 solutions, 37 of which were diagonally symmetrical (same words down as across). The other 2 of were each others’ transposes, and missed diagonal symmetry by only one letter.

A more practical crossword generator would prefer words toward the middle of the frequency-ordered word list. Ideally, crossword words should be known to the average person but not necessarily used in the average sentence.

5 Initial Parallelization Attempts

Unless specified otherwise, all results in this section are from tests run using a dictionary of 500 randomly selected common words on the following 4x4 crossword:

```
[ @ - - - ]
[ - - - - ]
[ - - - - ]
[ - - - @ ]
```

where underscores represent blank white spaces and ‘@’ represents a black square. While the solver can handle larger puzzles than this, increasing the size of the puzzle increases runtimes to the order of minutes rather than seconds, prohibiting analysis of event logs in Threadscope. In the following subsections, we will apply various parallelization techniques to the following (slightly squashed) code snippet:

2https://github.com/first20hours/google-10000-english
solve :: Map Int (Set String) -> [[Square]] -> [[[Square]]]
solve dict grid = do
    let cands = map transpose (fill dict grid)
    filter (check dict) cands

The third line yields a list of transposed candidate solutions. The fourth cross-references them against the provided dictionary to verify that the columns contain valid words.

All figures reported in this section represent an average of five trials on an otherwise lightly loaded system.

Run sequentially, the program finds all valid solutions in 11.07 seconds.

5.1 Naive parBuffer

Our first attempt at parallelization was simply to parallelize the checking, expecting each thread to generate and then check their respective candidates independently, as follows:

solve :: Map Int (Set String) -> [[Square]] -> [[[Square]]]
solve dict grid = do
    let cands = map transpose (fill dict grid)
    filter (check dict) cands `using` parBuffer 100 rdeepseq

We use parBuffer here rather than parList in an attempt to mitigate the extreme memory requirements of loading the whole list of candidates at once and rdeepseq to force complete evaluation of each candidate in each thread. In practice, this implementation failed to parallelize at all. Zero threads are created, and the overhead of adding a second core (on which some of the garbage collection seems to be running) slows the runtime to 14.44s. The Threadscope graph is included in Figure 3. The culprit here appears to be filter. Unlike map, whose weak head normal form representation would consist of a concatenation of one thunk for applying the check to the first element of the list and then another for completing the map operation on the remainder of the list, filter must evaluate each check on each element completely in order to determine the shape of
the list. This evaluation is performed sequentially before the parBuffer is even properly applied, completely negating the benefits of having multiple cores.

5.2 Parallelizing candidate creation

We can sidestep this problem by shifting the parallelization directly onto the code that generates candidate solutions:

```haskell
solve :: Map Int (Set String) -> [[Square]] -> [[[Square]]]
solve dict grid = do
  let l = fill dict grid
  let cands = map transpose l `using` parBuffer 100 rdeepseq
      filter (check dict) cands
```

The event log tells us that this change was successful: The workload here appears extremely well-balanced. However, since about 67% of the 11694774 sparks created either fizzle or are garbage collected, sequential garbage collection for excess sparks short enough not to be visible at this scale in Threadscope drives the runtime all the way up to 24.09s, most of which comes in the form of additional garbage collection: we have achieved a “speedup” of 0.46x. This indicates that the parallelization is far too granular, making it inefficient: each spark is performing much less work per candidate grid than its computational upkeep.

5.3 Parallelized candidate creation with chunking

Instead of parallelizing the output of map directly, we can group the candidates into chunks of more reasonable sizes and then hand off each one to be computed in parallel:

```haskell
solve :: Map Int (Set String) -> [[Square]] -> [[[Square]]]
solve dict grid = do
  let l = fill dict grid
  let m = map transpose l
```
let chunk n = Data.List.SplitchunksOf n
let cands = chunk 10000 m `using` parBuffer 100 rdeepseq
filter (check dict) $ concat cands

Figure 5: parBuffer applied to candidates w/ chunking, -N4

On this test, we see a 100% conversion rate: of all of the 19617 sparks created, none fail. The program spends much less time in the MUT phase of computation as a result, but the costs of garbage collection gives us our slowest runtime so far: 31.6s. Perturbing the chunk size to any of \{10, 50, 100, 500, 1000, 10000, 50000, 100000\} or changing the number of threads passed to parBuffer makes matters worse.

5.4 Parallelizing candidate creation & filtering

Instead of filtering candidates in a separate second step, we can check them at the same time:

solve :: Map Int (Set String) -> [[Square]] -> [[[Square]]]
solve dict grid = do
  let l = fill dict grid
  let m = map transpose l
  let f = filter (check dict)
  let chunk n = Data.List.SplitchunksOf n
  let chunks = chunk 20000 m
  concat $ map f chunks `using` parBuffer 100 rdeepseq

Note that we have heuristically increased the chunk size to 20000. This is a substantial improvement over the previous attempt; run on 4 cores. The
conversion rate for the 981 sparks created is 100%. The MUT phase of the program runs in just 3.57s. Unfortunately, the greater the number of cores, the greater the time spent on garbage collection. The total runtime on 4 cores is 16.36s—faster than in the previous section, but still not close to the 11.07s achieved sequentially.

Increasing or decreasing the core count, curiously enough, both have the effect of increasing the absolute time spent on garbage collection. The problem is only exacerbated on more complex tasks. When the size of the dictionary is increased to 1000, the parallel -N4 solver spends about 85 seconds in MUT and then another 285 in garbage collection, for a total of 370. The sequential solver using the same dictionary spends 242 seconds in MUT and then 52 in garbage collection, for a smaller total of 294.\(^3\)

Figure 6 shows that, like Marlow’s “kmeans_strat,” our program performs computations in staggered chunks separated by periods of protracted garbage collection. Though we can decreasing the width of each individual stop by decreasing the chunk size, the total time spent on garbage collection remains fairly constant. Unlike Marlow, for whom the suspected culprit was expensive I/O operations, we make a point of not printing anything during execution. On the other hand, an inspection of Threadscope’s “Raw events” tab reveals a number of heap overflows in individual threads, causing needless interruptions. We conclude that the issue must be related to inefficient memory constructions in the crossword source.

### 6 Speedup Attempts

We made several changes to our program in an attempt to fix the very large garbage collection overhead.

\(^3\)Due to their length, the 1000-word trials were only run once each
6.1 List Comprehension

One of our attempts to improve our solution involved changing how we generated solution candidates. Originally, we used a list comprehension and recursion when filling the grid.

```haskell
fill_crossword dict (l:ls) =
    [ line : rest | line <- fill_line dict l,
                  rest <- fill_crossword dict ls ]
```

At the suggestion of Prof. Edwards, we switched order of the list comprehension generators in the `fill_crossword` and `fill_line` functions. This has the effect of staggering memory-intensive recursive calls to `fill_crossword` and reducing the likelihood of expensive heap overflows.

```haskell
fill_crossword dict (l:ls) =
    [ line : rest | rest <- fill_crossword dict ls,
                   line <- fill_line dict l ]
```

As will be shown in section 7, this change led to a speedup for both solvers, particularly the parallel one.

Another attempt involved generating candidates without a list comprehension at all. Instead, we used the `sequence` function, which takes lists as a parameter and returns their Cartesian product, to generate all possible solution candidate combinations.

```haskell
fill_crossword dict grid = sequence result_grid
    where result_grid = Prelude.map (fill_line dict) grid
```

This modification led the program to consume even more memory and spend more time in garbage collection; we reverted it to the flipped list comprehension above.

6.2 Set and List Comparison

Another possible source of problems we identified was how words were being picked from the dictionary when building candidate grids. Instead of `Map Int (Set String)`, we changed the dictionary passed to the `fill` functions to be of type `Map Int [String]`, in order to avoid possible overhead from the `Set toList` function. This did not affect the speed or memory consumption, perhaps because the results of `Set toList` are already being efficiently cached.

6.3 Dictionary Word Storage

We attempted to speed up dictionary lookups by eliminating the conversion step from `String` to `Square` and back. This involved changing the dictionary type from `Map Int (Set String)` to `Map Int (Set Square)`. This did not end up changing much, but it was one of the few cases in which the parallel version ran slightly faster than the sequential version.
7 Improved Parallelization Results

After implementing these improvements and dramatically shrinking the size of the chunks being passed to our sparks (from 10000 → 50), we observe the results shown in Figure 7.

![Timeline](image.png)

**Figure 7: Parallelization strategy from 5.4 with improvements, -N4**

Note the consistently higher activity on the cores and the absence of gaps as wide as those in Figure 6. Since the chunk size is so much smaller than the previous test, we see a dramatically increased number of sparks: about 392,000. As in earlier tests, however, all but 16 GC’d sparks and 24 fizzles successfully converted. For the first time, we are able to observe a speedup relative to the sequential version. The optimizations in Section 6 also benefited the latter: it now runs the task in 10.70s. However, the parallel version is even faster, finishing in 9.05s, 7 seconds faster than in the previous test. This buys us a final speedup of about x1.18. Granted, though the two allocate a similar number of bytes over their lifetimes—about 40 billion for the sequential version and 36 for the parallel—the productivity of the sequential version is much higher. It spends 97.1% of its time in MUT where the parallel version spends just 24.5%, leaving apparent room for future optimizations.

8 Discussion

While we were able to achieve high degrees of parallelization, the memory requirements of our program overwhelmed most of the accompanying performance benefits. However, the performance benefits we did see underneath the garbage collection overhead lead us to believe that further improvements are still possible.
A Source Code

A.1 Main.hs

```hs
import Data.Map
import Data.Set
import Data.List
import Data.List.Split
import Control.Parallel.Strategies
import System.Random
import System.Environment (getArgs, getProgName)
import System.Exit (die)
import DictUtil
import SquareDefs
import GenerateGrid

fill_and_check_s :: Map Int (Set [Square]) -> [[Square]] -> [[[Square]]]
fill_and_check_s dict grid = do
    let grids = fill_crossword dict grid
    let candidates = Prelude.map transpose grids
    Prelude.filter (check_crossword dict) candidates

fill_and_check_p :: Map Int (Set [Square]) -> [[Square]] -> [[[Square]]]
fill_and_check_p dict grid = do
    let f = Prelude.filter (check_crossword dict)
    let grids = fill_crossword dict grid
    let m = Prelude.map transpose grids
    let c = Data.List.Split.chunksOf 50 m
    let p = parBuffer 100 rseq
    let sols = Prelude.map f c `using` p
    concat sols

main :: IO ()
main = do args <- getArgs
    case args of
        [dimS, mS, threshS, dictPath, deterministic, parallel] -> do
            seed <- case deterministic of
                "y" -> return $ mkStdGen 15
                _ -> newStdGen
            let dim = read dimS :: Int
            let m = read mS :: Int
            let thresh = read threshS :: Float
            let result = generateGrid dim m thresh seed
            let peeled = maybe [[[]]] (\x -> x) $ result
            mapM_ (putStrLn) $ Prelude.map show peeled
            dictStream <- readFile dictPath
            let dict = build_dict dictStream
```

10
filled <- case parallel of
    "y" -> return $ fill_and_check_p dict peeled
    _  -> return $ fill_and_check_s dict peeled
mapM_ (printGrid) filled
let num = show (length filled)
putStrLn $ "found " ++ num ++ " ways to fill the given grid"
_ -> do pn <- getProgName
die $ "Usage: "++pn++" <dimension> <min_word_length>\"<probability_threshold>\"<dict_filepath>\"<deterministic>\"<parallel>"

A.2  SquareDefs.hs

module SquareDefs where
import Control.DeepSeq
data Square = Black | White Char
instance Show Square where
    show Black = "@"
    show (White c) = [c]
instance Eq Square where
    (==) Black Black = True
    (==) (White a) (White b) = (a == b)
    (==) _ _ = False
instance NFData Square where
    rnf s = s `seq` ()
instance Ord Square where
    Black `compare` Black = EQ
    (White a) `compare` (White b) = (a `compare` b)
    Black `compare` _ = GT
    _ `compare` Black = LT

A.3  DictUtil.hs

module DictUtil where
import Data.Set
import Data.Map
import Data.Char
import Data.List
import SquareDefs

blockedSq :: Char
blockedSq = 'X'

set_unwrap :: Maybe (Set [Square]) -> Set [Square]
set_unwrap (Just x) = x
set_unwrap Nothing = Data.Set.empty

getWordSet :: Map Int (Set [Square]) -> Int -> Set [Square]
getWordSet dict len = set_unwrap $ Data.Map.lookup len dict

pick_word :: Set [Square] -> [[Square]]
pick_word dict_slice = Data.Set.toList dict_slice

fill_word :: Map Int (Set [Square]) -> [Square] -> [[Square]]
fill_word _ [] = [[]]
fill_word dict single_word = pick_word candidateSet
  where candidateSet = getWordSet dict $ length single_word

fill_line :: Map Int (Set [Square]) -> [Square] -> [[Square]]
fill_line _ [] = [[]]
fill_line dict xs = [ w ++ blocked ++ r | w <- fill_word dict firstWord,
  r <- fill_line dict restOfLine ]
  where (word, suffix) = break isBlocked xs
  (blocked, restOfLine) = span isBlocked suffix
  isBlocked Black = True
  isBlocked (White _) = False

fill_crossword :: Map Int (Set [Square]) -> [[Square]] -> [[[Square]]]
fill_crossword _ [] = [[]]
fill_crossword dict (l:ls) = [ c : r | c <- fill_crossword dict ls,
  r <- fill_line dict l ]

-- An alternative fill_crossword. In our tests, it did not perform as well
{- sequence result_grid
  where result_grid = Prelude.map (fill_line dict) unfilled_grid -}

check_word :: Map Int (Set [Square]) -> [Square] -> Bool
check_word _ [] = True
check_word dict single_word = Data.Set.member single_word candidateSet
  where candidateSet = getWordSet dict $ length single_word
52 check_line :: Map Int (Set [Square]) -> [Square] -> Bool
53 check_line _ [] = True
54 check_line dict xs = check_word dict firstWord && check_line dict restOfLine
55   where (firstWord, suffix) = break isBlocked xs
56     (_, restOfLine) = span isBlocked suffix
57     isBlocked Black = True
58     isBlocked (White _) = False
59
60 check_crossword :: Map Int (Set [Square]) -> [[Square]] -> Bool
61 check_crossword dict grid = Prelude.foldl (&&) True m
62   where m = Prelude.map (check_line dict) grid
63
64 check_crosswords :: Map Int (Set [Square]) -> [[[Square]]] -> Bool
65 check_crosswords dict grids = Prelude.foldl (&&) True m
66   where m = Prelude.map (check_crossword dict) grids
67
68 valid_dict_word :: String -> Bool
69 valid_dict_word w = Prelude.foldl (&&) True $ Prelude.map isAcceptable w
70   where isAcceptable c = isAscii c && isLower c
71
72 get_valid_words :: String -> [String]
73 get_valid_words streamIn = Prelude.filter valid_dict_word $ words streamIn
74
75 insert_word :: Map Int (Set [Square]) -> String -> Set [Square]
76 insert_word dict single_word = Data.Set.insert squarified_word len_set
77   where len_set = getWordSet dict $ length single_word
78     squarified_word = Prelude.map toSquare single_word
79     toSquare c = White c
80
81 update_set :: Map Int (Set [Square]) -> String -> Map Int (Set [Square])
82 update_set dict single_word = Data.Map.insert word_len updated_set dict
83   where word_len = length single_word
84     updated_set = insert_word dict single_word
85
86 build_dict_rec :: Map Int (Set [Square]) -> [String] -> Map Int (Set [Square])
87 build_dict_rec dict [] = dict
88 build_dict_rec dict (w:ws) = update_set (build_dict_rec dict ws) w
89
90 build_dict :: String -> Map Int (Set [Square])
91 build_dict streamIn = build_dict_rec Data.Map.empty valid_words
92   where valid_words = get_valid_words streamIn
93
94 fill_and_check :: Map Int (Set [Square]) -> [[Square]] -> [[[Square]]]
95 fill_and_check dict emptyGrid = Prelude.filter (check_crossword dict) candidates
96   where candidates = Prelude.map transpose f
97     f = fill_crossword dict emptyGrid
A.4 GenerateGrid.hs

```haskell
module GenerateGrid where

import Data.List (sortBy)
import Data.Map.Strict (Map, (!))
import qualified Data.Map.Strict as Map
import Data.Set (Set)
import qualified Data.Set as Set
import System.Random
import SquareDefs

type Index = (Int, Int)

probs :: RandomGen g => Int -> g -> [Float]
probs 0 _ = []
probs n s = r : probs (n-1) s'
  where
    (r, s') = randomR (0.0::Float, 1.0::Float) s

indices :: Int -> Int -> Set Index
indices h w = Set.fromList $ \[(i,j) | i <- [0..(h - 1)], j <- [0..(w - 1)]\]

initConstraints :: Int -> Int -> Constraints
initConstraints h w = Map.fromSet (\_ -> (0, w - 1, 0, h - 1)) (indices h w)

updateConst :: Index -> Index -> Constraint -> Constraint
updateConst (bi, bj) (i, j) (l,r,t,b) = (nl, nr, nt, nb)
  where
    (nl, nr) | not sameRow = (l, r)
    | left = (max l bj + 1, r)
    | right = (l, min r bj - 1)
    | otherwise = (i, j)
    (nt, nb) | not sameCol = (t,b)
    | above = (max t bi + 1, b)
    | below = (t, min b bi - 1)
    | otherwise = (i, j)
    sameRow = bi == i
    (left, right) = (bj < j, bj > j)
    sameCol = bj == j
    (above, below) = (bi < i, bi > i)

adjustWithKeyList :: Ord k => (k -> a -> a) -> [k] -> Map k a -> Map k a
adjustWithKeyList _ [] m = m
adjustWithKeyList f (x:xs) m = Map.adjustKey f x $ adjustWithKeyList f xs m
```
addBS :: Index -> Constraints -> Constraints
addBS (i,j) con = updateRow $ updateCol con
  where
    updateCol c = adjustWithKeyList f col c
    updateRow c = adjustWithKeyList f row c
    f = updateConst (i,j)
    col = filter \((_,b) -> b == j\) keyList
    row = filter \((a,_) -> a == i\) keyList
    keyList = Map.keys con

checkSquare :: Index -> Constraints -> Int -> Bool
checkSquare (i,j) c m | i - top < m && i - top /= 0 = False
  | bot - i < m && bot - i /= 0 = False
  | j - left < m && j - left /= 0 = False
  | right - j < m && right - j /= 0 = False
  | otherwise = True
  where
    (left, right, top, bot) = c ! (i,j)

spiralIndices :: Int -> Int -> [Index]
spiralIndices 0 _ = []
spiralIndices 1 w = [(0, i) | i <- [0..(w - 1)]]
spiralIndices h w = (++) (f 0 0) $ adj $ spiralIndices (h - 2) (w - 2)
  where
    f i j = (i,j) : l i j
    l i j | j < w - 1 && i == 0 = f i (j + 1)
    | j == w - 1 && i < h - 1 = f (i + 1) j
    | j > 0 && i == h - 1 = f i (j - 1)
    | j == 0 && i > 1 = f (i - 1) j
    | otherwise = []
    adj spiral = map \((a, b) -> (a + 1, b + 1)\) spiral

fillIn :: [Index] -> [Bool] -> Constraints -> Int -> [Square]
fillIn [] _ _ _ = []
fillIn _ [] _ _ = []
fillIn (i:is) (b:bs) c m | not b = white
  | checkSquare i c m = black
  | otherwise = white
  where
    white = (White '_') : fillIn is bs c m
    black = Black : fillIn is bs (addBS i c) m

unravel :: Int -> [[(Index, Square)]] -> [[Square]]
unravel h s = map getRow [0..(h - 1)]
  where
    getRow i = map snd $ sortBy my_compare $ filter (isRow i) s
    my_compare ((_,y1),_) ((_,y2),_) = compare y1 y2
isRow i ((x, _), _) = x == i

generateGrid :: RandomGen g => Int -> Int -> Float -> g -> Maybe [[Square]]
generateGrid dim m thresh seed | even dim = Just even_grid
| otherwise = Nothing

where
even_grid = half ++ rev_half
rev_half = reverse $ map reverse half
half = unravel half_h $ zip s $ squares
squares = fillIn s b c m
s = spiralIndices half_h dim
c = initConstraints half_h dim
b = map (\x -> x <= thresh) p
p = probs (half_h * dim) seed
half_h = div dim 2

printGrid :: [[Square]] -> IO ()
printGrid grid = do mapM_ (putStrLn) $ map show grid
putStrLn ""